Reducing Instrumentation Overhead when Reverse-Engineering Object Interactions

Hossein Mehrfard
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
mehrfard@sce.carleton.ca

Yvan Labiche
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
labiche@sce.carleton.ca

***Abstract*— Reverse-engineering object interactions from source code can be done through static, dynamic, or hybrid (static plus dynamic) analyses. In the latter two, monitoring a program and collecting runtime information translates into some overhead during program execution. Depending on the type of application, the imposed overhead can reduce the precision and accuracy of the reverse-engineered object interactions (the larger the overhead the less precise or accurate the reverse-engineered interactions), to such an extent that the reverse-engineered interactions may not be correct, especially when reverse-engineering a multi-threaded software system. One is therefore seeking an instrumentation strategy as less intrusive as possible. In our past work, we showed that a hybrid approach is one step towards such a solution, compared to a purely dynamic approach, and that there is room for improvements. In this paper, we uncover, in a systematic way, other aspects of the dynamic analysis that can be improved to further reduce runtime overhead, and study alternative solutions. Our experiments show effective overhead reduction thanks to a modified procedure to collect runtime information.**

***Index Terms*— Reverse engineering; Overhead; Multi-threaded; Hybrid analysis; Object interactions; Logging.**

I. INTRODUCTION

Object interactions, for instance rendered as UML interaction diagrams, can be recovered from source code through static, dynamic, or hybrid analyses. Such interactions can then be used for program comprehension and verification for example. A static analysis usually produces accurate diagrams through source code analysis regardless of all possible program inputs and behaviour [1]. However, a static analysis recovers program behaviour conservatively. Language features such as late binding, generalization, overloading, and aliasing hinder reverse engineering object interactions solely from source code, and sometimes make that even impossible; it is in general NP-hard [2]. Therefore, engineers turn to dynamic analysis to recover such interactions. However, the instrumentation that captures dynamic information adds runtime overhead to the execution of the system under study (SUS), increasing the SUS's response time to the extent that a deadline may be missed, resulting in an observed behaviour that may be different from the expected one. This is sometimes referred to as the "probe effect". Reducing the probe effect is important when reverse-engineering any behaviour since overhead can drastically increase execution times to the point that it is not practical for engineers to wait. Reducing the probe effect is especially important when reverse-engineering interactions in a multi-threaded system: as mentioned earlier, execution overhead may lead to deadline missed and therefore a different observed behaviour than the expected one. Therefore, one may turn to hybrid (static plus dynamic) approaches to benefit from the advantages of each kind of technique and minimize their drawbacks. A hybrid analysis has the potential to produce more accurate and less conservative behaviour by 1) capturing as much information as possible in the static analysis and 2) capturing, at runtime, the remaining amount of information while reducing the probe effect.

We have shown in our past work that combining static and dynamic analyses reduces runtime overhead compared to a purely dynamic approach [3]. In this paper, we optimize the design and implementation of the AspectJ instrumentation used in our hybrid approach [3] to further reduce the overhead. To do so, we systematically identify characteristics of our previous hybrid instrumentation (thereafter referred to as *Light*) that may lead to overhead and study to what extent they actually contribute to the overhead. Then, we systematically discuss a set of optimizations for each characteristic and study the extent of overhead reduction. More specifically, we answer the following research questions (RQ): RQ1: To what extent does each characteristic of the Light instrumentation contribute to overhead? RQ2: What are proper optimizations for the Light instrumentation and how much do they reduce overhead? RQ3: How effective is combining optimization strategies studied in RQ2 at reducing the probe effect?

Our contribution, which should be of interest to anyone who intends to develop a hybrid reverse-engineering technique, is five-fold: (i) A classification of the characteristics of our Light instrumentation that may produce overhead. This is a somewhat general result that would apply to other hybrid instrumentation techniques, in particular, those based on AspectJ instrumentation; (ii) A protocol to systematically study the contribution to overhead of each characteristic. Again, this should apply to other hybrid instrumentation techniques; (iii) A quantitative analysis of the contribution to overhead of each characteristic of our hybrid technique; (iv) A discussion about ways to optimize our hybrid technique to further reduce overhead; (v) A quantitative analysis of the result of implementing some of these optimizations, thereby reducing overhead.

Note that those contributions, although obtained in the specific context of a hybrid technique, based on AspectJ, to reverse engineer object interactions, should apply more broadly to other techniques to reverse engineer runtime details, not necessarily hybrid (e.g., dynamic) nor based on AspectJ. Our empirical results can be seen as actionable findings that others can build on when devising reverse-engineering technology.

The reminder of this paper is organized as follows. In section II, we review some concepts of aspect-oriented programming and discuss related work. We elaborate on our Light in section III. In section IV, we identify optimizations that would potentially reduce overhead due to Light. In section V, we explain our case studies and the experiments we conducted to answer research questions. We conclude in section VI.

II. Background and Related Work

We first review aspect-oriented programming (AOP) and AspectJ in particular, only focussing on details that relate to our overhead reduction objective. (We assume basic knowledge of AOP and AspectJ.) Next, we discuss works that relate to our approach to reduce instrumentation overhead with a particular focus on AspectJ as instrumentation technology.

