Logging Library Migrations: A Case Study for the Apache Software Foundation Projects

CONFERENCE PAPER · MAY 2016

READS
6

4 AUTHORS, INCLUDING:

Cor-Paul Bezemer
Queen's University
14 PUBLICATIONS **141** CITATIONS
SEE PROFILE

Weiyi Shang
Concordia University Montreal
22 PUBLICATIONS **147** CITATIONS
SEE PROFILE

Ahmed E. Hassan
Queen's University
223 PUBLICATIONS **3,233** CITATIONS
SEE PROFILE

Logging Library Migrations: A Case Study for the Apache Software Foundation Projects

Suhas Kabinna[1], Cor-Paul Bezemer[1], Weiyi Shang[2], Ahmed E. Hassan[1]
Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Canada[1]
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada[2],
{kabinna, bezemer, ahmed}@cs.queensu.ca[1], shang@encs.concordia.ca[2]

ABSTRACT

Developers leverage logs for debugging, performance monitoring and load testing. The increased dependence on logs has lead to the development of numerous logging libraries which help developers in logging their code. As new libraries emerge and current ones evolve, projects often migrate from an older library to another one.

In this paper we study logging library migrations within Apache Software Foundation (ASF) projects. From our manual analysis of JIRA issues, we find that 33 out of 223 (i.e., 14%) ASF projects have undergone at least one logging library migration. We find that the five main drivers for logging library migration are: 1) to increase flexibility (i.e., the ability to use different logging libraries within a project) 2) to improve performance, 3) to reduce effort spent on code maintenance, 4) to reduce dependence on other libraries and 5) to obtain specific features from the new logging library. We find that over 70% of the migrated projects encounter on average two post-migration bugs due to the new logging library. Furthermore, our findings suggest that performance (traditionally one of the primary drivers for migrations) is rarely improved after a migration.

1. INTRODUCTION

Logging records useful information during system execution. The information is used for maintaining the system [18, 19, 30], bug-fixing [15, 23, 29], detecting anomalies [23] and transferring knowledge [26]. Every logging statement contains a textual part, which describes the context, a variable part providing more information about the context, and a log verbosity level. Figure 1 shows an example of a logging statement in the code and the corresponding log that is generated at runtime.

Traditionally, logging was done using simple print (e.g. *System.out.println*) statements. Nowadays, a multitude of logging libraries exist that abstract the intricacies of logging and such libraries are used by most projects. The evolution of logging libraries has lead to the emergence of many new features such as uniform verbosity levels [5], reduced performance overhead for logging [7, 9, 11] and management of logs in distributed systems [17]. As a result, developers tend to migrate from older logging libraries to newer logging libraries as, e.g., observed within the *Maven* repository [27].

Logging statement:
LOG.info("Created a new CustomerEntity {} as no matching persisted entity found.", answer); [Apache Camel]

Generated log :
INFO Created a new CustomerEntity Customer[userName: james firstName: null surname: null] as no matching persisted entity found.

Figure 1: Example of a logging statement and output generated at runtime

By understanding how logging library migrations were proposed, discussed and performed in other projects, developers can better plan for logging library migration in their projects. Developers can account for the needed effort and the obtained benefits from migration. Developers can also be aware of the post-migration risks (i.e., bugs) that they might encounter. In this paper, we empirically study the logging library migration in projects from the Apache Software Foundation (ASF). We focus on ASF projects due to the variety of projects, their impact on today's computing practice (many of the projects are extensively used in data centers worldwide), and the availability of rich historical repositories for these projects due to the well defined processes.

Our paper makes the first attempt (to the best of our knowledge) to understand logging library migrations. Below, we highlight our most important findings.

1. **We identify 49 logging library migration attempts in ASF projects, of which 33 were successfully completed.** We find that it takes a median of 26 days to complete a logging library migration. We also find that the *Cassandra* and *Jackrabbit* projects migrate their logging libraries twice.
2. **The 14 cases of abandoned migrations are due to failure to reach a concensus about whether to migrate or not (28%), and failure to provide the necessary code changes (42%).**
3. **Flexibility (in 57% of the projects) and performance improvement (in 37% of the projects) are the primary drivers for logging library migrations.**
4. **In 60%, i.e., 21 of the migrated projects, the necessary code changes are performed by the top three committers of that project.**
5. **Logging library migrations are error-prone and**

over 70%, i.e., 24 of the migrated projects, encounter an average of two post-migration bugs. These bugs are due to unexpected interactions between the old and new logging library, missed dependencies where the new libraries are not included and configuration bugs where the new libraries are misconfigured. These results suggest that logging library migrations are not straightforward and developers have to be more cautious during migrations.

6. **The achieved performance improvement post-migration in Camel and Cassandra is negligible.** The improvement in performance is statistically significant with a large effect size only when the number of generated logs is large. However, when info level logs (i.e., the default level that affects most users) are enabled, we observe that the number of generated logs is relatively small and the effect sizes of the performance improvement are small or negligible. These results suggest that performance improvement should not be one of the primary drivers for logging library migrations in most projects.

The rest of this paper is organized as follows. Section 2 presents background information about logging and relevant prior research. Section 3 presents the methodology for extracting the needed data for our analysis. Section 4 discusses the prevalence of logging library migration within ASF projects. Section 5 shows the necessary effort for logging library migrations and the different types of post-migration bugs that are encountered. Section 6 describes the performance improvement analysis results. Section 7 discusses the threats to validity and Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section, we present an overview of the evolution of logging libraries, the process of migration and the prior research related to this topic.

2.1 Evolution of Logging Libraries

The history of logging libraries can be divided into four eras. Figure 2 gives an overview of the logging libraries that are developed in each era. We focus on logging libraries for C and Java as these libraries have exhibited the most important evolutionary changes, and are the most actively used in data centers today.

Ad Hoc Logging: Before the advent of logging libraries, developers primarily relied on ad hoc methods such as *System.out.println* in Java or *syslog* in Linux for generating outputs and monitoring the projects. However, these ad hoc methods suffered from various issues such as no verbosity settings, difficulties in configuring the output, non-uniform format of output and being difficult to maintain for large deployments.

String concatenation versus parameterized form: Another problem of using ad hoc logging, particularly in Java, is the use of string concatenation. When multiple variables are logged by a *System.out.println* statement in Java, for each logged variable a string concatenation must be performed. Since string concatenation in Java is a slow operation [14], this method of logging is inherently slow. A faster way is using a parameterized form as used by *printf* in C, as this method avoids concatenation but uses string replacement. Both methods are demonstrated below. In ad hoc logging, developers do not get any guidance in writing parameterized logging statements in Java.

