Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle

Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm, *University of Toronto*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao

This paper is included in the Proceedings of the
12th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Non-intrusive Performance Profiling for Entire Software Stacks based on the Flow Reconstruction Principle

Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, Michael Stumm
University of Toronto

Abstract

Understanding the performance behavior of distributed server stacks at scale is non-trivial. The servicing of just a single request can trigger numerous sub-requests across heterogeneous software components; and many similar requests are serviced concurrently and in parallel. When a user experiences poor performance, it is extremely difficult to identify the root cause, as well as the software components and machines that are the culprits.

This paper describes Stitch, a non-intrusive tool capable of profiling the performance of an entire distributed software stack solely using the unstructured logs output by heterogeneous software components. Stitch is substantially different from all prior related tools in that it is capable of constructing a system model of an entire software stack without building any domain knowledge into Stitch. Instead, it automatically reconstructs the extensive domain knowledge of the programmers who wrote the code; it does this by relying on the Flow Reconstruction Principle which states that programmers log events such that one can reliably reconstruct the execution flow a posteriori.

1 Introduction

Understanding the performance behavior of distributed server stacks at scale is non-trivial. Many incoming requests are serviced in parallel, and each such request may trigger multiple sub-requests on various software components spread out over many hosts. For example, a simple Hive query may involve a YARN Resource Manager, numerous YARN Application and Node Managers, several MapReduce tasks, and multiple HDFS servers.

Numerous tools have been developed to help identify performance anomalies and their root causes in these types of distributed systems. The tools have employed a variety of methods, all of which have significant limitations. Many methods require the target systems to be instrumented with dedicated code to collect information [1, 2, 9, 11, 24, 26, 27, 34]; as such, they are intrusive and often cannot be applied to legacy or third-party components. Other methods are non-intrusive and instead analyze already existing system logs; they either use machine learning approaches to identify anomalies [25, 38] or they rely on static code analysis [42]. Approaches that use machine learning techniques cannot understand the underlying system behavior and thus may not help identify the root cause of each anomaly. Approaches that require static code analysis are limited to components where such static analysis is even possible, and they are unable to understand the interactions between different software components.

We present a new approach for obtaining and presenting information useful for identifying performance issues and their causes in large distributed server stacks. Our method focuses on objects, their interactions, and their hierarchical relationships. Using pattern matching techniques on existing logs alone, we are able to identify which objects are participants in each logged event, which objects are of the same type, and how objects of different types relate to one another, allowing one to infer execution structure. Our method is non-intrusive and does not require static code analysis, yet enables complex cross-component performance analysis regardless of the programming languages used.

We have created a tool that extracts information from standard logs and is capable of visually displaying individual objects over their lifetimes showing how and when objects interact with each other. The gathered information on object relationships allows our tool to initially display high-level object instances (e.g., a Hive query) that can then be drilled down to view lower-level instances (e.g., HDFS blocks, MapReduce tasks, or containers). This hierarchical approach to displaying information is critical given the overwhelming number of events captured in logs and the number of objects involved. With a set of realistic sample scenarios, we show in Section 5 that such a tool can be useful for identifying performance anomalies and their root causes. Section 2 provides a motivating example.

This paper makes the following specific contributions:

- Our approach is the first to be able to construct a system model of an entire software stack without any built-in domain knowledge.
- We propose the first non-intrusive method that enables diagnosis of complex cross-component failures.
- Our method focuses on objects, their relationships and interactions as a way to deal with complexity (as opposed to focusing on events). Complexity is managed, for example, by initially displaying only high-level objects until a user decides to drill down on target objects.

Our approach has two limitations. Firstly, it is not able to directly identify causal relationships between events. However, we show that having this information is not necessary to debug complicated performance issues. Secondly, the efficacy of our approach relies on the quality of logging in the analyzed system. Specifically, our approach relies on the **Flow Reconstruction Principle**:

> programmers will output sufficient information to logs so as to be able to reconstruct runtime execution flows after the fact;

and more specifically:

> programmers will insert, for each important event, a log printing statement that outputs the identifiers[1] of all relevant objects involved with the event in order to be able to reconstruct execution flows a posteriori.

Inserting such log statements is a widely followed practice. As we show in Section 5, Hive/Hadoop, Spark, OpenStack, and even syslog [37] logs exhibit these properties. Many object identifiers, like process ID and thread ID, are automatically output by the underlying logging libraries for each event [21, 30, 35, 37].

The Flow Reconstruction Principle applies because a post mortem analysis is typically performed after each failure and the programmer will be asked to reconstruct what exactly transpired up to the point of failure in order to identify the root cause. This is non-trivial given the amount of concurrency and parallelism prevalent in scalable distributed server stacks. Hence, programmers will insert log statements to allow them to reconstruct how the failure occurred. Specifically they will:

- log a sufficient number of events — even at default logging verbosity — at critical points in the control path so as to enable a post mortem understanding of the control flow leading up to the failure.
- identify the objects involved in the event to help differentiate between log statements of concurrent/parallel homogeneous control flows. Note that this would not be possible when solely using constant strings. For example, if two concurrent processes, when opening a file, both output "opening file", without additional identifiers (e.g., process identifier) then one would not be able to attribute this type of event to either process.
- include a sufficient number of object identifiers in the same log statement to unambiguously identify the objects involved. Note that many identifiers are naturally ambiguous and need to be put into context in order to uniquely identify an object. For example a thread identifier (tid) needs to be interpreted in the context of a specific process, and a process identifier (pid) needs to be interpreted in the context of a specific host; hence the programmer will not typically output a tid alone, but always together with a pid and a hostname. If the identifiers are printed separately in multiple log statements (e.g., hostname and pid in one log statement and tid in a subsequent one) then a programmer can no longer reliably determine the context of each tid because a multi-threaded system can interleave multiple instances of these log entries.[2]

Programmers have a key advantage when using log messages to reconstruct the paths taken by the system when servicing requests: they can interpret the meaning of constant strings, and many identifiers contain string sequences that have meaning to the programmer. An automated tool does not have this advantage. As such, we disregard constant strings and do not attempt to extract semantics from object identifiers. Instead we extract information about objects by analyzing various patterns that exist in the logs. Our approach consists of the following five steps: (1) extract from the log messages the object identifiers; (2) associate each extracted object identifier with an object type; (3) identify how each object type is related to the other object types with respect to participating in the same event; (4) identify the specific object instances that are involved in each event; and (5) identify execution structure and hierarchy between objects. Using the extracted information, we are able to display the hierarchy of objects in play when servicing requests. We are also able to visually display objects along a timeline, as shown in the next section.

With the extracted hierarchy information, the visual display initially shows only the highest-level objects,

[1] In this paper, 'identifier' refers to the variable value that can be used to differentiate objects. Examples of identifiers include thread ID, process ID, file names, and host names. Examples of non-identifiers include the value of a counter or CPU usage statistics. Note that the counter itself is an object, but its value is not an identifier because it is not intended to be used to differentiate different counter instances.

[2] Note that the validity of this principle hinges on each log statement being thread safe. All of the widely used logging libraries we examined, including syslog [37], log4j [21], java.util.logging [16], SLF4J [35], log4cpp [20], Boost.log [5], and Python's logging module [30] are thread-safe.

but the user can selectively drill down incrementally by clicking on any of the objects of interest to expose more details at the lower-level. This enables identification of performance bottlenecks and analysis of potential root causes. A controlled user study showed that with Stitch, developers were able to speed up the analysis and debugging process by a factor of 4.6 compared to when they were restricted to using raw logs only. Our evaluation of how well Stitch is able to reconstruct a system model from logs showed that Stitch could do so with 96% accuracy when applied to Hive stack, Spark stack, and OpenStack logs produced by 200 nodes as well as logs from our production server stacks.

2 Motivating Example

We demonstrate the usefulness of Stitch using a real-world, user reported performance anomaly [31] that manifests itself across multiple software layers. Some Hive users reported that they occasionally experienced much longer than normal job completion times. We reproduced the anomaly on a three node cluster. We evaluated Stitch's effectiveness in a controlled user study comprising 14 experienced programmers, where half of the users were asked to debug with Stitch while the other half only had access to the raw logs. They were given 45 minutes. (Section 5.1 provides more details of the user study and other cases we used.)