A. Aspect Oriented Programming

AOP is a software development and maintenance paradigm that either abstracts away existing crosscutting concerns from core concerns or adds new concerns to core concerns in the form of aspects. In our case, we add new concerns to core concerns. The programming language of the SUS (e.g., C++, Java) drives the choice of AOP technology. The target programming language for the SUS we chose is Java. Different Java AOP technologies exist: e.g., AspectJ [4], DiSL [5], Spring [6], JBoss-AOP [7]. We selected AspectJ [4] because it is widely used and more mature than other technologies.

AspectJ provides constructs to implement individual concerns (i.e., aspects) and their weaving with other (new or core) concerns. New aspects are implemented through advice and inter-type declaration constructs. An advice is a piece of Java code that executes at certain points (i.e., join point) in the SUS, and uses the AspectJ API to collect information about the SUS. AspectJ inter-type declaration constructs provide a mechanism to alter the static structure of the SUS, such as adding new methods or fields. Weaving rules (i.e., pointcuts) select particular join points from the core Java concern (i.e., the SUS) or other concerns where advice(s) must execute.

The AspectJ compiler uses a byte-code weaving approach that first supplies the SUS to the Java compiler, compiles the aspects, and then weaves the compiled aspects into the compiled SUS to generate woven JVM compliant byte-code files [4]. The AspectJ compiler performs advice weaving in two phases: lookup and invocation. Lookup selects a set of advices that applies to each join point whereas invocation runs the selected set of advices on that join point. The compiler performs these in two modes: compile-time weaving (a.k.a. static weaving) executes lookup at compile time and invocation at runtime whereas load-time weaving performs both at runtime [8]. In our experimentations, we use compile-time weaving.

B. Optimizing Instrumentation for Performance

Discussing overhead reduction of a reverse-engineering (or probing) technology generally includes reducing the number of probes, reducing the cost of probing and reducing the cost of data collection (e.g., [9, 10]). The number of probes and the cost of data collection directly depend on the intent of the reverse-engineering activity: the more one wants to collect, the larger number of probes and the more data collected by each probe. Reducing overhead due to those characteristics is exactly what triggered our use of a hybrid technology [3].

In this section we rather discuss overhead reduction of probing, which is about weaving and also a very context dependent issue: one tries to optimize weaving for C++ very differently from Java simply because of the characteristics of the target languages. Some general weaving optimization principles may apply regardless of the technology and one may get inspired by AspectC++ weaving optimization solutions (e.g., [11]) when trying to optimize AspectJ weaving. We do not contribute to AspectJ weaving mechanisms or to the AspectJ language: we consider this outside the scope of this paper.

In the realm of Java programs, different technologies exist to probe behaviour: e.g., AspectJ [4] and DiSL [5] offer high level languages to facilitate probing whereas ASM [12] and BCEL [13] provide APIs to directly work on the byte-code. DiSL [5] is a very appealing, recent solution since the authors argue that it is equally expressive as AspectJ and as efficient as ASM (and more efficient than AspectJ), thereby having the advantages of both kinds of solutions without their drawbacks. The overhead reduction reported by the authors on several case studies, when comparing DiSL to AspectJ is not precise enough to make a decision as to use one technology or the other in a specific context. Specifically, we know (section II.C) that several AspectJ constructs are very expensive (high overhead) and that several attempts have been made to remedy the situation; some authors also suggest efficient usages of some AspectJ constructs to reduce overhead. Unfortunately, the comparison between DiSL and AspectJ does not disclose which of those constructs were used. This is an important piece of information that is missing since, as discussed later, we do not use those expensive AspectJ constructs. Plus, our own overhead study (see below) shows that AspectJ itself is a very small contributor to overhead. We conclude that, at the time of writing, there is no compelling argument showing that, in our context, we should use DiSL rather than AspectJ.

C. Optimizing the Performance of AspectJ Programs

There are two general approaches to improve the performance of an AspectJ program: making an efficient use of the AspectJ language, improving the AspectJ compiler or the JVM. Since we intend to use AspectJ as a toolbox, we only focus on the former and not contribute to the latter: we want to devise an efficient use of the AspectJ language for the purpose of reverse-engineering object interactions.

An efficient use of the AspectJ language requires both efficient weaving rules (pointcuts) and efficient concerns (advices and inter-type declarations). We report on the few works we have found that discuss efficient practices for AspectJ pro-

gramming. Dufour et al. [14] suggest AspectJ programming guidelines for reducing overhead, noticing programmers impose considerable overhead to a base program (i.e., SUS) when they use loose pointcuts, i.e., pointcuts that match too many join points, generic advices (in particular generic `around` advice), the `cflow` pointcut, or when they introduce too many new constructors through inter-type declarations. Similarly, the AspectJ reference books [4, 15] provide recommendations on how to improve the performance of AspectJ programs, such as an efficient use of APIs for dynamic context collection (join point APIs versus Java reflection APIs).

Programming and refactoring of Java programs for performance improvement [16-18] is also related to our work since AspectJ declarations are written in Java. We simply mention them here to indicate that we account for refactoring for performance improvement opportunities in our work.

An efficient implementation of byte-code weaving, i.e., improving the AspectJ compiler, involves optimizing either the AspectJ compiler or the JVM. A number of works suggest optimizations to the original AspectJ weaving mechanisms [8, 19-24], such as the `around` advice, the `cflow` pointcut, or advice dispatch. Although we do not follow this path of instrumentation optimization, we note we do not use computationally expensive AspectJ constructs for which optimizations have been proposed (`around` advices or `cflow` pointcuts). In our experiments, we optimize the implementation of AspectJ concerns, and use the standard AspectJ compiler and the standard JVM.