String concatenation:
System.out.println("Variables: "+var1+"," +var2);
Parameterized form:
printf ("Variables: %s, %s", var1, var2);

Basic Libraries: Two of the oldest logging libraries are *Log4j* (2001) and Java utility logging *JUL* (2002). These logging libraries introduced verbosity levels such as 'Error', 'Fatal', 'Warn', 'Info', 'Debug' and 'Trace', and were easy to configure. Developers can configure the logging properties for the entire project through one file. Amongst properties that can be configured are the date and time format for the logs and the output location of the logs, (i.e., console, file and remote server) [6]. These logging libraries also provide additional features such as asynchronous logging, a common format for the generated logs and adding appenders, (i.e., package where the logging statement is located) to the generated logs, which assists developers during debugging. In addition, these libraries provide wrapper methods for parameterized logging.

Developers in C followed the example of Java and developed basic libraries such as *Log4c* (2002) and *BoostLogV2* (2003). These libraries provided similar features to Java libraries.

In the wild, a variety of logging libraries were developed [4]. However, as most of these libraries were not licensed under ASF standards, projects within in ASF typically adopted *Log4j* since it was developed under ASF, and *JUL* since it was the default logging library from Java 1.4 onwards.

As the number of logging libraries increased, developers faced difficulties when using different logging libraries in one project. To embed a project using *JUL* into another project using *Log4j*, developers had to change their logging library to *Log4j*, which resulted in changing thousands of lines of code. It also makes integrating newer releases of the project difficult, as newer releases also have to be migrated to *Log4j* before integration.

Log Abstraction Libraries: To overcome the problem of having multiple libraries in a large system, library abstractions were developed. Apache commons logging *JCL* (2003), was the first logging library that supports log abstraction. In *JCL*, developers write their logging statements in the *JCL* format, but have the option of calling any other library such as *Log4j* or *JUL* in the back-end. In essence, log abstraction provides a skeletal structure on which developers could leverage a logging library of their choice.

Slf4j is also a log abstraction library similar to *JCL* with one major difference being that it does not suffer from class loading problems as commonly observed in *JCL* [1]. *Slf4j* also has lower performance overhead than its predecessors because it uses parameterized form of logging statements instead of string concatenated form [11] and helps write cleaner code by avoiding the use of *isLogLevelEnabled* checks.

However, the main drawback of such log abstraction libraries was that they need other libraries for generating logs and it is challenging to configure the interaction between the different libraries that are used within a project.

Log Unification Libraries: To solve the problems faced by log abstraction, log unification libraries were developed and they contained the best of both worlds, i.e., it provides both a basic library and a log abstraction library. *Logback* (2011) was the first unification logging library for Java, as it had a basic library and a log abstraction library, (i.e.,

Figure 2: The eras of logging library development in C and Java

Slf4j) built into it. Similarly, libraries named *Pantheios* and *Google log* (G3log) were developed for C.

Log4j 2 (2014) is the latest unification logging library, which builds on top of *Logback*. *Log4j 2* incorporates all the features of *Logback* and improves the performances of generating logs. However, unlike *Logback*, *Log4j 2* is licensed under ASF. Hence, the studied ASF projects are more likely to integrate it into their code base.

2.2 Library Migration Process

As more advanced logging libraries emerge and evolve, projects migrate to these libraries to reap the benefits of their new features. To understand the process of logging library migration, we focus our study on Java projects as more than 200 [10] out of 322 ASF projects are Java based.

Though the process of logging library migration can vary from one project to another, any logging library migration requires the following three stages:

1. Proposal stage: The proposal stage includes the identification of the drivers for migrating to another logging library. In ASF projects, the discussion is typically conducted on JIRA, where developers explain the problems with the existing library and the benefits of new logging library.

2. Planning stage: The planning stage involves the discussion of which library to migrate to and deciding who will provide the migration patch. The planning stage is broken into two steps namely:

1. Finding the appropriate logging library: After agreeing on migration, developers select the appropriate logging library for their project.
2. Identifying the people to provide migration patch: Upon agreeing which library to migrate to, developers decide on the people who will work on the migration patch.

3. Implementation stage: After selecting the appropriate logging library, one or more developers offer to make the necessary code changes for migration. The implementation stage includes two steps:

1. Adding library dependencies and modifying library property files: Adding dependencies involves adding the necessary library files (e.g., jar files) into a project. Though the process seems straightforward, developers have to select the correct library files when the project has dependency with other libraries. As there can be multiple libraries being used within a project, developers have to be careful when introducing new dependencies by including new libraries.
   Modification to properties files includes setting the default verbosity level, where to log, the output log format and configuring log appenders (i.e., whether output logs to a file or to remove servers). Developers also have to configure the interaction between the different logging libraries (i.e., which basic library interacts with the abstraction library) within the project.
2. Updating the logging method invocations throughout source code files: Developers must update the logging method invocations in the source code to utilize the new logging library.

Automated tools [13] are available for assisting developers with logging library migration. These tools automatically perform parts of the necessary code changes such as updating import statements and log invocations as shown below.

Updating import statements
- import org.apache.commons.logging.Log;
- import org.apache.commons.logging.LogFactory;
+ import org.slf4j.Logger;
+ import org.slf4j.LoggerFactory;
Updating log invocations
public MyClass {
- Log logger = LogFactory.getLog();
+ Logger logger = LoggerFactory.getLogger();

However, these tools have major limitations since they are not capable of doing the following: 1) transforming the logging statements from string concatenated form to parameterized form, 2) adding the necessary dependencies and 3) modifying the build and configuration files.

There exist several open source tools [3, 8, 12] that support updating logging statements from string concatenated to parameterized form. However, these tools are not mature (i.e., with less than five commits) and poorly maintained, making them unsuitable for large projects.

The elaborate stages necessary for logging library migration and the absence of mature automated tools to assist in migration make the migrations of logging libraries a challenging task. The need for these elaborate stages motivates us to understand what drives logging library migrations, the migration process in different projects and the faced post-migration bugs.

2.3 Related Work

In this subsection we present the prior work done on logging library migrations.

Prior research empirically studies the migration of libraries within open source projects. Lämmel et al. [21] uses abstract syntax trees to detect API usage in open source Java projects. They find that *Log4j* and *JCL* are amongst the top 10 most frequently used APIs in Java projects. Cedric et al. [27] study the migration of libraries in projects by mining the *Maven* repository. They find that logging library migrations are the most frequent migrations within *Maven*. A follow-up work done by the same authors [28] shows that *Log4j* is being replaced in favor of different libraries such as *commons-logging*, *Slf4j* and *Logback*. The authors try to investigate the drivers for logging library migration, however due to insufficient data and absence of data from JIRA-like issue tracking systems for Maven projects, the authors only perform a preliminary analysis on the drivers for logging library migrations.