On average, the users who did not use Stitch spent 38 minutes debugging this case whereas Stitch users only spent 12 minutes – a speedup factor of 3. In fact, six of the seven users not using Stitch failed to diagnose the anomaly whereas all seven Stitch users were successful. We first describe the users' experiences debugging without Stitch before discussing how Stitch helps.

Debugging without Stitch. Users in this control group immediately started grep'ping, awk'ing, and perl'ing the log files. Primarily, they took one of two approaches:

- "Bottom-up," where they searched for keywords such as "Error", "Warn", or "slow", or
- "Top-down," where they tried to understand the high-level hierarchy of the system before focusing on a particular lead.

The users who took a bottom-up approach almost entirely ended up in a wild-goose chase. For example, searching for "Error" and "Warn" returned log messages about non-zero container exit codes and failed container deallocations, neither of which pertained to the root cause. Three users stayed on this path with tunnel-vision, eventually reaching the time limit (45 minutes) without having gotten anywhere close to the root cause.

The three users who took a top-down approach got closer to finding the root cause, but none of them were able to identify it within the time limit. These users started by examining the latency of each Hive query in order to identify the slowest one. However, drilling down further proved to be daunting: users needed to follow the control path from Hive to the MapReduce layer to identify the map and reduce tasks created to process the query, then they had to carefully compare the timestamps of each task's first and last log message to identify the task-level bottleneck.

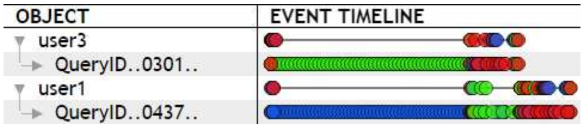

Figure 1: Stitch's timeline graph. Each line represents an object with its IDs listed in the left panel. Users can drill down to objects at the next level by clicking on the object. Each circle in the right panel represents an individual event, and its color indicates the node where the corresponding log message was output. Only two out of many queries are shown.

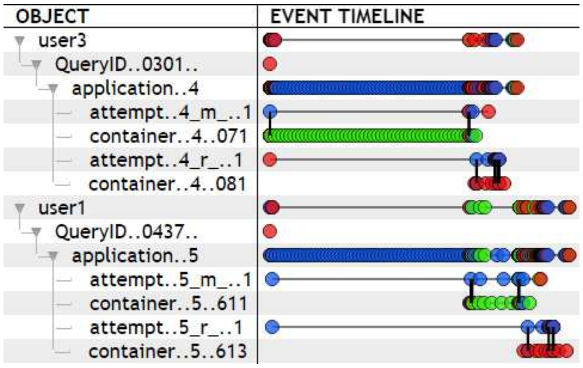

Figure 2: Stitch's timeline graph expanded to show more levels of the hierarchy. Clearly visible is that `user1`'s jobs start processing as soon as `user3` releases its containers. The vertical lines show the interactions among objects, which are inferred from the events that included multiple objects.

Even if the user could identify the slowest map/reduce task, the task's numerous interactions with other components makes it even more difficult to determine why it is slow. For example, a grep on particular (slow) map task may return over 500 log messages concerning its dealings with YARN containers, HDFS blocks, etc. Stitch serves to automate this process, freeing the user to focus on *debugging* rather than reconstructing the request.

Debugging with Stitch. Figure 1 shows Stitch's web-based GUI for this case. Objects are organized hierarchically allowing users to understand the system's structure as they drill down on each object. The event timeline shows the events where the object ID appeared such that the first and last event can be used to infer the object's lifetime. Thus, a user can immediately determine that Query "0437" has the longest execution time and drill down to investigate.

Figure 2 shows the interface following the expansion of both queries down to the level of map/reduce tasks. Under each query is the YARN application[3] created to process it, and under each application are the map/reduce task attempts and containers spawned to process the request. It shows that user1's Query 0437 has attempts that were created early on but only received containers much later. This allocation seems to correspond with the release of a container in user3's Query 0301, suggesting allocation was being serialized across users.

This serialization was caused by a bug in YARN's Capacity Scheduler which incorrectly limited all users' resource usage whenever a single user reached his or her limit. The study participants were not required to find this bug in the source since the study was limited to information that could be garnered from the logs alone. Instead they were only asked to point out that the serialized container allocation being the cause, which is conceivably the key step for the developers to diagnose the bug.

3 System Stack Structure Graph

In this section, we describe the algorithm Stitch uses to extract information from the logs necessary to identify objects, their interactions and their hierarchical relationships. The algorithm outputs a System Stack Structure (S^3) Graph – a directed acyclic graph (DAG), where each node represents an object type and each directed edge captures a hierarchical relationship between a high-level object type (parent) and a low-level object type (child).

We treat each logged *event e* as a set of *identifiers*, $id_{e1}..id_{en}$. We extract object identifiers by disregarding static substrings and applying a number of heuristics; e.g., we disregard variables preceded by common non-identifier types (e.g., "size") or succeeded by units (e.g., "ms"). How we extract IDs may seem somewhat ad hoc, but it works; ID parsing is considered a solved problem in the industry using tools such as Splunk [36], VMWare LogInsight [22] or Logstash [23]. Examples of IDs we extract this way include machine IP addresses, process IDs, thread IDs, and filenames. Note that the extracted IDs are often ambiguous until they obtain a context within which they can be interpreted; e.g., a process ID is unambiguous only if interpreted within the context of a specific host.

Each identifier is of a *type*, which is the type of the object it represents (e.g., a process, a thread, an IP address, a host name). In this section we represent identifiers in lowercase and their types in uppercase; e.g., both $host_1$ and $host_2$ are of type HOST. Stitch identifies the type by applying a number of heuristics that identify, for example, common static strings surrounding identifiers, common static substrings within identifiers, or the structure of the identifiers. Note that Stitch does not attempt to understand or identify what the actual type is (i.e., IP address, pid, filename, etc.), but simply differentiates between types abstractly (i.e., $TYPE_A$, $TYPE_B$, $TYPE_C$, etc.).

We say that two objects, obj_i and obj_j, are *correlated*, represented as $\mathrm{obj}_i \sim \mathrm{obj}_j$, if both objects were participants in the same logged event, meaning both of their IDs appeared in the same log message. We further define that obj_i *subsumes* obj_j, or $\mathrm{obj}_i \vdash \mathrm{obj}_j$, if and only if: (1) $\mathrm{obj}_i \sim \mathrm{obj}_j$, and (2) obj_j is not correlated with any other object of the same type as obj_i. For example, in Hive, user u_i subsumes query q_k because u_i will submit many different queries (including q_k), yet two queries with the same name will not typically be submitted by different users since each query is assigned a globally unique ID based on its timestamp and global order.

For all of the object types we identified, $T_{1..t}$, we categorize the relationship between each possible pair $(T_I, T_{J \neq I})$ as one of (*i*) empty, (*ii*) 1:1, (*iii*) 1:n, or (*iv*) m:n. We use this categorization to, in subsequent steps, help identify objects unambiguously and to identify the system stack object structure. The relationship is empty if object IDs of the two types never appear in the same log message. The relationship is 1:1, i.e., $T_I \equiv T_J$, if it is not empty, and $\forall\ \mathrm{obj}_i \in T_I, \forall \mathrm{obj}_j \in T_J, \mathrm{obj}_i \sim \mathrm{obj}_j \Rightarrow (\mathrm{obj}_i \vdash \mathrm{obj}_j) \wedge (\mathrm{obj}_j \vdash \mathrm{obj}_i)$; for example, IP_ADDR $\equiv$ HOST if there is no IP remapping. It is 1:n, i.e., $T_I \rightarrow T_J$, if it is not empty or 1:1, and $\forall \mathrm{obj}_i \in T_I, \forall \mathrm{obj}_j \in T_J, \mathrm{obj}_i \sim \mathrm{obj}_j \Rightarrow \mathrm{obj}_i \vdash \mathrm{obj}_j$. Finally, the relationship is m:n, i.e., $T_I \bowtie T_J$, if and only if $\exists \mathrm{obj}_i \in T_I, \exists \mathrm{obj}_j \in T_J$, s.t. $\mathrm{obj}_i \sim \mathrm{obj}_j$ while $\mathrm{obj}_i \nvdash \mathrm{obj}_j$ and $\mathrm{obj}_j \nvdash \mathrm{obj}_i$.