Since adding new concerns to core concerns adds overhead, an instrumentation alternative could be to manually hard code the new concerns into the SUS in Java instead of using (AspectJ) aspects. Studies [19, 25] reported the performance of an aspect program is equal to or better than the equivalent non-aspect program due to better encapsulation of advices.

Finally, one may argue that AspectJ performs faster with load-time weaving compared to compile-time weaving. On the one hand, compile time weaving reduces the runtime overhead by executing lookup at compile time. On the other hand, opposite to load-time weaving, compile-time weaving is unable to take advantage of runtime data and JVM internal structure to implement optimizations for the inserted aspects [8]. However, empirical studies show that the AspectJ compiler causes lesser runtime overhead with compile-time weaving than with load-time weaving when there is a large number of classes loaded or join points executed [26]. Thus, we opted for compile-time weaving as this paper target industrial-sized software systems.

D. Optimizing Traces for Performance

A dynamic analysis monitors and gathers different types of data from a SUS, and typically stores this data in a file as an execution trace using a specific format for offline consumption [27]. Capturing less data or condensing the gathered data before storage can potentially reduce the overhead of a dynamic analysis. Several trace formats exist for recording object interactions or other kinds of data (e.g., OTF focuses on recording performance data [28]). For obvious reasons, we focus on trace formats for storing object interactions. Prominent works on trace format for storing object interactions discuss encoding, condensing, compacting data in a trace file since such files tend to be huge: [29], [30]. In our context, although these are valid objectives, we are more interested in reducing the overhead due to producing the data than the format of storage of the data in a file. A trade off needs to be found between the overhead due to producing the data, possibly condensed/compacted, and the amount of data to store. In our case, since we privileged overhead reduction and we collect as few data as possible which we believe is already condensed enough, we decided to not use any of the existing condensing/compacting solutions.

Baca minimizes the overhead of producing execution traces within procedures thanks to a hybrid solution [31]. We rather focus on object interactions. Last, path profiling techniques (e.g., [32]) measure the frequencies of path executions. Again, we rather focus on object interactions.

III. Dynamic Analysis in the Hybrid Approach

Our hybrid approach [3] instruments SUS code to generate traces, analyzes source code to create control flow graphs, and then transforms an instance of the trace model and instances of control flow graphs (for several methods) into a UML scenario diagram. We refer the reader to other documents [3, 33] for more details. In this section, we provide self-contained discussion of the dynamic analysis part of the hybrid approach, which we intend to improve, and study possible sources of overhead.

A. Trace Model

Our execution Trace model (Fig. 1) is very close in structure to the UML 2 Superstructure's `Message` components to facilitate transformations to UML scenario diagrams. `Log` represents a single program execution and contains a sequence of `MessageLog`s. A `MessageLog` represents a message sent to the logger to signal the start of an execution between a sending object and a receiving object (the two associations to `MessageLogOccurenceSpecification`). In class `MessageLogOccurenceSpecification`, attribute `covered` is a `String` containing the identification of an object (a unique identifier representing an object of a class). `MessageLog`'s attributes specify the kind of message (`messageSort` attribute), the message's signature, and the name of the class whose instance executes the called method (`bindToClass`). For a `MessageLog` instance, using `bindToClass` and `signature` attribute values, we know exactly which method in a hierarchy of classes actually executed, i.e., the data allow us to account for overriding. In the case of a static call, `bindToClass` contains the class defining this static method. This way, the transformation algorithm can determine the specific class and method invoked by the method call. `SourceLocation` (in `MessageLog`) specifies the location (`name` of the class and `lineNumber`) in the source code from where the logged method call has been made; this is the call site where we can bridge the dynamic information to static information (control flow graph).

B. Light Instrumentation

The Light instrumentation collects information to instantiate the Trace model (Fig. 1) thanks to several AspectJ aspects (`MethodAspect` in Fig. 2, `IdentifierAspect` in Fig. 3), an interface (`ObjectID` in Fig. 4), and a logger class. The latter simply gets trace information (`MessageLogs`) from aspects and writes to a file on disk. The `MethodAspect` aspect defines three pointcuts `callMethod()`, `callStaticMethod()`, and `callConstructor()` lines (3, 4, 5 in Fig. 2) in order to intercept all calls to methods (either static or not, `synchCall` message type) and constructors (`createMessage` message type), thereafter referred to as "method call" for any call type, in a SUS. The pointcuts ensure that advices execute on `call` joint points in the SUS and exclude calls to `objectIDgenerator` and methods in the `instrumentation` package. More accurately, the pointcuts specify `before()` advices to execute on `call` joint points, i.e., before a method call is made (lines 6, 37, 61 in Fig. 2). The `call` joint point enables an advice to collect information about both the caller, i.e. the sending object (e.g., line number (`lineNumber`), and the source file name (`name` of the class) from where the call was made) and the callee, i.e., the receiving object (e.g., class name (`bindToClass`) or object identity (`covered`)). Advices in the `MethodAspect` rely on the capability of the instrumented code to count classes' instances, and report on a unique identifier for each instance, which is achieved thanks to the `IdentifierAspect` aspect and the `ObjectID` interface. Interface `ObjectID` defines method `getObjectID()` which returns the object identity information: a unique `String` for each instance of a given class (line 3 in Fig. 4). Method `getObjectID()` is implemented by changing the static structure of the SUS through inter-type declaration in the `IdentifierAspect` aspect, which adds the implementation of the `getObjectID()` method for every class in the SUS (line 14 in Fig. 3). For each method call (except for the call to a constructor), advices in the `MethodAspect` uniquely identify interacting objects' instances, i.e., caller and callee, through `getObjectID()`. In case of a call to a constructor, the advice cannot capture the object identity information (since the advice executes before object creation has completed); instead the `IdentifierAspect` aspect applies the `objectIDgenerator()` method (through inter-type declaration) for each constructor to automatically record (by calling the logger) the object identity right after its initialization to compensate the missing object identity prior to constructor call (lines 8, 9 in Fig. 3). More specifically, for each class in the SUS,