Yana et al. [24] study the trends of library usage within ASF projects and find that developers tend to upgrade the logging libraries to newer version. Kapur et al. [20] present

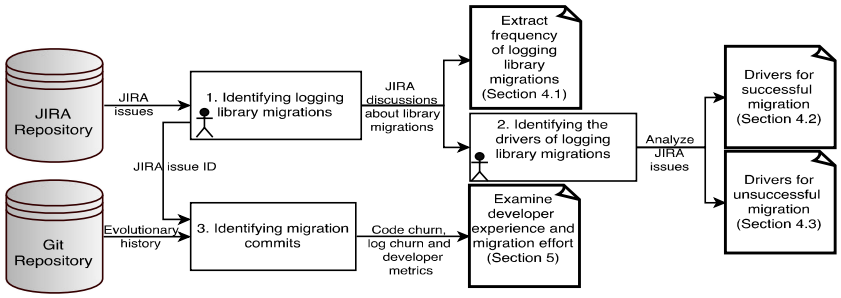

Figure 3: Provides an overview of data extraction and analysis

a tool named *Trident* that supports library migrations by helping developers change the method invocations and arguments. The aforementioned work shows that logging library migrations occur frequently within software projects. However, prior work has never investigated the drivers for logging library migrations in a thorough manner nor did they investigate the effort involved in such migrations. Our work is the first to investigate the drivers, needed effort and challenges of such migrations.

3. METHODOLOGY

In this section, we describe our methodology for identifying the logging library migrations in ASF projects. In addition, we explain the main drivers and the needed effort for such migrations.

3.1 Studied Projects

To identify projects that attempt logging library migrations, we select all Java based ASF projects. We select ASF as it contains more than 200 Java based projects which are actively used in most data centers today. These projects also have issue tracking systems in JIRA, which helps in identifying logging library migrations within these projects.

3.2 Data Extraction Approach

For our study, we need the following data:

1. JIRA issue reports. JIRA issues contain information about the changes made to a project throughout its development and are suitable for identifying projects which migrated their logging libraries.

2. Git commit history. We use the Git commits to analyze the source code changes made during logging library migrations.

The data extraction approach consists of three steps, which are further explained in this section.

1. Identify all issues in JIRA which attempt to migrate logging libraries.
2. Manually analyze the collected JIRA issues to find the drivers for logging library migrations.
3. Collect churn metrics such as code churn and developer metrics such as developer experience to understand the effort spent on logging library migrations.

Figure 3 shows a general overview of the extraction process and we detail below the aforementioned steps.

Identifying Logging Library Migrations: To identify projects that undergo logging library migration, we search the JIRA issues for keywords such as *switch, migrate or change* in conjunction with *Slf4j*, *Log4j*, *JCL*, *logback* or *log*. We collect over 450 issues from our search and manually analyze them to identify all the issues that relate to logging library migrations.

Identifying the Drivers of Logging Library Migrations: To understand the drivers of migrations, we manually analyze all the issues in JIRA which attempt logging library migration. In addition, we examine whether the issue was fixed and closed in order to know whether the logging library migration was completed.

Finally, we identify issues in which logging library migration is considered but not carried out (the issues that are not fixed). We analyze the reasons for these failed attempts at migration, as they can help developers avoid such mistakes.

Identifying Migration Commits: To find the code churn during the migration of a logging library, we extract the JIRA issue IDs of the issues that discuss logging library migrations. We clone the projects corresponding to the issue locally. Using *git log*, we match the JIRA issue IDs to commit messages to identify the commits related to logging library migration.

We use the *git diff* command to extract all the changes made during migration commits. With the extracted commits, we calculate code churn, touched files and changed logging statements to measure the effort spent on logging library migrations. Using the information in the JIRA issues, we calculate the number of involved developers in the discussion posts, the experience of developers and the time taken to resolve the issue.

To measure the experience of the developers involved in the JIRA discussions, we use the Git repository to calculate the number of commits by each developer until the resolution of the JIRA issue and rank the developers based on the number of commits (i.e., the developer with most commits gets the highest rank). Next, we match the unique developer IDs from JIRA issues to the committer IDs in the Git repository. Using such ranks, we can find the experience of the developers that are involved in the JIRA issue and the one providing the migration patch. If there is no match found between the JIRA developer IDs and Git committer IDs, we manually search the project contributor pages to identify if they are contributors to the project.

4. LOGGING LIBRARY MIGRATIONS IN ASF PROJECTS

In this section, we present the results of our manual analysis of issues attempting migration. First, we look at how often logging library migrations occur in ASF projects and the drivers for logging library migrations. We also identify the reasons behind abandoned logging library migrations. By learning from abandoned migrations, developers in other projects can save time and effort .

4.1 Frequency of Logging Library Migrations

Finding 1: We identify 49 attempts of logging library migration in ASF projects, out of which 33

projects (i.e., 14% of 223 main projects) underwent at least one logging library migration. From Table 1 we find that the majority of the projects (26 out of 33) migrate to *Slf4j* followed by migrations to *JCL* in 6 projects (Open Web Beans is the only exception, were developers implement a custom logger API as discussed in the OWB-674 issue). This trend of migration to *Slf4j* and *JCL*, may be because older projects used basic libraries and developers opted for migration to abstraction libraries upon recognizing their new features.

We observe that multiple migrations can occur within a project. In Cassandra and Jackrabbit, developers migrate from *Log4j* to *Slf4j*, followed by the migration to *Logback*. In both projects, developers opt for migration to *Logback* as it incorporates *Slf4j* and provides additional features [9], making it an attractive choice for migration. We also observe that several other long running projects like HBase and Hadoop attempt to migrate their logging libraries twice as seen in the HBASE-10092 issue and the HADOOP-9864 issue respectively, but are abandoned.

Finding 2: Logging library migrations take a median of 26 days to complete (time taken between opening of a JIRA issue until it is closed), involving an average of three developers in the JIRA discussion posts. We find that there are about nine posts on average for completed logging library migrations. The longest discussion post is in the Zookeeper project (ZOOKEEPER-850[1]), with over 63 discussion posts and involving 15 developers as there is a confusion over completely removing all dependencies of the older *Log4j* library as several tests rely on *Log4j* to verify if the tests passed or failed.

We observe that in abandoned logging library migrations, developers discuss the merits and drawbacks of migration, as there is no common consensus. We find that abandoned migrations have more developers involved (1.6 times higher than completed migrations) and longer discussion posts (1.5 times longer than completed migrations). Finding 2 suggests that logging library migrations require team effort and developers actively participate during discussions because developers may not fully understand the positive and negative effects of migration.

Finding 3: In 22 of the migrated projects (66%), at least one of the top three committers participates in the discussions. A top committer is always involved in projects that abandon migration. Our finding suggests that logging library changes impact all developers and require experienced developers for the migration.

4.2 Library Migration Drivers

To understand what drives logging library migrations, we manually examine all the 49 JIRA issues that attempt a migration. We find the common drivers between different projects and find that projects can have multiple drivers for migration. The main drivers are listed below.

***1. Flexibility*:** For a project to be easy to integrate, it should not force developers to use its logging library. Newer libraries provide a library abstraction feature, which provides a skeletal structure, allowing developers to easily integrate other projects using different logging libraries without code changes. The other added benefit is that library abstractions helps end-users unify the configuration of the logging of all projects.