The size of the logs being used for the analysis needs to be sufficiently large for us to be able categorize relationships correctly. If the size is too small, then some of the type relationships might be miscategorized; e.g., (USER, QUERY) will be categorized as 1:1 instead of 1:n if the log spans the processing of only one query. In contrast, logs spanning too large a time frame may also cause miscategorizations, at least theoretically; e.g., (USER, QUERY) might be categorized as m:n if the query ID wraps around. However, mature distributed systems like Hadoop, Spark, and OpenStack use universally unique identifier (UUID) [18] libraries to assign key identifiers. Therefore, the likelihood of identifier reuse is extremely low [18].

We can intuitively draw certain conclusions about a pair of identifiers based on the relationship between their types. For example, two identifiers with types in a 1:1 relationship indicate that one might be able to use the two identifiers to identify an object interchangeably. Two

[3] Stitch identifies the YARN application ID as interchangeable with the MapReduce job ID.

identifiers with types in an m:n relationship suggests that their combination is required to unambiguously identify an object (as discussed below). Finally, two correlated objects with IDs of types in a 1:n relationship indicate a hierarchical relationship between the objects they represent; i.e., one likely created or forked the other.

To illustrate how the relationship between object types can be useful, consider the example log snippet in Figure 3, which is the slightly simplified log output when processing two Hive queries. Messages 2-25 are output when processing query_14 submitted by Hive. YARN assigns it to an application (app_14), which in turn spawns two map and two reduce task attempts. Each attempt is dispatched and executed in a container. After the map phase, each reduce attempt creates two fetchers. Each fetcher is a thread that shuffles output from map attempts. Messages 27-33 show events related to query_15 where map and reduce attempts fail and get reassigned to different containers. It also shows that the same container can be reused by multiple attempts.

Figure 4 shows the relationships between each pair of object *types*. We call this the *Type Relation graph*. We explain a few of the relationships. First, while a user can submit multiple queries, a query is always uniquely associated with a single user; hence USER → QUERY, a 1:n relationship. Further, the application ID (e.g., 14 in "app_14") is included as part of both the identifiers of the MapReduce attempts and the containers spawned by this application; hence APP → ATTEMPT_M, APP → ATTEMPT_R, and APP → CONTAINER. Note that Stitch is able to parse map attempts as being of a different type than reduce attempts because they have different "schemas" ("attempt(.*)_m_(.*)" versus "attempt(.*)_r_(.*)"). The details of our log parser will be discussed in Section 4. ATTEMPT_R has an m:n relationship with CONTAINER because a container can be reused by multiple attempts while an attempt can also be assigned to multiple containers, given container failures.

Algorithm 1 shows how we further identify objects unambiguously given that some of the identifiers used to refer to objects are ambiguous. The algorithm takes two inputs: the Type Relation graph and the entire set of EVENTS. Each event, E, is represented as a set of object types based on the IDs found in the event log message. For example, E might be {USER, QUERY, APP}, as extracted from line 2 in Figure 3. The algorithm converts the *Type Relation graph* into a *System Stack Structure graph*, or S^3 graph, in a sequence of steps. Each node in the graph represents an object *type* along with its *signature*. This signature is the set of object identifier types defined by the requirement that an ID of each type must be present in order to unambiguously identify the object.

We start by setting the signature of every node in the Type Relation graph to the type of the object. The algorithm then goes through the following steps:

```
1  Hive user1 login successfully
2  Hive user1 submits query_14 : app=app_14
3  RM Application app_14 is submitted
4  AM app_14 created task attempt14_m_0
5  AM app_14 created task attempt14_m_5
6  AM app_14 created task attempt14_r_0
7  AM app_14 created task attempt14_r_1
8  RM app_14 allocated container14_2
9  RM app_14 allocated container14_3
10 RM app_14 allocated container14_8
11 RM app_14 allocated container14_9
12 AM Dispatch attempt14_m_0 on container14_2
13 AM Dispatch attempt14_m_5 on container14_3
14 AM Dispatch attempt14_r_0 on container14_8
15 AM Dispatch attempt14_r_1 on container14_9
16 MR container14_8 creates thread fetcher1
17 MR container14_8 creates thread fetcher2
18 MR container14_9 creates thread fetcher1
19 MR container14_9 creates thread fetcher2
20 MR container14_8 fetcher1 shuffle attempt14_m_0
21 MR container14_8 fetcher2 shuffle attempt14_m_5
22 MR container14_9 fetcher1 shuffle attempt14_m_5
23 MR container14_9 fetcher2 shuffle attempt14_m_0
24 RM app_14 finished
25 Hive Ended query_14
26 .. ..
27 Hive user1 submits query_15 : app=app_15 app_16
28 AM attempt15_m_0 failed on container15_0
29 AM Reassign attempt15_m_0 on container15_7
30 AM Dispatch attempt15_r_1 on container15_8
31 AM Dispatch attempt15_r_2 on container15_8
32 AM attempt15_r_2 failed on container15_8
33 AM Reassign attempt15_r_2 on container15_9
```

Figure 3: Log snippet output by two Hive queries. The software component that output each log message is shown at the beginning of the line. RM and AM stand for YARN's Resource Manager and Application Manager, respectively. MR stands for MapReduce. Identifiers of objects of different types are shown in different colors.

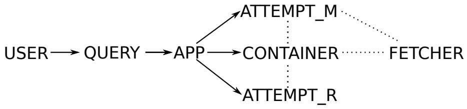

Figure 4: The Type Relation graph for the Hive log shown in Fig. 3. Each node is an object type. A solid arrow represents a 1:n relationship between the source and the destination object types; a dotted line represents an m:n relationship. There is no 1:1 relationship between types in the Hive log example.

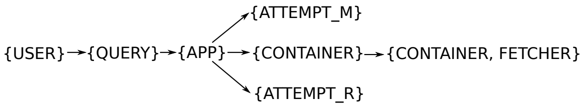

Figure 5: S^3 Graph of the Hive log snippet shown in Fig. 3.

Step 1: Merge 1:1 nodes. We first attempt to merge the nodes that are connected with ≡ edges. If two types have a 1:1 relationship, then the IDs of those types may often be used interchangeably to represent the same object. However, this is not always true. For example, YARN creates a unique url for each reduce task attempt so that

Algorithm 1: S^3 graph construction

```
   Input  : G: Type Relation Graph, EVENTS
   Output: System Stack Structure ($S^3$) Graph
   /* Step 1:  merge ≡ nodes in G        */
1  foreach connected component C of ≡ relations do
2      foreach subset S: {$T_1$,..$T_n$} in decreasing size do
3          if $\forall i \in [1,n], \forall obj_i \in T_i$,
           $\exists obj_1 \in T_1, ..obj_n \in T_n$ s.t.
           $obj_1 \equiv ..obj_i \equiv ..obj_n$ then
4              Node N ← merge($T_1, T_2, .., T_n$);
5              N.sig ← hash($T_1$.sig, $T_2$.sig, .. $T_n$.sig);
6              replace $T_i$.sig with N.sig in EVENTS;
7          end
8      end
9  end
10 remove any outstanding ≡ edges;
   /* Step 2:  process ⋈ relations       */
11 foreach connected component C of ⋈ relations do
12     S ← {all the types in C};
13     foreach $E \in EVENTS$ do
14         sig ← $E \cap S$;
15         if ∄ Node n where n.sig = sig then
16             nv ← new Node() with nv.sig ← sig;
17             if ∃ Node n' where n'.sig ⊂ nv.sig then
18                 add edge n' → nv;
19             end
20         else
21             mark n;
22         end
23     end
24     remove all unmarked nodes from C;
25 end
   /* Step 3:  filter non-object-types   */
26 foreach Node n do
27     if ∃Node n1,n2 s.t. n1.sig ∩ n2.sig = ∅ and
       n1.sig ∪ n2.sig ⊆ n.sig then
28         remove n;
29     end
30 end
```

a user can monitor the progress of this attempt in a web browser. Consequently, we infer ATTEMPT_R ≡ URL. However, URL is a generic type, and there can be other urls that are not related to any reduce attempt. For example, every job has its configuration information stored in an XML file that is referenced by a url. This XML file url does not appear together with any reduce attempt in any event. Therefore, we cannot say that URL and ATTEMPT_R may be used interchangeably. (Note that Stitch still infers ATTEMPT_R ≡ URL because for every pair of reduce attempt (att_i) and url (url_j) such that $\text{att}_i \sim \text{url}_j$, we have $\text{att}_i \vdash \text{url}_j$ and $\text{url}_j \vdash \text{att}_i$.)