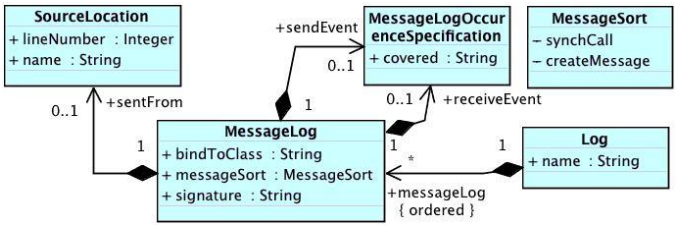

Fig. 1. Trace Model

the static method `objectIDgenerator()` implements an extension of the singleton design pattern to count the number of instances of that class, to assign a new unique number to each new instance of that class, and log the class name and the new instance unique number in the trace file (lines 3 to 12 in Fig. 3).

Note that there is a call to the logger class for each advice

```
1  package instrumentation;
2  public aspect MethodAspect {
3           pointcut callMethod() : call (!static * PackageName..*(..));
4           pointcut callStaticMethod() : call (static * PackageName..*(..))
            && !call (*PackageName..objectIDgenerator(..));
5           pointcut callConstructor() : call (PackageName..new(..)) &&
            !within (instrumentation..*);

6           before(): callMethod ()   {
                    //Content of advice
36          }
37          before(): callStaticMethod () {
                    //Content of advice
60          }
61          before(): callConstructor () {
                    //Content of advice
86          }

87          private static String getLineNumber(String s) {...}
88          private static String getFileName(String s) {...}
89          private static String getBindToClassName(String s) {...}
90          private static String getStaticClassName(String s){...}
91          private static String getStaticBindToClassName(String s) {...}
92          private static String getNewBindToClassName(String s) {...}
93          private static String getMethodSignature(String s) {...}
94          private static String getStaticLifelineName (String s) {...}
95 }
```

Fig. 2. Excerpt of the `MethodAspect` aspect class

```
1  private int ClassName.objectID = ClassName.objectIDgenerator(objectID);
2  private static int ClassName.currentObjectID = 1;
3  private static int ClassName.objectIDgenerator(int i) {
4           int id = i;
5           if(i<1){
6                   LinkedList log = new LinkedList();
7                   id = ClassName.currentObjectID++;
8                   log.add("<lifeline                  className=\"ClassName\"
   name=\"className_" + id + "\"/>");
9                  Logger.getLoggingClient().instrument(log);
10                  }
11          return id;
12 }
13 declare parents : ClassName implements ObjectID;
14 public String ClassName.getObjectID() {
15          if (objectID < 1){
16          objectID = ClassName.objectIDgenerator(objectID);
17          }
18          return "ClassName_" + objectID;
19 }
```

Fig. 3. Excerpt of the `IdentifierAspect` aspect class

```
1 package instrumentation;
2 public interface ObjectID {
3          public String getObjectID();
4 }
```

Fig. 4. `ObjectID` interface

in `MethodAspect` as well as for each object creation (i.e., in method `objectIDgenerator()`) in `IdentifierAspect` to record the captured dynamic information, resulting in one call to the logger for each intercepted method call in the SUS and two calls to the logger for each intercepted constructor call in the SUS. Each time the logger is called, it records the received information to a file on disk; therefore, we can expect a large number of accesses to the disk for any typical program.