2. Performance improvement: Logs are generated faster with minimal overhead. For example, *Slf4j* supports the parameterized from of logging which avoids the overhead from string concatenations.

***3. Code maintenance*:** As a project matures, it may rely on several components from other projects for additional features. However, these different logging libraries can have different verbosity levels and logging formats. To reduce the cost incurred due to code maintenance, developers opt for migrating to newer libraries that have a unified logging format which is easy to configure and code, and produces cleaner looking code.

4. Functionality: Developers migrate to newer libraries to obtain very specific functionalities which are beneficial for their projects. Some of these features are mapped diagnostic support, (i.e., encoding a log with a unique identifier to help debugging in distributed systems) in *Slf4j*, auto reloading of configurations in *Logback*.

5. Dependency: Developers are forced to migrate logging library when a newer version of a package on which they depend on moves to a different library.

6. Undefined: Developers do not provide any information on JIRA about the drivers for migration.

Table 2 shows the different drivers for logging library migration and the percentage of projects that mention these drivers in the JIRA discussion.

Finding 4: Flexibility offered by a new logging library is the most important driver for logging library migrations. From Table 2, we find that developers reference flexibility in more than 57% of the projects attempting migration. For example, in the HADOOP-211 and ZOOKEEPER-850 issues, developers migrate to *JCL* and *Slf4j* respectively as these libraries provide logging abstraction. Developers quote that this migration is beneficial as they can switch logging implementations without changing thousands of lines of logging code.

From an end-user perspective, in the PLUTO-553 issue, a developer mentions that '*a) only using Java logging (*JUL*) or b) using Apache* Log4J*, directly forces end users (integrators) of Pluto to leverage the same logging solution which imo is too restricted*'. Hence, developers opt for migration to *Slf4j* as its more flexible for end-users.

From Table 1, we observe that the migrated projects are mainly service providers (i.e., provide a service upon integration in other projects) and development frameworks (i.e., help in developing applications). In the case of service providers, migration to log abstraction libraries is beneficial because it is necessary for the project be to integrable in large systems. In the case of development frameworks, the projects themselves may need to incorporate different features from other projects, forcing migration to log abstraction libraries.

Flexbility Achieved Post-Migration. To measure the flexibility achieved post-migration, we count the logging libraries used within a project. As a build file contains all the dependencies of a project, we inspect the build files to identify the number of used logging libraries in a project.

We find that pre-migration only 9 projects use a single logging library. However, post migration we find that all projects use more than one logging library after migration and in 22 projects there is an increase (median increase of

[1]In the rest of the paper we do not show weblinks for JIRA issues, since the link always follow the same pattern as - https://issues.apache.org/jira/browse/ZOOKEEPER-850

Table 1: Statistics of the studied projects

Projects	Project type	Project size	# of Contributors	Migrating from	Migrating to	Migrating version	# of discussion posts	# of developers providing migration patch	Total code churn	% of log churn in migration commit	% of affected project files
Airavata	Service provider	823 K	24	Log4j	Slf4j	0.1	2	1	5,109	494 (9.66%)	212 (20.36%)
AMQ	Service provider	293 K	18	JCL	Slf4j	5.5.0	1	1	2,366	64 (2.70%)	572 (17.39%)
Bookkeeper	Service provider	226 K	5	Log4j	Slf4j	4.0.0	7	1	495	99 (20%)	98 (31.81%)
Bval	Library	17 K	6	JCL	Slf4j	0.3	8	1	180	25 (13.88%)	17 (4.37%)
Camel	Service provider	1.08 M	134	Log4j	Slf4j	2.7.0	12	2	14,487	1,559 (10.76%)	1,329 (18.55%)
Cassandra (Slf4j)	Service provider	340 K	128	Log4j	Slf4j	0.7	9	1	2,015	10 (0.49%)	116 (8.56 %)
Cassandra (Logback)	Service provider	340 K	128	Slf4j	Logback	2.1.Beta	10	2	661	9 (1.36%)	21 (1.55 %)
Configuration	Library	662 K	18	JCL	JUL	2.0.0	19	1	281	17 (6.04%)	8 (2.48%)
DOSGi	Service provider	16 K	4	JUL and Log4j	Slf4j	1.4.0	1	1	934	188 (20.12%)	76 (23.6%)
Empiredb	Database service	43 K	5	Log4j	Slf4j	2.1.0	4	1	974	43 (4.41%)	97 (23.37%)
Flume	Service provider	87 K	11	Log4j	Slf4j	0.9.2	13	1	508	150 (29.52%)	165 (25.82%)
Hadoop	Service provider	1.9 M	69	Log4j	JCL	0.3.0	33	1	610	130 (21.31%)	55 (1.07%)
HDFS	Database Service	158 K	5	Log4j	JCL	2.0.0-Alpha	6	4	1,983	53 (2.67%)	45 (0.87%)
Hcatalog	Database Management	112 K	2	Log4j	Slf4j	0.4.1	40	1	606	76 (12.54%)	30 (9.58%)
Isis	Web development framework	407 K	20	Log4j	Slf4j	1.3.0	2	1	2,646	135 (5.10%)	304 (4.16%)
Jackrabbit (Slf4j)	Web content framework	357 K	17	Log4j	Slf4j	1.0	8	1	179	14 (7.82%)	106 (6.12%)
Jackrabbit (JCL)	Web content framework	357 K	17	Slf4j	JCL	2.2.0	2	2	1,188	12 (1.01%)	46 (2.65%)
JSPWiki	Wiki engine	109 K	4	Log4j	Slf4j	3.0.0	41	1	980	139 (14.18%)	182 (16.6%)
Karaf	Service provider	228 K	44	Log4j	Slf4j	2.2.0	3	1	399	12 (3.00%)	41 (5.27%)
Mahout	Machine learning application	126 K	17	Log4j and JCL	Slf4j	0.1.0	6	1	13	0 (0%)	13 (2.41%)
Nutch	Web crawler	117 K	10	Log4j	JCL	0.8.0	3	1	1,426	271 (19%)	105 (8.94%)
ODE	Process management	151 K	9	JCL	Slf4j	1.3.7	5	1	570	31 (5.43%)	40 (1.36%)
OpenEJB	Service provider	534 K	7	Log4j	Slf4j	4.0	3	1	108	15 (13.88%)	108 (1.78%)
OWB	Library	94 K	7	JUL	Custom	1.1.5	2	1	1,112	371 (33.36%)	74 (5.15%)
PDFBox	PDF creator	132 K	6	JUL	JCL	0.8.0	9	1	210	3 (1.42%)	29 (4.54%)
Pluto	Service provider	138 K	7	JCL	Slf4j	2.0.0	6	1	401	11 (2.74%)	74 (11.38%)
Santuario	XML security provider	217 K	4	JCL	Slf4j	2.0.0	1	1	665	29 (4.36%)	126 (6.36%)
Shiro	Security framework	60 K	2	JCL	Slf4j	1.0.0	3	1	553	5 (0.90%)	83 (16.11%)
Struts-1	Web framework	60 K	2	Log4j	JCL	1.3	1	1	503	46 (0.91%)	12 (8.11%)
Sling	Web framework	453 K	14	JCL	Logback	4.0.0	8	3	7,867	141 (1.79%)	94 (3.10%)
ServiceMix	Integration framework	708 K	18	JCL	Slf4j	3.4.0	3	3	6,879	37 (0.53%)	53 (11.57%)
Thrift	Cross language development framework	226 K	71	JUL	Slf4j	0.1.0	14	1	72	29 (40.27%)	7 (1.17%)
Tiles	Web framework	32 K	3	JCL	Slf4j	2.2.9	1	1	336	0 (0%)	48 (4.51%)
Zookeeper	Configuration framework	143 K	9	Log4j	Slf4j	3.4.0	63	1	587	86 (14.65%)	176 (20.92%)
Median							**9**		**610**	**37(5.43 %)**	**76 (6.12%)**