Instead, we only merge those nodes $T_1, T_2, ..T_n$ in a ≡ connected component whose types can indeed be used interchangeably (line 3); i.e., when for any obj_i of type T_i, there exists obj_1 of type T_1, obj_2 of type T_2, ..., obj_n of type T_n such that $\text{obj}_1 \equiv \text{obj}_2 .. \equiv \text{obj}_n$, where $\text{obj}_i \equiv \text{obj}_j$ iff $\text{obj}_i \sim \text{obj}_j \wedge \text{obj}_i \vdash \text{obj}_j \wedge \text{obj}_j \vdash \text{obj}_i$. This prevents ATTEMPT_R and URL from being merged because there exist urls, such as the XML file url, that are not correlated with any reduce attempt. The fact that the types of the merged nodes can be used interchangeably indicates they are redundant. To merge $\{T_1, ..T_n\}$, we hash their signatures into a single value representing a new "type", and we replace every T_i in EVENTS with this hash value. After this, the outstanding ≡ edges, such as ATTEMPT_R ≡ URL, are removed as the types that are connected by them are not truly interchangeable.

Step 2: Process m:n nodes. In order to be able to identify objects unambiguously, we consider combining types with m:n relationships. The challenge is to determine which types should be combined. For example, "HOST", "PID", and "TID" (i.e., thread ID) have an m:n relationship between each pair. While {HOST}, {HOST,PID}, and {HOST,PID,TID} are meaningful combinations as they unambiguously identify hosts, processes, and threads respectively, the combination of {HOST,TID} is meaningless. To eliminate meaningless combinations, we consider all of the different combinations the programmers output in the log statements and only include the type combinations that appear in at least one log message. The reasoning is as follows: if a combination of identifiers is necessary to represent an object unambiguously, then the programmer will always output them together. A meaningless combination, such as {HOST,TID}, will likely never be found alone in a log message without "PID", so combinations such as these are discarded.

Therefore, for each ⋈-connected component, C, we only consider the type subsets where there exists an E ∈ EVENTS represented by a log message that contains exactly the types in this subset, but not any type in its complement set (line 11-25). A node whose type always appears with other types in the same C is removed at the end (line 24).

For the Type Relation graph shown in Figure 4, Step 2 creates four new nodes: {CONTAINER, FETCHER}, {CONTAINER, FETCHER, ATTEMPT_M}, {CONTAINER, ATTEMPT_R}, and {CONTAINER, ATTEMPT_M}. After creating the nodes, we further add 1:n edges from less constrained object types to more constrained object types (line 17-19). For example, a → edge will be added from node {CONTAINER} to {CONTAINER, FETCHER}.

Step 3: Filter non-objects. One should note that not every node created in the previous step is an actual object type in the system. Among the nodes that are created in Step 2 for the Hive example, only the one whose signature is {CONTAINER, FETCHER} represents a true object type, namely a fetcher thread in a container process. To filter out non-object types, Stitch removes nodes that are a combination of two existing object types. Hence, in our example, it would remove {CONTAINER, FETCHER, ATTEMPT_M}, {CONTAINER, ATTEMPT_R} and {CONTAINER, ATTEMPT_M} because they are combinations of other object types.

Figure 5 shows the S^3 Graph constructed by Stitch from the Hive log. This graph provides a simple model of the system. Each node is a type of object, and each edge represents a 1:n relationship, which indicates a hierarchical relationship, such as fork or creation.

Note that the S^3 Graph should be cycle free, because objects do not generally have a circular fork or creation pattern. If a cycle exists, then it must already have existed in the Type Relation graph since Algorithm 1 does not introduce any cycles. Therefore, we first run a cycle detection algorithm on the Type Relation graph. If a cycle is detected, every 1:n edge in the cycle is conservatively changed to an m:n edge. In our experimental evaluation, however, we never once encountered a cycle.

Generating a graph of object instances with their interactions, like the ones shown in Section 2, from an S^3 graph is merely a pattern matching appliction. Section 4.2 describes the algorithm in detail.

4 Implementation

Stitch uses a client-server model. A client runs on all hosts being monitored to: (1) locate active logs, (2) parse every log event into a set of object identifiers, and then (3) send the events to the server. A centralized server analyzes the events from all clients to build the S^3 graph and instantiates it with object instances.

4.1 Client Implementation

The client, implemented on Linux, runs as a daemon process. The interval between the times the client wakes up is called an *epoch*. Every time it wakes up, it scans the `/proc` file system to find all running processes. The client then examines each process' file descriptors to locate log files. We treat a file as a log if its type (determined by the file's magic number) is ASCII text and its name or a parent directory's name contains the text "log".

For each process with an open log file, the client tries to locate executables of the process (including dynamically linked libraries) by searching through its file descriptors and memory mapped files. For JVM processes, Stitch also searches the process' classpath for all .jar, .war, and .class files. This ensures that executables are found even if they were already closed by the JVM. Similarly, for Python processes, the client identifies the starting script from the shell command (e.g., ./script.py) and then uses Python's ModuleFinder package to locate the remaining scripts in the dependency graph, regardless of whether they are currently open.

Next, the client extracts all constant strings from each executable. For ELF executables, we extract constants from the read-only data segments (i.e., .rodata and .rodata1) by treating "\0" as a string terminator. For Java class files we extract the strings from each file's constant pool. For Python bytecode, we extract strings from the co_consts field in the Python code object. Currently, these are the only executable formats we support.

Parsing logs. The goal of log parsing is to extract the identifier values and infer their types from each log message. If an executable's constant string contains format specifiers, then this string can be directly used as a regular expression, where the specifiers are metacharacters (e.g., "%d" can be converted to "(\d+)" to extract an integer). We treat the pattern matched by a format specifier as a variable value.

However, most variable values output to log messages from Java, Scala, C++, and Python programs use string concatenation operators; e.g., "2016-04-02T00:58:48.734 MongoDB starting : pid=22925 port=27017 dbpath=/var/lib/mongodb" is printed by the following code snippet:

```
l << "MongoDB starting : pid=" << pid
  << " port=" << serverGlobalParams.port
  << " dbpath=" << storageGlobalParams.dbpath;
```

For these output methods, we use an approach general to all of the aforementioned languages: for each log message, any segment that matches a constant string is treated as static text, leaving only the variable values. In the example, "MongoDB starting : pid=", " port=", " dbpath=" are three of the constant strings parsed from MongoDB's executable, leaving "22925", "27017", and "/var/lib/mongodb" as variable values.

Stitch solves this string matching problem with a dynamic programming algorithm. Given a log string of length n, $L[0..n-1]$, let $M(i)$ be the maximum number of characters in $L[0..i]$ that are matched by constant strings. Our goal is to find the subset of constant strings that matches $M(n-1)$ characters of L in a non-overlapping manner. To compute this, we first define a function $match()$ as the following:

$$match(a,b) = \begin{cases} b-a+1 & \text{if } L[a..b] \text{ matches a constant} \\ 0 & \text{otherwise} \end{cases}$$

Now we can iteratively compute $M(i)$:

$$M(i) = \max\{match(0,i), \max_{0\leq j<i}\{M(j)+match(j+1,i)\}\}$$

This string matching is only necessary the first time a log message type is being parsed. In this example, after parsing the message, the client builds a regular expression: "MongoDB starting : pid=(\d+) port=(\d+) dbpath=(.*)". The next time if another message is printed by the same statement, Stitch can directly match it against the regular expression. We also use a heuristic to discard any string literals with fewer than three characters since the executables we evaluated often contained most permutations of all one and two character strings; using them would miscategorize identifier values as static text.

Next, Stitch infers the type of each variable. First, the client expands the variable to include characters within the word boundary delimited by whitespace. If the expansion includes static strings, then this "schema" of constant strings serves as the variable's type. For example, consider this Hadoop log message: "app_14 created task attempt14_r_0". Initially, the occurrences of "14" and "0" are recognized as variables, while "app_", " created task ", "attempt", and "_r_" are constant strings. Following expansion, the types of these two variables are "app_(\d+)" and "attempt(\d+)_r_(\d+)".