We review the collected information for a typical method call by looking at representative advices, the `before():CallMethod()` and the `before():callConstructor()` advices which capture information for calls to non-static methods (Fig. 5) and constructor (Fig. 6). The `before():callStaticMethod()` advice collects similar (though less) information to the `before():callMethod()` advice. We later refer to these advices during our discussion of possible optimizations. The collected information (six and five pieces of data for `before():callMethod()` and `before():callConstructor()` advices respectively as shown in Fig. 5 and Fig. 6) in these advice include: unique identifiers of interacting objects (lines 12, 21 and 66) or classes in case of a static caller (lines 18 and 72), the signature of the method being called (lines 28 and 78) as well as the name of the called class (lines 26 and 76), and the file name from where the call is made, i.e., in the caller (lines 32, 33 and 82, 83). The instrumentation should handle calls between the SUS code and code outside of the SUS code itself (e.g., third party library, JVM/JRE); therefore, each time an advice collects information regarding unique identifiers of interacting objects, it accounts for the possibility that the caller or the callee is not instrumented and therefore does not have a unique identifier: lines 14 to 16, 22 to 24, and 68 to 70.

```
6  before(): callMethod ()   {
7          String thisID = new String();
8          String targetID = new String();
9          LinkedList log = new LinkedList();
10         if (thisJoinPoint.getThis() != null) {
11                 try {
12                     thisID = ((ObjectID) thisJoinPoint.getThis()).getObjectID();
13                  }
14                  catch (ClassCastException e) {
15                         thisID = "Caught NonInstrumentedCaller";
16                 }
17         } else {
18                 thisID =
   getStaticClassName(thisJoinPointStaticPart.getSourceLocation().toString());
19         }
20         try {
21                 targetID = ((ObjectID) thisJoinPoint.getTarget()).getObjectID();
22         } catch (ClassCastException e) {
23                 targetID = "Caught NonInstrumentedCallee";
24                 }
25         log.add("<messageLog bindToClass=\""
26                 +MethodAspect.getBindToClassName(thisJoinPoint.getTarget().toStr
           ing())
27                 + "\" messageSort=\"synchCall\" signature=\""
28                 + MethodAspect.getMethodSignature(thisJoinPoint.toString()) +
   "\">");
29         log.add("         <sendEvent covered=\"" + thisID + "\"/>");
30         log.add("         <receiveEvent covered=\"" + targetID + "\"/>");
31         log.add("         <sentFrom lineNumber=\""
32                 +MethodAspect.getLineNumber(thisJoinPointStaticPart.getSourceLoc
                   ation().toString() + "\" name=\""
33                 +MethodAspect.getFileName(thisJoinPointStaticPart.getSourceLocat
                   ion().toString()) + "\"/>");
34         log.add("</messageLog>");
35         Logger.getLoggingClient().instrument(log);
36 }
```

Fig. 5. `callMethod` advice

```
61 before(): callConstructor () {
62         String thisID = new String();
63         LinkedList log = new LinkedList();
64         if (thisJoinPoint.getThis() != null) {
65                 try{
66                     thisID = ((ObjectID) thisJoinPoint.getThis()).getObjectID();
67                  }
68                  catch (ClassCastException e) {
69                     thisID = "Cought NonInstrumentedCaller";
70                 }
71         } else {
72                 thisID =
   getStaticClassName(thisJoinPointStaticPart.getSourceLocation().toString());
73         }
74
75         log.add("<messageLog bindToClass=\""
76                 + MethodAspect.getNewBindToClassName(thisJoinPoint.toString())
77                 + "\" messageSort=\"createMessage\" signature=\"new "
78                 + MethodAspect.getMethodSignature(thisJoinPoint.toString())+
   "\">");
79         log.add("         <sendEvent covered=\"" + thisID + "\"/>");
80         log.add("         <receiveEvent covered=\"not yet available\"/>");
81         log.add("         <sentFrom lineNumber=\""
82                 + MethodAspect.getLineNumber(thisJoinPointStaticPart.getSourceLo
   cation().toString()) + "\" name=\""
83                 + MethodAspect.getFileName(thisJoinPointStaticPart.getSourceLoca
84 tion().toString ()) + "\"/>");
           log.add("</messageLog>");
85         Logger.getLoggingClient().instrument(log);
86         }
```

Fig. 6. `callConstructor` advice

C. Characteristics of the Light Instrumentation

We characterize the Light instrumentation from four different viewpoints that impact overhead: 1) the mechanism by which data is collected (e.g., use of the AspectJ API); 2) the data (amount and type) being collected (e.g., to instantiate elements of the Trace model); 3) the encoding of the data transferred from one instrumentation component to another or to the

JVM (e.g., the information transferred from aspects to the logger); 4) the logging of data, i.e., the mechanism to store the dynamic information (e.g., recording to a file).

More precisely, these characteristics lead us to systematically discuss sources of overhead in the Light instrumentation. We can study the first characteristic to identify more efficient ways to gather the information such as refactoring pointcuts, advices and our object identification mechanism. The second characteristic may lead to collecting less information at run time and compensate for the missed runtime information by additional static analysis of the code, though this would additionally require that we modify the Trace and the Static models. The third characteristic is about minimizing the generated information from each instrumentation component to minimize the overhead. Finally, for the fourth characteristic we can study different ways of storing dynamic information.

Our intuition from past experiments [3] is that logging and encoding of data are major contributors to overhead, though such contributions have not yet been quantified to warrant optimization activities. Plus, the implementation of the Light instrumentation followed good programming practices to improve maintenance and modularity for instance; Light was not designed with optimization in mind.

IV. Optimizing the Dynamic Analysis

We suggest optimizations for collecting data, encoding data, and logging data. We later report on experiments with different combinations of these optimizations (section V). Studying how to collect less data at runtime is left to future work.

A. On Collecting Data

Optimizing data collecting includes choosing optimized weaving rules, optimizing aspect implementation based on Java refactoring or AspectJ refactoring.

1) Weaving rules:

Optimizing pointcuts involves choosing the right join points for advices, and capturing the chosen join points at the right time during execution. The Light instrumentation intercepts interactions in the SUS with `call` join points and pointcuts execute advices before each method (either static or not) and constructor call. Among many AspectJ join points (e.g., `execution`) the `call` join point is the only one that can capture sufficient data, as required by the Trace model (e.g., identifying callers and callees data), with a minimum number of join points (i.e., with the smallest overhead). Recall that capturing a fewer number of join points reduces the instrumentation overhead (fewer probe points). We decided to keep this instrumentation as we found it adequate from an overhead point of view.