50%) in the number of used logging libraries after migration.

Finding 5: Performance improvement is one of the primary drivers for logging library migration in 37% of the projects attempting migration. For example, in the HADOOP-6884 issue, developers run a performance test, where they run a unit test over 10,000 times to measure the performance gained by avoiding *isLogLevelEnabled* checks and string concatenations. Similar discussions are seen in the ODE-983, and HCATALOG-68 issues.

Finding 6: Code maintenance forces developers to migrate to a single library in 33% of the projects attempting migration. From Table 2, we observe that 33% of the migrations are driven by the desire to reduce the effort spent on code maintenance within the project. By using library abstractions such as *Slf4j* or *JCL*, developers can avoid the complexity of maintaining multiple configuration and build files for each logging library. For example, in the KARAF-427, DOSGI-135 issues, developers use multiple logging formats and decide to migrate to *Slf4j* to provide homogeneity in the project. In the PDFBOX-472 issue, developers migrate to *Slf4j* as its easier to code and prevents the creation of large log dumps. In the HTTPCLINET-416 issue, developers migrate to *Slf4j* as it helps avoid *isLogLevelEnabled* checks which makes the code clean and easier to follow.

4.3 Reasons for Abandoned Migrations

We find 14 abandoned attempts for logging library migration and the main reasons for these failed attempts are listed below.

1. Consensus is not reached: In four projects, developers do not reach a common consensus about logging library migration. For example, in the project PDFBOX, developers migrate to *JCL* initially. However, when migration to *Slf4j* is suggested in the PDFBOX-693 issue, developers do not like the idea of adding additional dependencies to the project. Moreover, a senior developer quotes '*Yes, there are things that other logging packages do that JUL does not. I have yet to be involved in any project where those additional 'features' have had any impact at all.*'. Due to these reasons migration is not carried out.

Table 2: Drivers for migrating logging libraries in ASF.

Drivers behind migration	(%)
Flexibility	57.4
Performance improvement	37.0
Code maintenance	33.3
Dependency	7.4
Functionality	11.1
Undefined	11.1

In the HTTPCLIENT-416 issue, developers argue about the performance benefits that are achieved by migration to *Slf4j*. To settle the argument the developers vote on the migration to *SLf4j*, however a majority is not achieved.

2. Failure to provide a patch: In six projects, no developer takes responsibility for the migration or provides the initial patch. For example, in the HBASE-2608 issue, developers agree that migration to *Slf4j* is beneficial. However, nobody submits a patch with the initial migration work and the issue is closed as "Won't Fix". A similar case occurs in the KAFKA-105 issue, where developers suggest migration to *Slf4j*. However, nobody provides a patch and the issue is closed years later.

3. Delayed for upcoming library: We find that in two

projects (HBase and Pivot) the migration is delayed until a better library is released. In the PIVOT-882 and HBASE-2608 issues, developers opt for migration to *JCL* and *Slf4j* respectively. However, in Pivot developers decide to wait and migrate to *Log4J 2*. In HBase, developers decide on migration to *Slf4j* however, no developer submits the patch for migration and the issue is closed as developers decided to migrate to *Log4J 2* in HBASE-10092 issue.

4. Dependency issues: We find that in two projects (HBase and Droids), dependency on other projects prevents migrating to new logging library. For example in the HBase project, developers consider migration to *Slf4j* in the HBASE-11334 issue to standardize logging. However, as HBase is closely related to Hadoop[2] which itself uses many logging libraries, migration to *Slf4j* is not pursued.

These findings show that 1) library migrations are not straightforward, 2) migrations can be beneficial to both end-users and developers and 3) the process of migration is not trivial and requires experienced developers

5. EFFORT SPENT ON MIGRATIONS

In the previous section, we find that 33 projects underwent at least one logging library migration. However, we find that 6 out of the 14 abandoned attempts for logging library migrations fail because no-one provides the migration patch. The failed migrations suggest the non-trivial effort that is required for logging library migrations. Hence, to further understand the needed effort for logging library migrations, we calculate the code and log churn during the migration process and the bugs faced post-migration.

To calculate the effort spent by developers (i.e., code and log changes made) we extract the commits that are related to logging library migration as explained in Section 3. As developers may have used library migration tools, we exclude the library import and log invocation changes from the total churn. The remaining changes in a commit are made by the developers during logging library migration.

Finding 7: Logging library migrations have significant log and code churn in the studied projects. From Table 1 we observe that in 33 migrated projects, close to 10% of all the project files are changed during the migration. We find that the majority of these files are Java and XML files. The XML files are changed to include the dependencies on the new logging library, and Java files are modified to remove custom log levels, such as 'finer', 'finest' and 'warning'.

Finding 8: In 84% (i.e., 27) of migrated projects, only one developer contributes in the logging library migration patch. In the remaining projects, a patch for library migration is contributed by multiple developers.

Over 60% of logging library migration commits are provided by at least one of the top three committers. This result suggests that logging library migrations may need the knowledge of experienced developers who have a thorough understanding of the project.

Finding 9: Developers do not have any guidelines to assist them during logging library migrations. We observe that 35% of the migrated projects are split into multiple sub-issues (i.e., several JIRA issues); while in the remaining projects, we cannot find evidence of splitting the migration to sub-tasks. Finding 9 suggests that there exists no guidelines to help developers perform the migration. This discrepancy is also apparent from Table 1, where we find that some projects have higher log churn and files affected as they change the logging code from string concatenated to parameterized form. However, majority of the studied projects do not change their logging code as it involves significant code changes and the performance benefits from parameterized from is not obtained.