If a variable still does not include constant strings after the expansion, we trace backwards starting from the variable and use the first matched static text alphabetical word as the type. For example, in the MongoDB example, the three variables would have the types "pid", "port", and "dbpath" respectively.

Finally, the client uses a pair of heuristics to avoid capturing non-identifier variables. The first heuristic eliminates variables with types that do not end with a noun since intuitively, identifiers have noun-based types. For example, in the log, "Slow BlockReceiver write packet to mirror took 20ms", the latency variable is eliminated since the preceding static text, "took", is a verb. The next heuristic eliminates variables whose types are common non-identifiers (e.g., "size", "usage", "progress", etc.).

In practice, these heuristics did not eliminate all non-identifiers necessitating user-intervention. For example, "user" sometimes referred to a program's user-space execution time rather than a username. Rather than implementing heuristics for every such corner case, we allow the user to modify the generated regexes, which is only a one-time effort for each system.

Network protocol. At the end of each epoch, Stitch sends the parsed log messages from the last epoch to the server. The network protocol includes the following fields: (1) the timestamp of the epoch; and (2) a list of tuples, each with the format:

<severity, log file, {ID1:type1, ID2:type2, ..}, count>

All log messages from the same log file with the same set of identifiers and severity (e.g., INFO, WARN, etc.) are aggregated into a single tuple with the "count" field indicating the number of such log messages. This protocol message is then sent using Rsyslog [33] since communication is unidirectional.

The length of an epoch presents a trade-off between the timeliness of monitoring and the amount of network traffic. An epoch length of zero will force the client to stay awake and send parsed log messages one at a time; a large epoch will "compress" log messages that have the same set of identifiers within the epoch into a single tuple. Since log messages often arrive in a bursty manner, even a small epoch can significantly reduce network traffic. We set the epoch to be one second in our experimental evaluation.

4.2 Server Implementation

The Stitch server is also implemented as a daemon process. It consists of two threads: The first matches the stream of incoming events against the S^3 graph to generate an instantiated S^3 graph, henceforth called an S_i^3 graph. Each node in the S_i^3 graph is an object *instance*, whose signature is a set of identifier *values* instead of types as in the S^3 graph. The node also records the set of events that include the object instance.

For each event, e, we say e *instantiates* a node, N, from the S^3 graph if the set of identifier *types* in e is a superset of those in N's signature. For example, both events {app_14} and {app_14, attempt14_m_0} instantiate node {APP}. Initially, when no object instances have been created, for each incoming event, Stitch checks whether the event instantiates any of the *root* nodes in the S^3 graph. If so, Stitch creates an object instance node in the S_i^3 graph. For example, event {user1} will cause the creation of a node in S_i^3 graph, with signature {user1}.

Once an object instance node has been created in S_i^3 graph, for each incoming event, Stitch first checks if it matches any of the existing S_i^3 nodes. We say an event, e, matches a node, n, in the S_i^3 graph if e's identifier set is a superset of n's signature. If so, e is added to the event set of n. For each node, n, that e matches, Stitch further checks if e can instantiate any of the children of node, N, in the S^3 graph (where n is instantiated from N). If so, Stitch further instantiates the children of N and adds them as children of n. If one event matches multiple S_i^3 nodes that are not on the same path, a link is created between each node pair, indicating an interaction between them (Figure 2 in Section 2 showed an example of links represented as vertical lines between nodes).

Consider the S^3 graph for the Hive example shown in Figure 5. Figure 6 shows the S_i^3 graph Stitch generates after analyzing the first five log messages. The first message instantiates the node "user1". The second message matches the first node, instantiates a child node "query_14", and then further instantiates node "app_14"

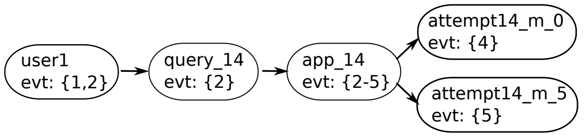

Figure 6: The S_i^3 graph generated from the first five log messages in Figure 3. Each node is an object instance. The events that include the object are also shown on each node.

Name	Software components	Benchmark
Hive	Hive, YARN, MapReduce, HDFS	HiBench
Spark	Spark, Spark SQL, Hive, HDFS	BigBench
OpenStack	Horizon, Glance, Nova, Keystone	VM cycle
Production	CRON, dbus, dhclient, dnsmasq, NetworkManager, ntpd, sshd, su, PostgreSQL	Common Linux ops

Table 1: The systems we used in evaluation.

as a child of node "query_14". The third message matches node "app_14", but does not instantiate any new nodes. The fourth and fifth messages match node "app_14" and then instantiate two nodes with signatures "attempt14_m_0" and "attempt14_m_5" respectively.

The server's second thread builds the S^3 graph, and incrementally updates it based on new patterns observed from incoming events. The thread first updates the Type Relation graph incrementally based on the observation that the relationship between two object types can only be updated in one direction: 1:1 → 1:n → m:n. Once the Type Relation graph is up to date, the server rebuilds the S^3 graph using Algorithm 1 and notifies the first thread so that it always uses the latest S^3 graph to build S_i^3 graph.

Visualization. Stitch's visualization front-end loads the S_i^3 graph as a JSON file and displays each node and its events as a row in a two-panel layout as shown in Figure 1. In order to scale to thousands of events per node, Stitch renders the graph using HTML5 Canvas and avoids drawing hidden elements where possible. For each node the user drills-down, Stitch performs a transitive reduction on the edges between this node and its children. Performing transitive reduction interactively like this avoids the overhead of doing it upfront since this is an $O(E)$ algorithm, where E is the number of edges.

5 Experimental Evaluation

We answer the following questions in evaluating Stitch: (1) How much time does Stitch save in profiling and debugging real-world systems? (2) Do real-world systems follow the Flow Reconstruction Principle? (3) How accurate is Stitch in identifying objects and their interactions? (4) What is the performance of Stitch?

We evaluated Stitch using both a controlled user study and lab experiments. The experiments consisted of using Stitch to monitor each system and workload listed in Table 1. In total there are 19 software components, programmed in C/C++, Java, Scala, and Python. The distributed system stacks were: (1) Hive stack – consisting of Hive, YARN, MapReduce, and HDFS – driven by the HiBench [15] workloads that repeatedly create, query, and delete tables; (2) Spark stack, with BigBench [12] used to send queries to Spark SQL, which then submitted them to a Spark cluster that read data from HDFS; and (3) OpenStack – consisting of four major software components – driven by our own workload generator. The generator used 80 concurrent processes to repeatedly create, boot, suspend/resume, pause/un-pause, and shutdown each VM at a randomized pace. Each distributed system workload was run for more than 24 hours on a cluster of 200 VMs.

Case	Description
OpenStack	Identify the hierarchy of components involved in creating VM instances.
MapReduce	Identify the bottleneck of a job when a reduce task was scheduled on a slow CPU.
YARN	Debug the anomaly described in Section 2.
Spark	Jobs slowed down by 500% because all but one HDFS datanode went down.
HDFS	Jobs slowed down due to a slow network link which affected HDFS read latency [29].

Table 2: Real-world profiling and debugging tasks we evaluate with Stitch. The first three are further used in our user study.

The production system was our own 24-node cluster, used for a variety of daily development workloads. Over the course of five months, Stitch identified and monitored nine log-printing components in the system, including job-scheduling, system-messaging, networking, session-management, privilege-management, and database utilities. Eight of these components logged using syslog.

5.1 User Study

Ultimately, the value of any debugging tool should be measured by the amount of time it saves. We evaluated this in a controlled user study on the first three debugging and profiling tasks listed in Table 2. They cover some real-world scenarios where Stitch can be used: (*i*) understanding the object hierarchy of a software stack ("OpenStack"); (*ii*) identifying bottlenecks to debug an anomaly we experienced in our production cluster that took us a long time to understand without Stitch ("MapReduce"); and (*iii*) debugging the anomaly described in Section 2 ("YARN"). We reproduced each case on a three nodes cluster. 14 individuals participated in the study including an experienced system administrator, professional developers, and both graduate and senior undergraduate students. All of them were experienced programmers familiar with Linux utilities such as grep, but their experience

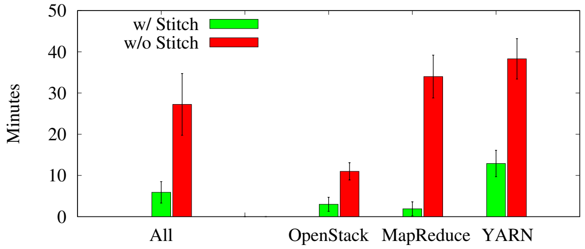

Figure 7: The result of Stitch's user study. The error bars show a 95% confidence interval.

with OpenStack and Hadoop varied, ranging from "no familiarity" to "expert developers". No co-author of this paper was a participant.