A user of this technology, with a priori knowledge about the SUS could tailor instrumentation to parts of the SUS that are of prime interest, for instance avoiding GUI components.

2) Java refactoring:

In aspects, the choice of data-types and the modularization of the aspect code into functions can impact overhead, i.e., increasing the number of method calls (NMC). The `MethodAspect` and `IdentifierAspect` aspects use a local variable of type `LinkedList` to prepare the logging information to be passed to the Logger. In addition, `IdentifierAspect` weaves global `static` variables of type `int` to each class in the SUS at runtime. Refactoring this Java code with performance in mind may lead to using fewer global variables to increase performance (local variables operate more efficiently in Java). In addition, other Java data-types, such as `StringBuilder`, `String`, `byte`, `short`, may perform more efficiently than `LinkedList` and `int` as they may require less memory. Another refactoring could be to reduce modularity (i.e., reducing NMC) within aspects. For instance the Light instrumentation uses helper methods in its advices; their body could be copied directly into advices at the expense of reusability since we are primarily concerned with performance.

3) AspectJ refactoring:

We can examine different refactorings of AspectJ aspects, such as: 1) the choice of advice, 2) alternative object unique identification, and 3) the choice of APIs for dynamic information collection. With respect to the advice, an `around()` advice would be more expensive than the current `before()` advice. One advantage of a `before()` advice over an `after()` advice is that it keeps the order of invocations in the trace as they happen at runtime; an `after()` would require expensive post-processing to re-construct the correct order. We therefore keep the current `before()` advice.

A different object unique identification mechanism, not logging `objectID` during inter-type declaration, could be to define an `after():execution(constructor)` advice to log the object identification instead of logging during inter-type declaration. However, such an advice would need to collect more data to compensate for the data provided by the missing inter-type declaration. Yet another solution could be to remove all inter-typed `objectIDgenerator()` methods as well as calls to the logger in the `IdentifierAspect` aspect. In this case, we can identify object instances, again with the `after():execution(constructor)` advice, and use a Java data-structure (e.g., a `HashMap`) that counts, stores, and looks up object instances. The JVM does not provide any facility to uniquely identify objects over time; even methods such as `hashcode()` or `identityHashCode()` do not guarantee that two distinct values will be obtained for two different objects.

To collect dynamic data in an advice one can use the AspectJ APIs, the Java reflection APIs, or directly change the Java code (without using any API). Literature indicates that AspectJ performs faster than the Java reflection APIs or a equivalent ad-hoc Java implementation [4, 19]. We thus did not change the use of the AspectJ API to collect dynamic data.

B. On Encoding Data

The (dynamic) trace data is a string of characters which is eventually converted into bytes in computer memory. Passing a lesser amount of data from one instrumentation component to another or to the JVM without losing any piece of data could reduce the amount of computer resources (memory and CPU) and the overhead. The Light instrumentation can be optimized in four ways with this respect: 1) choosing a proper Java character encoding, 2) generating a lesser amount of characters for each log item in advices and inter-typed methods, 3) condens-

ing the data passed by aspects to the logger based on existing trace formats, and 4) changing the mechanism by which data is passed from aspects to the logger.

Aside from the mentioned Java data-types (`LinkedList` and `int`) used to record dynamic data in memory, data is converted to bytes from those data-types based on the character encoding used by the JVM. Using a character encoding such as US-ASCII (7 bit encoding), ISO-8859-1 or UTF-8 (8 bit encodings) can reduce overhead (reduced memory usage).

The Light instrumentation produces dynamic data in a form similar to XMI, which requires some (formatting) processing. Instead, producing raw data, i.e., without formatting (and therefore processing) can reduce overhead. In addition, SUS package and class names with long character strings impose more overhead as we collect the class name information for caller, callee, and object identification. Mapping fully qualified class names to integer identifiers (e.g., with a hash table) can solve this issue, though at the expense of some processing (i.e., use of a hash table); a trade-off needs to be examined and only experiments can tell us what is right. Uniquely identifying object instances with an integer value rather than a string (as currently done in Light) would reduce overhead.

Another way to optimize encoding is to use an existing trace format when producing data (recall the related work section). While such a format reduces memory usage and leads to sending lesser data to the logger, it imposes more computation to actually format the data. The tradeoff needs investigation.

Lastly, the logger currently received log data in a generic way (a `LinkedList` object) regardless of the type of each log item (e.g., logs for constructor are different from others). Instead of such a generic logger, we can investigate loggers specialized to the different types of logs. This would reduce the amount of data passed to the logger (e.g., no need to pass the kind of call); Since the logger is invoked for each method call in the SUS, a gain, even small, for each call can count.

C. On Logging Data

With the Light instrumentation, whenever an advice executes or an object is created, the Logger is called (synchronous call). The logger first reads log items from the passed *`LinkedList`* variable and writes the data to the disk. There are two main sources of overhead here: 1) the high coupling between the log generation process (i.e., aspect instrumentation) and the log storage process (i.e., the logging mechanism) which happen in the same thread of execution. Reducing this coupling would reduce overhead. 2) Writing to the disk adds largely to the runtime overhead, as there is one such operation per method call in the SUS, and two operations per constructor call. In general, there are two main approaches for storing dynamic data: logging locally or remotely.