Findings 7 and 8 show that logging library migrations require significant log and code churn and Finding 9 suggests that there exists no proper guideline which developers can follow to assist in the migration. The absence of guidelines can lead to post-migration bugs which might cripple the system. To identify such bugs, we first collect new issues which have keywords such as 'log', 'logging library', 'Slf4j', 'Log4j', 'JCL' and 'logging migration' from the migrated projects, within 6 months after the logging library migration. Next, we manually look at all the collected issues to identify the issues caused by logging library migrations. We use 6 months because most ASF projects have a 4 to 6 month software release cycle and users face post-migration bugs after the release. In the remainder of this section, we discuss the types of bugs that occur after logging library migration. We also study the effort spent on resolving these bugs.

Finding 10: 24 projects out of the 33 migrated projects face an average of two post-migration bugs. We find that the median resolution time for these bugs is three days and the bugs involve an average of three developers. By analyzing the post-migration bugs and the Git commits that fix the bugs, we categorize the 48 post-migration bugs that are identified into three categories namely 1) Unexpected interactions, 2) Forgotten dependencies and 3) Configuration bugs.

1. Unexpected interactions: 13 (27%) post-migration bugs arise because of unexpected interactions between logging libraries in the studied projects. Such interactions arise when the older logging library and the new added library override one another. For example, in the CAMEL-4568 issue, developers intend to use *Slf4j* library for generating the logs and a JDBC library for storing them. However, due to unexpected interaction between JDBC and *Slf4j* libraries, JDBC library is used for both generating and storing logs which is not desired.

2. Forgotten dependencies: 21 (45%) post-migration bugs arise because of missing to add dependencies during migration. Such bugs arise when developers fail to add the migrated libraries or remove the older libraries from build files. For example, in the post-migration bug ZOOKEEPER-1371, developers remove dependencies on *Log4j* in the source code after migration. In the BOOKKEEPER-128 issue, developers forget to add the new logging libraries as dependency which causes release bugs.

3. Configuration bugs: 14 (28%) post-migration bugs occur due to misconfiguration during migration. Misconfiguration can be due to misconfigured path to output logs as in HADOOP-272 or poorly configured properties as in NUTCH-318.

6. BENEFITS OF MIGRATION

We find that the performance improvement is one of the two primary drivers behind logging library migrations. To understand the benefits obtained post-migration, in this section we measure the performance improvements observed.

6.1 Performance Improvement Post-Migration

[2]https://hadoop.apache.org/

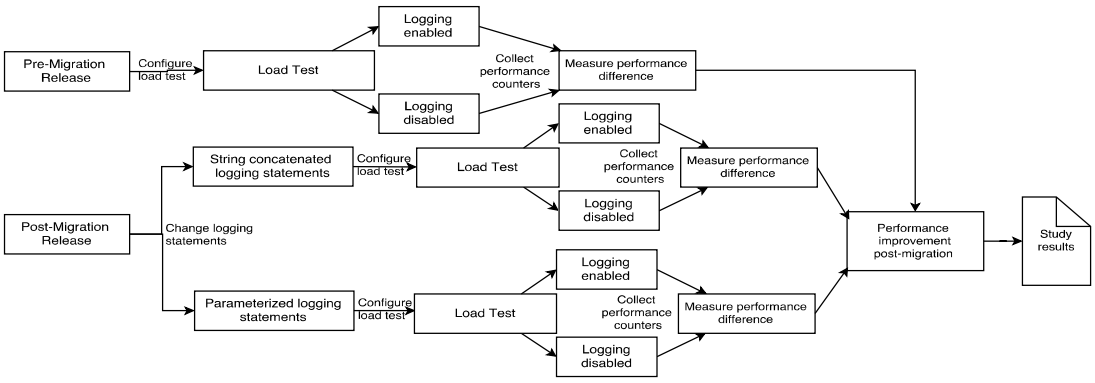

Figure 4: Overview of load test setup, in the studied projects

Table 3: Overview of projects used in performance study

Projects		Version	Release date	Logging library
Camel	Pre-migration	2.6.0	2011-01-30	Log4j
	Post-migration	2.7.0	2011-03-17	Slf4j
Cassandra	Pre-migration	2.0.9	2014-06-30	Log4j
	Post-migration	2.1.0	2014-09-16	Slf4j
EmpireDB	Pre-migration	2.0.7	2010-12-12	Log4j
	Post-migration	2.1.0	2011-03-06	Slf4j

In Section 4.2 we find that in 37% of the migrated projects performance improvements is one of the drivers for logging library migrations. However, there exists no study that quantifies the gain in performance post-migration. To measure whether using performance improvement as motivation is justified, in this section we analyze the performance that is gained by three projects which migrate logging libraries. Figure 4 shows an overview of our approach.

6.1.1 Studied Projects

We use the following criteria to pick the projects for our performance study:

1. **Presence of load tests or examples:** Projects must have load tests or examples, in order to run the project and to emulate real world scenarios.
2. **Project activity:** Projects must be actively maintained (i.e., more than three years of commit history and still active), to make sure that the project is currently used and has large development history.
3. **Slf4j or Logback migration:** Projects must migrate to the *Slf4j* or *Logback* library, as *Slf4j* is associated with performance benefits and *Logback* uses the *Slf4j* library.

We find three projects, which fit the above mentioned criteria. Camel is an open source integration platform, Cassandra is an open source database project and EmpireDB is a relational database abstraction layer with data persistence component. Table 3 shows an overview of the releases of the project pre and post-migration.

6.1.2 Test Setup

To measure the performance improvement after logging library migration, we run the performance tests twice, i.e., pre and post logging library migration.

As improvements can be made to the projects during logging library migration, we establish a baseline by running the project with logging disabled. We run the load test with 'Error' level enabled to establish the initial baseline.

To measure the performance gain, we run the same load test twice with 'Debug' and 'Info' level logs enabled. Debug level is used by developers for debugging purposes during development and info level is the default level when the project is released for end-users. By measuring the time taken for generating logs at both debug and info levels we can understand the performance improvements achieved for developers and end-users.

We measure the time taken from the time a load test starts until the time all logs are generated for the load test. By subtracting the execution time of the debug level test from the baseline test (i.e., error level test), we obtain the time taken for generating the logs in the debug enabled tests.

As the time taken for generating logs can be affected by the number of logs generated and the number of variables printed in logs, we calculate these two factors pre and post-migration. To calculate the number of logs generated during a load test we dump the load test logs into a file and count the number of log lines within the file. To find the average number of variables printed in the logs, we analyze the source code of the load tests to find all the logging statements and the number of variables in each logging statement. Next, we run the load test once and match the logging statements from source code to generated logs and identify which logging statements are run multiple times. We tally all the logging statements executed during our load tests and calculate the average number of variables generated during a load test.

To control for the number of logs generated and average number of variables printed pre and post-migration, we divide the time taken for generating logs, by the number of logs generated during a load test (the number of variables remain the same for our load tests). This ratio is the time taken for generating one log, pre and post-migration. We multiply this taken time against the total number of logs generated during the pre-migration release to obtain controlled values for our load tests. Table 4 shows the number of generated logs and the average number of variables printed during our load tests.