Each participant was given all three cases. Half of the participants were allowed to use Stitch for two cases but only logs (and Linux utilities) for the other case. The other half was assigned the opposite configuration. As a result, the collective expertise and experience was evenly spread across the different cases. Before the experiments, we gave each participant a five-minute verbal introduction to both OpenStack and Hadoop as well as a five-minute demo of Stitch. The demo was on a simple MapReduce job and served to familiarize users with the user interface. We then asked the participants to debug each case. Since the experiment is a single-blind trial (i.e., we, the co-authors, knew the answers), we gave the participants written instructions for each case and then minimized our interactions with them in order to avoid any potential influence. A 45 minute time limit was imposed on each case.

We marked a case as solved when the user identified the underlying cause to the degree that the raw logs allowed. A case was marked as unsolved if the user exceeded the 45 minute time limit. In the OpenStack case, success meant identifying the components involved in the creation of a VM instance. For the MapReduce case, it was to identify which job was slower and that the job was likely bottlenecked by the machine's slow CPU. For the YARN case, it was to identify that the longest query was bottlenecked waiting for a container to be released by another user's query.

Note that this is only a best effort experiment. The following potential threats to validity should be considered when interpreting the results. First, there is a potential bias in the selection of participants, as many of them were not familiar with the target systems. We note, however, that a comprehensive study [39] has shown many real-world debugging tasks are also performed by programmers unfamiliar with the target system. Another threat is that a study on just three cases may not be representative. We actually started to evaluate a fourth case ("HDFS" in Table 2) but decided to drop it after two trials (despite the initial results showing the largest time savings) because the participants became too exhausted given the complexity of debugging real-world systems without appropriate tools.

Figure 7 shows the results of user study. On average users spent 27.3 minutes without Stitch and 5.9 minutes with Stitch for a 4.6x speedup. An unpaired T-test shows that the hypothesis "Stitch lowers debugging time" is true with a probability of 99.99% (P value < 0.0001).

This measured speedup is likely an underestimate due to the 45-minute time limit we imposed. Among the 21 trials where Stitch was not provided, participants reached the time limit in eight of them without having found the issue. This suggests that debugging real-world distributed systems is indeed a complex task. In contrast, all of the Stitch users successfully diagnosed the case within the time limit.

Interestingly, participants using Stitch accidentally found an anomaly we were not targeting: when debugging the YARN case, some participants noticed that the map containers seemed to have long idle periods in the middle of their executions. It turned out that we had inserted a sleep call in map tasks so to reproduce the bug. Once we realized this, we removed the sleep call and instead used a larger input size to reproduce the anomaly.

Fundamentally, Stitch offers a "top-down" approach, whereas other practices, like "grep", are "bottom-up". While we show that Stitch is effective overall, the user study also showed that there are cases where a "bottom-up" approach is more effective. For example, when debugging the YARN case, a participant not using Stitch diagnosed the anomaly in 10 minutes because he spotted a log message clearly stating that a container could not be allocated. This suggests a desirable capability that would enable Stitch to *detect* some anomalies more quickly. While a comprehensive solution is beyond the scope of this paper, simple opportunities exist like highlighting events with "ERROR" or "FATAL" verbosity.

5.2 Other Real-world Cases

Table 2 shows two other real-world failures used to evaluate Stitch. The Spark failure was an anomaly we personally experienced. Stitch helped diagnose this failure by correlating objects along the path from a slow query all the way to the HDFS file read bottlenecking the query.

The HDFS case is a real-world issue [29] where the output generated at default verbosity does not provide sufficient information. At default verbosity, Stitch only allowed users to detect the fact that HDFS file access was slow and that a particular datanode was slow, but it was necessary to increase the logging level to allow Stitch to help identify a slow network link as the root cause; i.e., "DEBUG" logging was necessary to have the per packet latencies output to the log files.

	Log events		Identifiers		Objects			Edges		
	Instances	Types	Instances	Types	Instances	Types	Accuracy	Instance	Types	Accuracy
Hive	3,981,149	206	915,872	69	295,042	32	90%	401,924	73	87%
Spark	8,203,395	151	3,627,885	56	192,969	31	94%	485,133	60	93%
OpenStack	2,336,227	20	766,203	13	214,822	14	100%	2,196,315	19	100%
Production	312,779	36	123,668	22	8,141	24	100%	16,056	41	98%
Total	14,833,550	413	5,433,628	160	711,034	101	96%	3,099,428	193	95%

Table 3: Accuracy of Stitch's object reconstruction.

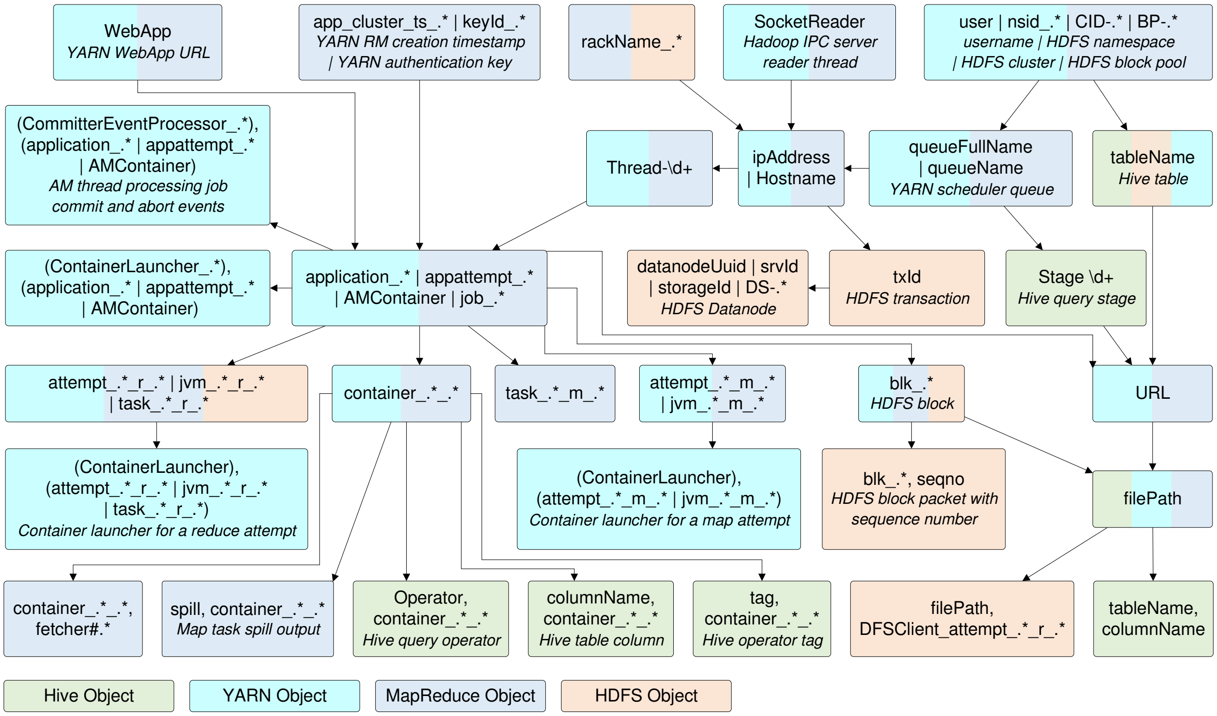

Figure 8: The entire, unfiltered System Stack Structure graph for the Hive stack after applying a transitive reduction. We explain the object type in italics when necessary. A '|' in the signature indicates that the identifier types can be used interchangeably, while a ',' indicates the types must appear together to unambiguously identify an object. The colors of each node indicate the components that output its object instance.