When logging on the same machine, we can fill a buffer of logs in memory and flush the buffer to disk when it is full (a producer-consumer implementation). This reduces the number of accesses to the disk and therefore overhead, at the expense of longer disk accesses. The performance of this solution also depends on the capacity of the memory and the disk technology (e.g., SSD). In situations where the aspect instrumentation generates a large number of log items, the faster pace of log generation over the pace of log storage may eventually exceed the queue capacity and slow down instrumentation. In addition, although the aspect instrumentation may not interact directly with the logger, the log storage process has a negative effect on the aspect instrumentation as it consumes resources.

When logging remotely, the storage process happens in a log server machine and log items are sent over the network, possibly combined with a buffer on the client (log generation) side. The overhead due to local storage is replaced with overhead due to packets construction according to the selected network protocol and the overhead of sending packets to the log server through the network. The throughput of the network bounds logging: a slow network communication (either due to the selected protocol or the network configuration) causes a bottleneck and consequently hurts performance. In addition, depending on the selected protocol, there is a chance to lose some of the transferred packets from the client to the server.

TABLE I. CHARACTERISTICS OF THE TWO CASE STUDY SYSTEMS

Case study		Classes	LOC	NMC_{SUS}	RQ
Weka	Weka_TC	1,180	238,556	3,993,699	1,2
Producer-Consumer	TC1	9	237	50,002	3
	TC2			225,002	
	TC3			500,002	
	TC4			5,000,002	

V. CASE STUDIES

We designed three experiments to answer each research question (Introduction). This section first discusses the experimental design of these experiments by presenting case studies and our measurement of overhead. We then present results.

A. Case Study software

We rely on two case studies to answer the Research Questions: Table I. Weka, an open source, industry sized data-mining software, is used to understand how much each characteristics of the Light instrumentation contributes to overhead and how much the optimized instrumentation reduces overhead, especially on a large size software. The Producer-Consumer system is a well known, typical producer-consumer: The producer creates an (empty) object, puts the object in a FIFO queue, and then pauses its execution (causing delay by calling the Java `Thread.sleep()` method) for a specified time period (deadline); Simultaneously, the consumer checks the queue constantly to take out any object it may contain, and consumes it. Consuming takes time, which we simulate by executing a deterministic computation (with loops, method calls, object creations). We control the magnitude of this computation (delay), e.g., number of method calls, with a configuration parameter. If the queue is full when the producer wants to deposit an element, the producer throws an exception. This design gives us the opportunity to set a constant delay in the producer, vary the consumer delay (configuration parameter), and study the impact of instrumentation (the computation is traced).

Reverse-engineering experiments require executions. We used a comprehensive test case that comes with the Weka distribution (Weka_TC): it asks Weka to apply multiple classifiers to a dataset. We designed four test cases in the Producer-

TABLE II. THE OVERHEAD OF DIFFERENT COMPONENTS OF LIGHT

		Light	Light\ CallsToLogger	Light\ AdviceObject	Light\ EmptyAdvice	AspectJ Overhead	Light\ NoDiskSave
AspectJ interception mechanisms		x	x	x	x	x	x
Object identification		x	x	x	x		x
Advices	***Object info***	x	x	x			x
	Other info	x	x				x
Logger		x					x
Disk writes		x					

Consumer, each one using a different value for the delay configuration parameter: TC1 with input 10^4, TC2 with input $4.5*10^4$, TC3 with input 10^5, and TC4 with input 10^6.

Table I summarizes characteristics of the two case studies: number of classes (accounting for inner classes) and lines of code (without counting blank lines and comment lines). The NMC_{SUS} column reports on the total number of calls to constructors, static and non-static methods we observed within the SUS, which we computed thanks to a dedicated simple AspectJ aspect. The last column indicates which case study system is used to answer which research question.

B. Overhead Measurement

We evaluate overhead by timing program executions in two ways: the Linux time command (www.gnu.org/software/time) times the difference between the start and end of each program execution; calls to the Java `currentTimeInMillis()` method at the start and end of the SUS. We note that, unlike the latter, the former times the start of the JVM as well as other JVM bookkeeping activities (e.g., garbage collector), and not only the time spent executing the (instrumented) SUS. Since we compare the execution time for different instrumentations with the same time measurement in our experiments, this should not have any impact on our conclusions. In addition, we deemed our number of executions (100 executions of each test case) sufficient to average out such unexpected behaviours. We do not report on the overhead imprint on hardware resources (e.g., CPU and memory usage) since systems with limited resources (such as embedded systems) is out of our scope.

Execution were performed on a Asus machine (laptop) with an Intel(R) i7-3610QM (at 2.3 Ghz * 8), 16 GB of memory and 250 GB Solid-State drive, running Ubuntu 12.4 64x, Open JDK 1.6.0_30, and AspectJ 1.6.7. For the server, we used a Dell PC with an Intel(R) Xeon(R) (at 2.66 Ghz) quad core and 16 GB memory, running WindowsXP 64x, JDK 1.7.0_21. We did not collect any execution time for the server.