In EmpireDB and Cassandra projects, we find that developers do not change the logging statements from string concatenated form to the parameterized form. It is already established that string concatenations have a high performance overhead [2]. To reduce this overhead we use an open source script [12] to change the format of all logging statements to a parameterized form to optimize the performance for post-migration. As these migration tools are not tested,

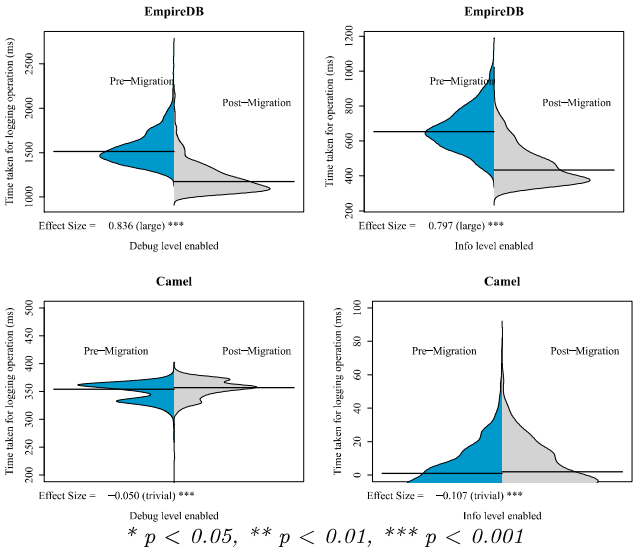

Figure 5: Comparing the difference in time to execute the load tests pre and post-migration, with their respective effect sizes

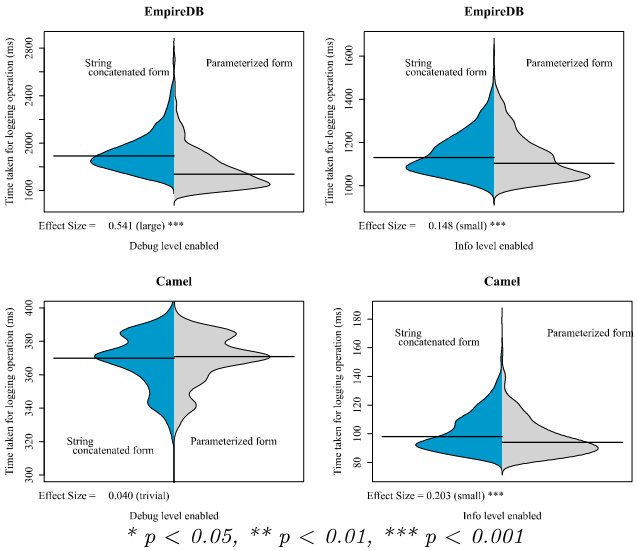

Figure 6: Comparing the difference in time to execute the load tests for string concatenated v.s parameterized logging statements, with their respective effect sizes

Table 4: Log output generated during load tests

Projects	Log level		Pre-migration	Post-migration
EmpireDB	Debug	# of generated log lines	42,325	50,397
		Avg. # of printed variables	3	3
	Info	# of generated log lines	14,148	1,168
		Avg. # of printed variables	3	3
Camel	Debug	# of generated log lines	5,095	5,076
		Avg. # of printed variables	1	1
	Info	# of generated log lines	245	256
		Avg. # of printed variables	1	1
Cassandra	Debug	# of generated log lines	200,783	450,265
		Avg. # of printed variables	2	2
	Info	# of generated log lines	350	371
		Avg. # of printed variables	1	1

we manually debug all the warnings and errors which occur after converting the logging statements from string concatenated to parameterized form in the two projects.

6.1.3 Used Load Tests

EmpireDB Advanced Example: To load test EmpireDB, we use the provided *advanced example*, which showcases many database operations such as database creation, insertion and deletion of records, modification of records, bulk read and processing of records. We run a load test with the *advanced example* application by increasing the number of records to 1,000 and running the application for 1,000 iterations, measuring the time taken to complete the load test and generate the logs at each iteration.

Camel Loan Broker Application: We load test the *Camel loan broker* application [**?**] by increasing the number of servers and run the application for 1,000 iterations, recording the time taken for generating logs at each iteration.

Cassandra Stress Test Application: We use the stress test application available in Cassandra to run the load test. This stress test can insert, read and index a large number of records within the Cassandra cluster.

We load test the application by inserting 100,000 records into the Cassandra cluster at each iteration and run 1,000 iterations, recording the time taken for generating logs during each run.

6.1.4 Test Evaluation

To find if there is a statistically significant difference between the time taken to complete a load test pre and post-migration, we use the *MannWhitney U test* (Wilcoxon rank-sum test) [16]. The *MannWhitney U test* is a non-parametric test, hence it does not have any assumptions about the distribution of the sample population. A p-value of ≤ 0.05 means that the difference of the time taken to generate logs between the pre and post-migration releases is statistically significant and we may reject the null hypothesis. By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us that there is a statistically significant difference of the time taken to generate logs between pre and post-migration releases.

We also calculate the *effect sizes* in order to quantify the differences in performance, pre and post-migration. Unlike the *MannWhitney U test*, which only tells us whether the difference between the two distributions is statistically significant, the effect size quantifies the difference between the two distributions. Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value are likely to be small even if the difference is trivial). We use *Cliff's Delta* to quantify the effect size [22] and use the following thresholds for *Cliff's Delta* [25]:

$$\begin{cases} \text{trivial} & \text{for } |d| \leq 0.147 \\ \text{small} & \text{for } 0.147 < |d| \leq 0.33 \\ \text{medium} & \text{for } 0.33 < |d| \leq 0.474 \\ \text{large} & \text{for } 0.474 < |d| \leq 1 \end{cases} \quad (1)$$

6.1.5 Results

Finding 12: We find a 28-44% improvement in performance in two of the studied projects post-migration. Figure 5 and Table 5 shows that there is an

Table 5: Effect sizes of the time taken to complete a load test during info and debug enabled in the studied projects. Effect sizes are bold if P-values are smaller than 0.05

Projects		Debug	Info
EmpireDB	Pre vs post-migration	**0.83 (large)**	**0.79 (large)**
	String Concatenated vs Parameterized from	**0.54 (large)**	**0.14 (small)**
Camel	Pre vs post-migration	**-0.05 (trivial)**	**-0.10 (trivial)**
	String Concatenated vs Parameterized from	0.04 (trivial)	**0.20 (small)**
Casandra	Pre vs post-migration	**0.91 (large)**	**0.32 (small)**
	String Concatenated vs Parameterized from	**0.32 (small)**	**0.89 (large)**

improvement in performance (statistically significant with large effect size) post-migration in EmpireDB and Cassandra when debug level logs are enabled (due to constrained space only the plots of EmpireDB and Camel are displayed).