5.3 Object Reconstruction Accuracy

Table 3 shows Stitch's accuracy at identifying objects across four different system stacks. For each workload, we used Stitch to analyze the complete set of logs without performing sampling. The total number of instances is formidable: there are 14.8 million log messages, from which Stitch extracted 5.4 million identifier values to then infer 700,000 objects and 3.1 million edges. However, Table 3 also shows Stitch's power in reducing complexity by extracting the underlying system model. There are only 413 log event types, 160 identifier types, 101 object types, and 193 edges.

Figure 8 shows the entire, unfiltered S^3 graph inferred from the 4 million messages produced by the Hive stack. It clearly shows the hierarchical structure of the system.

For each system, we evaluated the precision of Stitch's object and hierarchy identification by carefully verifying every node and edge in the S^3 graph. Overall, 96% of the objects inferred by Stitch were accurate. Among the 101 object types Stitch identified, only four are incorrect. Three are from the Hive stack while the fourth is from Spark. The first one occurred because the "Thread-ID" in the Hive stack can represent threads from two components, namely YARN NodeManager (NM) and MapReduce ApplicationManager (AM). In AM, a thread ID always appears together with the AM container ID (container is a process), and the types of the corresponding objects have an m:n relationship. However, in NM a "Thread" ID can be printed by itself without a pid. This caused Stitch to infer an object type whose signature is "Thread" by itself. While this seemingly violates the Flow Reconstruction Principle, it turns out that there is only one NM process running on each node, therefore a thread ID by itself can still unambiguously identify a thread in this process. In fact, the NM's pid is never output to a log.

The second incorrect object type, "Stage", seemingly occurred due to a violation of the Flow Reconstruction Principle for a similar reason. Each stage of a Hive query has an identifier that is unique only within the context of a query. In practice however, a stage and query identifier are never printed in the same log message. This is because the Hive shell only runs a single query at a time, meaning the stages of two queries never interleave. As a result, Stitch incorrectly infers (based on the Flow Reconstruction Principle) that a stage identifier in multiple queries represents the same object.

The third incorrect object type occurred because of the Hive/Hadoop configuration we used. In Hadoop, "Socket_Reader" is the ID of a thread within the IPC Server process. The identifier is unique only within the context of an IPC Server process. Stitch only observes a 1:n relation between thread and IPC process (instead of the correct m:n relation) because, by default, the thread pool's size is set to one thread that has a fixed thread ID, yet there are multiple IPC Server processes.

The final incorrect object type occurred because of the limited scale of our Spark workload. Stitch inferred that IP address and inode can be used interchangeably (they have 1:1 relationship). However, they should have an m:n relationship since an HDFS inode is replicated onto multiple machines and each machine can host multiple HDFS inodes. This inaccuracy occurred because the two identifiers are only ever correlated in a rarely printed ERROR log message. The small scale of our workload did not output a large enough number of such errors to allow accurate inference of this relation.

Stitch achieves 100% accuracy on OpenStack because there are only 14 objects types and identifiers are automatically logged by the underlying logging library; each message is prepended with pid, user ID, project ID, etc.

The high accuracy on the production workload is also due to the underlying logging library in addition to a longer time window for analysis. Given that the production software was not distributed, we expected to find many messages with identifiers which were ambiguous across the cluster. Interestingly, this was not the case since by default, syslog automatically appends timestamp, hostname and pid to every log message. That said, analyzing just 24 hours of logs did result in poor accuracy since many relationships were 1:n instead of m:n. We had to extend the analysis time window until the model stabilized. Even then, one edge remained incorrect after over five months: only one user was using databases, thus making the relation 1:n rather than m:n. This highlights that the required log size for Stitch depends on how long it takes the system to observe all possible workload patterns.

Do real-world systems follow Flow Reconstruction Principle? Figure 9 shows the cumulative distribution of the number of objects included in each log message of the systems we evaluated. It shows that almost all of the log messages include at least one object (which also indicates that at least one identifier is included). This suggests that real-world systems are indeed following the Flow Reconstruction Principle in terms of logging identifiers. A primary reason for such a high percentage is that the underlying systems and logging libraries automatically include some identifiers. For example, by default OpenStack's logging framework prepends each message with pid, user ID, project ID, etc. In both Hive and Spark, the container and executor IDs are included in the file path of the container and executor logs respectively.

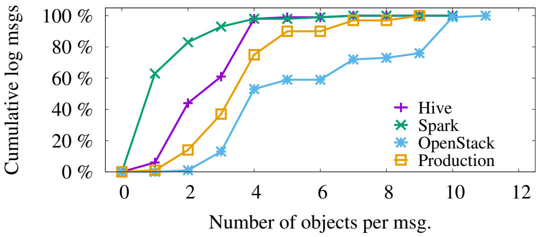

Figure 9: The cumulative distribution of the number of objects per log message.

Figure 9 also shows that over 50% of log messages contain more than one object. This suggests that objects interact in complex ways in real-world systems that further complicates debugging. This is where Stitch is particularly advantageous since it can (1) separate important fork or creation relationships from other interactions, and (2) identify which objects, at which time, interact with each other, and for how long.

We leave an evaluation of whether events are logged at critical points in the control flow as future work. However, we observe that most requests log the start and end of their execution, a clear indication of control flow.

5.4 Performance

To be viable in real-world systems, Stitch must meet two performance requirements: (*i*) it must process logs faster than they are generated; and (*ii*) it must not perturb the analyzed system's performance. We benchmarked Stitch's performance against these goals using the four previously mentioned software stacks.

Log processing. Stitch's client takes an average of 1.2 milliseconds in each one-second epoch to parse and send incoming log messages to the server. On average, the server takes 3.9 hours to process every 24 hours of log messages. Thus, both daemons are able to handle the rate of incoming log messages.

Performance perturbation. The client negligibly affects the workloads it monitors since it is CPU-bound.

Therefore, for I/O-bound jobs like the Hive workload, the client has no effect on execution time. The Spark workload can reach 100% CPU utilisation but this tends to happen in bursts. So again, the client has no effect on execution time. However, Stitch must manage the overhead of transmitting log messages over the network.

On average, Stitch achieves a 15% compression ratio compared to trasferring the complete log files for a workload. We automatically achieve 50% of this since we only need to transmit the identifiers from each log message. The other 35% comes from using Rsyslog's zlib compression on each of our protocol messages.

6 Limitation and Discussion

The efficacy of Stitch fundamentally depends on the quality of the logs. If there are no logs, or the log messages do not contain the right identifiers, then Stitch will not be able to accurately reconstruct the system model it needs. However, in that case, the logging would not even be useful for manual inspection by the developers who wrote the software. Perhaps our biggest (and happy) surprise from this project is the high quality and usefulness of the real-world logs generated by the mature distributed systems we evaluated. Nevertheless, we have also encountered real-world logs that did not contain sufficient information, as for example at one company where we tried to use Stitch (where log quality needs to be improved by enforcing the Flow Reconstruction Principle).

Another limitation of Stitch is that it cannot accurately infer causal dependencies. Instead, it can only identify correlations between objects. According to counterfactual theories of causation [19], event B causally depends on event A iff *an event B would not have occurred if event A had not occurred*. For Stitch, a causal dependence between events can be captured by two mechanisms. First, the 1:n relationships between objects can capture an object B that is created by an object A, so that their output events will have a causal dependency. Secondly, the events that are output along the same execution path involving the same object (i.e., a thread) are also captured since Stitch associates these events with that object. However, it is not true that every 1:n edge in Stitch's output captures object creation, or that every two events belonging to the same object have a control-flow dependency. We leave the job of distinguishing whether the correlation is causal or not to Stitch's user.

This limitation presents a fundamental trade-off in our identifier-only design since we ignore static text in log messages. For example, an event involving an operation on file B is causally dependent on an event involving process A if the latter event is "Process A created file B", but not if the event is only "Process A read file B". But this is something Stitch cannot identify. To capture this type of causal dependence, Stitch would need to either understand message semantics using Natural Language Processing or leverage a static analysis approach such as lprof [42]. [4]

Stitch's efficacy is also sensitive to the accuracy of extracting object identifiers and their types. In practice, it is trivial for developers or administrators to annotate the type of each identifier and its regular expression. In fact, projects like LogStash [23] already provide a database containing hundreds of regular expressions and their corresponding object types for commonly used identifiers.