From Table 4 we observe that performance improvements are observed in EmpireDB and Cassandra which produce a larger number of log lines in comparison to Camel. We also observe that the average number of variables in EmpireDB and Cassandra is higher than Camel which suggests that performance improvements depend on the number of logs and the number of variables in the output.

We find that parameterized logging statements are faster than concatenated logging statements for both debug and info level enabled. Table 5 shows that there is an improvement in performance post-migration in all the projects. However, we find that improvement is statistically significant only in Cassandra and EmpireDB with small to medium effect sizes. This may be because parameterized form of logging statements only improves the performance of those logging statements which have more than one output variable as see in Table 4. This result suggests that changing from string concatenated to parameterized form can increase performance only in specific cases.

Finding 13: We find that performance improvement is negligible in two of the studied projects, when info level is enabled. Table 5 shows that the improvement in performance is statistically significant with large effect size only in EmpireDB. In Camel and Cassandra, we find that the difference in performance is small, implying that the improvement is too small to be noticeable for most users.

From Findings 12 and 13, we conclude that improvement in performance should not be the main criteria for considering logging library migration, as the results are negligible when the default logging level (i.e., info) is enabled.

7. THREATS TO VALIDITY

External Validity. Like all empirical studies, our analysis is subject to the representativeness of the studied projects. To address this threat, we looked into all the projects within the ASF foundation. Our projects are all open source and from different domains. More studies on other open source foundations, with other programming languages are needed to see whether our finding can be generalized.

We study logging library migrations for only Java projects, as many logging libraries exist for Java making migrations more likely to occur. More studies on other languages are needed to see whether our findings can be generalized.

Construct Validity. Our heuristics to extract logging library migrations may not be able to extract every migration within ASF. A manual evaluation on extracted migrations and expanding our keyword based heuristics is necessary to address this threat.

The JIRA issue IDs may not match all the commits related to logging library migrations. A manual evaluation of the commits is necessary in future work to identify the recall of our approach.

The tags for the main drivers of migration are based on our manual analysis of the JIRA issues and therefore can be biased. However, each issue was verified by two authors and common consensus was reached before tagging each issue if there is disagreement between the authors.

Internal Validity. Our analysis is based on data from Git and JIRA repositories. The quality of data contained in the repositories can impact the internal validity of our study.

Our analysis of performance, measures the difference in time when the system is load tested against debug and info level. However, the I/O speed in our environment can be very different from the I/O speed in the field, where several different process may be competing for resources. In such scenarios the performance improvements observed may be even lesser than observed in our environment.

We measure the performance of projects pre and post-migration. However, in the long run, developers may add more logging statements into the code and the migration to newer logging library might be useful.

8. CONCLUSION

Logging libraries assist developers in logging their code by providing a clean API to configure where to log, when to log and in what format to log.

Due to the plethora of logging libraries developed in recent years, developers face the challenge of migrating from one library to another to leverage new features. To understand such logging library migrations, in this paper, we study the migrations within Apache Software Foundation (ASF). The goal of our work is to first identify how frequent are logging library migrations, the effort necessary for migrations and the drivers for migrations. We also aim to understand why migration attempts are abandoned and to identify the common post-migration bugs and the benefits gained after migration. With such knowledge, developers would have a more realistic view of the post-migration benefits, in order to better plan for migrations. The highlights of our work are:

1. We identify 33 projects within ASF which migrate logging libraries, which take a median of 26 days to perform and involve at least one top developer in the discussions.
2. Flexibility and performance improvement are the primary drivers for logging library migrations referenced in 57% and 37% of the migrated issues respectively.
3. Migrations are non-trivial: 28% of attempted migrations in ASF are abandoned and 70% of migrated projects face an average of two post migration bugs.
4. Performance benefits from logging library migration are not readily visible for most users.

These highlights suggest that logging library migration is not a trivial task. Developers should better estimate the effort needed and the performance improvements achieved from migration. They should also evaluate the risks of migration and plan for mitigating post-migration bugs.

References

[1] Class loader - http://articles.qos.ch/classloader.html.

[2] Concatenated vs parameterized form. `http://www.javacodegeeks.com/2013/03/java-stringbuilder-myth-debunked.html`.

[3] https://github.com/rosarinjroy/log4j-to-slf4j.

[4] Java logging libraries :. `http://java-source.net/open-source/logging`.

[5] Log level :. `https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html`.

[6] Log4j manual :. `https://logging.apache.org/log4j/1.2/manual.html`.

[7] Log4j2- https://logging.apache.org/log4j/2.x /performance.html.

[8] Log4j2–migrator :. `https://github.com/mulesoft-labs/log4j2-migrator`.

[9] Logback – http://logback.qos.ch/reasonstoswitch.html.

[10] Projects in asf :. `https://projects.apache.org/projects.html?language`.

[11] Slf4j – http://www.slf4j.org/faq.html.

[12] Slf4j–migrator. `https://github.com/ghosert/SLF4J-Migrator`.

[13] Slf4j tool - http://www.slf4j.org/migrator.html.

[14] String concatenation performance :. `{http://blog.eyallupu.com/2010/09/under-hood-of-java-strings.html}`.

[15] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM'09: Proceedings of the 9th IEEE International Conference on Data Mining*, pages 149–158. IEEE, 2009.

[16] E. A. Gehan. A generalized wilcoxon test for comparing arbitrarily singly-censored samples. *Biometrika*, 52(1-2):203–223, 1965.

[17] M. . http://logback.qos.ch/manual/mdc.html.

[18] Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *ICSM '08: Proceedings of the IEEE International Conference on Software Maintenance*, pages 307–316, IEEE, 2008.

[19] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *ICSM '09: Proceedings of the IEEE International Conference on Software Maintenance*, pages 125–134. IEEE, 2009.

[20] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *ACM SIGPLAN Notices*, volume 45, pages 726–738. ACM, 2010.

[21] R. Lämmel, E. Pek, and J. Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *SAC'11: Proceedings of the ACM Symposium on Applied Computing*, pages 1317–1324. ACM, 2011.

[22] J. D. Long, D. Feng, and N. Cliff. Ordinal analysis of behavioral data. *Handbook of psychology*, 2003.

[23] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX'10: Proceedings of 10th Conference on USENIX Annual Technical Conference*, pages 24–24. USENIX Association, 2010.

[24] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 57–62. ACM, 2009.

[25] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohens'd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*, 2006.

[26] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[27] C. Teyton, J.-R. Falleri, and X. Blanc. Mining library migration graphs. In *WCRE '12: Proceedings of the 19th Conference on Reverse Engineering*, pages 289–298. IEEE, 2012.

[28] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, 2014.

[29] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOPS 2009: Proceedings of the 22nd Symposium on Operating Systems Principle*, pages 117–132.

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