7 Related Work

The fundamental challenge faced by any performance analysis tool for distributed systems is to capture the semantics of the event sequences output by the system. Existing solutions fall into one of three categories: (*i*) instrumenting the target system with predefined event semantics, (*ii*) use of static analysis to infer the system model, or (*iii*) the use of machine learning. We discuss each category in detail below, but note that Stitch does not rely on any of these techniques. Instead, it relies on programmers' intuition in event logging and builds a simple algorithm entirely based on identifiers from unstructured logs.

Intrusive approaches. Most of the existing tools capable of analyzing distributed system performance rely on instrumenting the target system [1, 2, 7, 8, 9, 10, 11, 13, 14, 24, 26, 27, 28, 32, 34]. The key benefit afforded by instrumentation is that the semantics of each event is defined by the analysis tool; therefore, the events can be unambiguously interpreted. For example, *ÜberTrace* [9] instruments Facebook's system with events that include a unique request ID and a predefined event name. Pivot Tracing [24] compiles a user's query into code that gets dynamically patched into the running system to collect tracing events; furthermore, critical identifier information is propagated to the different software components to be included in every event by the tracing code.

Stitch faces a different challenge, namely how to understand unstructured log events without any knowledge of how they are generated. This challenge presents interesting trade-offs between Stitch and the aforementioned intrusive approaches. On the one hand, because the semantics of events can be precisely interpreted, intrusive approaches can conduct more precise analyses. For example, Mystery Machine [9] and Pivot Tracing [24] can infer the *causal relationship* between events, a limitation

[4] However, it is debatable whether a log message "Process A created file B" really implies a causal relationship between process A and file B as it is possible that file B would have been created by other processes, regardless. Therefore, rigorously inferring causal dependencies between events may not be possible without domain expertise.

of Stitch we discussed in Section 6. On the other hand, the necessity of instrumentation imposes significant deployment hurdles. First, instrumentations are often specific to a particular system and cannot be generally applied to another. For example, Pivot Tracing's instrumentation requires dynamically patching code into system-specific data structures. This challenge is perhaps why, to the best of our knowledge, Stitch is the first system that has been evaluated on heterogeneous systems implemented in different languages. Second, vendors are often reluctant to modify their production systems in the fear that the instrumentations may open up reliability vulnerabilities and add performance overhead. Finally, it simply may not be possible to instrument legacy software or third-party components.

Log analysis guided by static analysis. Tools like lprof [42] and SherLog [41] are able to analyze a system's source and bytecode to infer how the processing of a request outputs log messages. For example, via control-flow analysis, lprof is able to infer that log messages are causally related if they are on the same execution path; its data-flow analysis can infer which variables are unchanged between two log printing statements, which can be used to group the logs. However, these approaches cannot be used to correlate events across heterogeneous software components because static analysis is limited to a single software component. In addition, static analysis needs to be customized for different languages (lprof only worked on Java bytecode), requires the source for all third-party libraries, and for languages like C and C++ where non-standard dialects are prevalent, it is challenging to even get the static analysis tool to compile the code [4].

Log analysis with machine learning. Several tools use machine learning on log files to detect anomalies [3, 25, 38, 40]. These tools first learn a model from logs output by correct runs, and then apply the model on logs from failure runs. For example, CloudSeer learns an automata model from clean logs that can then be used to detect anomalies [40]. Xu *et al.,* uses Principal Component Analysis (PCA) to detect unusual patterns in the logs [38]. Stitch is complementary to these tools: these tools' goals are to detect anomalies, while Stitch's goal is to profile entire systems by analyzing the performance behavior and relationships between every object in the system. Furthermore, the machine learning approaches could be used on Stitch's graphs to identify anomalies.

Log visualization tools. In recent years, most tools in this area (e.g., Splunk [36], VMware LogInsight [22], Kibana [17], etc.) have focused on system statistics and monitoring rather than breaking down the system model like Stitch. However, tools similar to Stitch exist in the area of file systems and web browsers. InProv [6] gives the user a top-down view of file system provenance data, allowing them to click through the hierarchy of interactions between processes and files over time. Chrome's Developer Tools allow a user to zoom in to and out of the hierarchy of objects on a web page. InProv, Chrome, and Stitch highlight the importance and usefulness of browsing hierarchy when debugging.

8 Concluding Remarks

This paper presented Stitch, the first system capable of reconstructing, in a non-intrusive manner, the end-to-end execution flow of requests being serviced by distributed server stacks. Without any system specific knowledge, Stitch is able to analyze unstructured log output from heterogeneous software components and construct a system model which captures the objects involved, their lifetimes and their hierarchical relationships.

At its core, Stitch is enabled by the observation that programmers follow certain principles so that they can reliably reconstruct the executions a posteriori. These principles, which we collectively refer to as the *Flow Reconstruction Principle*, state that a sufficient number of events need to be logged at critical points in the control path and that each event log output must include a sufficient number of object identifier values to be able to disambiguate the concurrent and homogeneous executions. While these principles may seem straightforward in hindsight, discovering them was challenging and took us over two years to identify and refine.

Acknowledgements

We greatly appreciate the insightful feedback from the anonymous reviewers and our shepherd, Margo Seltzer. We thank Xiang Ren, Yongle Zhang, David Lion, Serhei Makarov, and Abhishek Rudra for the useful and thought-stimulating discussions. We also thank all of the user study participants for their time spent debugging. The idea that identifiers can be parsed by excluding static text came from a discussion with David Lie. This research is supported by an NSERC Discovery grant, a NetApp Faculty Fellowship, and a Huawei grant.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89. ACM, 2003.

[2] 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*

Opearting Systems Design & Implementation, OSDI '04, pages 259–272. USENIX Association, 2004.

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277. ACM, 2011.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[5] Chapter 1. Boost.Log v2 - 1.61.0. `http://www.boost.org/doc/libs/1_61_0/libs/log/doc/html/index.html`.

[6] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, Dec. 2013.

[7] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 17–30. ACM, 2007.

[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604. IEEE Computer Society, 2002.

[9] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 217–231. USENIX Association, 2014.

[10] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197. ACM, 2015.

[11] 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.

[12] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208. ACM, 2013.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126. ACM, 1982.

[14] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '11, pages 299–312. USENIX Association, 2011.

[15] 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.

[16] java.util.logging (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html`.

[17] Kibana: Explore, visualize, discover data. `https://www.elastic.co/products/kibana`.

[18] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.

[19] D. L. Lewis. *Counterfactuals*. Blackwell Publishers, 1973.

[20] Log for C++ Project. `http://log4cpp.sourceforge.net/`.

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

[22] VMware vCenter Log Insight: Log management and analytics. `http://www.vmware.com/ca/en/products/vcenter-log-insight`.

[23] Logstash – normalizing varying schema. `https://www.elastic.co/products/logstash`.

[24] 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.

[25] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 353–366. USENIX Association, 2012.

[26] Nagios: the industry standard in IT infrastructure monitoring. `http://www.nagios.org/`.

[27] NewRelic: Application performance management and monitoring. `http://newrelic.com/`.

[28] OProf - A system profiler for Linux. `http://oprofile.sourceforge.net/`.

[29] Poor HDFS performances: Slow BlockReceiver write packet to mirror. `http://stackoverflow.com/questions/27984331`.

[30] Section 15.7 logging - Logging facility for Python - Python 2.7.12 documentation. `https://docs.python.org/2/library/logging.html`.

[31] JIRA: YARN bug 4610. https://issues.apache.org/jira/browse/YARN-4610.

[32] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation*, NSDI '06, pages 115–128. USENIX Association, 2006.

[33] RSYSLOG: the rocket-fast system for log processing. www.rsyslog.com.

[34] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[35] SLF4J. http://www.slf4j.org/.

[36] Splunk log management. http://www.splunk.com/view/log-management/SP-CAAAC6F.

[37] The syslog protocol. http://tools.ietf.org/html/rfc5424.

[38] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132. ACM, 2009.

[39] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36. ACM, 2011.

[40] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 489–502. ACM, 2016.

[41] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 143–154. ACM, 2010.

[42] X. Zhao, Y. Zhang, D. Lion, M. FaizanUllah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 629–644. USENIX Association, 2014.