C. Experiments

1) Experiment for RQ1

We conducted a set of experiments with Weka and measured execution time (using the Linux `time` command). The objective was to identify the contribution of several components of the Light instrumentation to overhead, specifically, the contribution to overhead of: AspectJ interception mechanisms, that is aspects without advice code (i.e., empty code) and without object identification mechanism and therefore without logging (which we refer to as `AspectJOverhead`); Object identification mechanism only, i.e., inter-typed methods in the `IdentifierAspect`, but not accounting for the lines that prepare and log object ID (referred to as `ObjectInterOverhead`); Aspect advices accounting for log preparation in inter-typed methods, i.e., the code created to capture data to be recorded in trace statements (referred to as `AdviceOverhead`); Logging mechanism though without writing to the disk (referred to as `LoggerOverhead`); Writing data to the disk (referred to as `DiskOverhead`). `AdviceOverhead` includes `AdviceObjectOverhead` and `AdviceContextOverhead`, which therefore indicates the overhead due to inquiring for the object information in advices and the overhead due to collecting and preparing the rest of the information.

We measured `ObjectInterOverhead`, `AdviceOverhead`, `LoggerOverhead` and `DiskOverhead` indirectly. Their direct measure could be done by inserting calls to Java `System.currentTimeMillis()` at adequate places and printing out the result. This would however introduce additional overhead, though small. Instead, we used other measurements and computed the six overhead values mentioned above as discussed next.

In a first experiment, we timed execution while commenting out the calls to the logger in all advices and inter-typed methods. This way the aspect code intercepts everything as in the full-fledged Light version, collects all the required information, but does not send the information to the logger, and the information is therefore not saved on disk. In a second experiment, we timed while not only commenting out the calls to the logger, but also commenting out lines that prepare the object data in aspects (advices and inter-type methods). Again, the aspect code intercepts everything but does not make calls to the logger, nor get object data, nor prepare the log information. In another (3^{rd}) experiment, we timed with empty advices and without object identification, i.e., the aspect code intercepts everything but does not collect any information, does not identify any object instance, does not call the logger, which does not save anything. In another (4^{th}) experiment, we timed with empty advices, though this time including object identification, but commenting out lines for preparing and saving log (i.e., call to logger). In yet another (5^{th}) experiment, we only commented out the statements that save information to the disk in the logger (the lines that write to the file in the logger). The aspect code therefore intercepts everything, collects and sends all the required to the logger, which prepares the trace statements to

- $ObjectInterOverhead = Light\backslash EmptyAdvice - AspectJOverhead$
- $AdviceOverhead = Light\backslash CallsToLogger - Light\backslash EmptyAdvice$
- $AdviceObjectOverhead = Light\backslash AdviceObject - Light\backslash EmptyAdvice$
- $LoggerOverhead = Light\backslash NoDiskSave - Light\backslash CallsToLogger$
- $DiskOverhead = Light - Light\backslash NodiskSave$

Fig. 7. Measurement computations

be saved but does not save anything on disk.

We therefore obtain five different execution times, in addition to the execution time of the full-fledged Light version, respectively (to the above discussion): Light\CallsToLogger, Light\AdviceObject, AspectJOverhead, Light\EmptyAdvice, and Light\NoDiskSave. The components of the Light instrumentation that are activated (or not) in these experiments are summarized in TABLE II. With such measurements, we can compute `AspectJOverhead`, `AdviceOverhead`, `LoggerOverhead`, and `DiskOverhead`: Fig.7.

TABLE III. AVERAGE EXECUTION TIMES (100 EXECUTIONS) IN SECOND FOR EACH PART OF THE LIGHT INSTRUMENTATION (WEKA TEST CASE)

Base	Light	AdviceOverhead		ObjectInter Overhead	AspectJ Overhead	Logger Overhead	Disk Overhead
		AdviceContext-Overhead	*AdviceObject-Overhead*				
0.37	17.50 (%100)	3.57 (%20.4)	0.89 (%5)	0.076 (%0.4)	0.435 (%3)	0.025 (%0.1)	12.5 (%71.4)

Although the suggested overhead measurements are mindful of the overhead of the AspectJ instrumentation and the logging, they do not offer a way to quantify the amount of overhead due to the encoding of information. Designing such an experiment would be difficult since the encoding of information involves many different activities that are dispersed around the instrumentation process. We will indirectly answer this question by measuring the amount of optimization due to better encoding of information after we conduct experiments in response to RQ2. Therefore, instead of measuring the amount of overhead due to this characteristic, we measure the amount of overhead reduction due to implementing encoding optimizations in the Light instrumentation.

2) Experiment for RQ2

We experimented with many of the suggested optimizations separately (section IV). We do not report on each optimization individually in this paper (page limits); instead we packaged different simple, promising (when tried separately) optimizations for collecting, encoding, and logging data in three experiments, and applied all those optimizations in a fourth one. We confirmed that all those optimizations are lossless when compared to Light.

In a first experiment we used `String` instead of `LinkedList` for local variables in advices and for parameters to calls to the logger (`String` proved to outperform other data types we tried) and removed helper methods either by using the AspectJ API or copying their body where necessary. In a second experiment we used raw data instead of the format originally used in Light when transferring data to the logger, using `int` instead of `String` for object unique identifier, using `String` instead of `LinkedList` for local variables, using the ASCII character encoding, and using a specialized logger instead of generic logger for each advice. In a third experiment we optimized the logger by using a custom remote logger communicating over TCP, keeping everything else unmodified. We chose TCP, over UDP, as a reliable protocol to transfer data. Our experimentation with UDP resulted in (trace) data being lost (the amount of data-loss depends on network configurations, network routers, distance between client and server, etc.). We also considered existing logging frameworks such as Log4J, which proved to be inadequate as it led to too much overhead by either not providing a large enough buffer size or collecting unnecessary data, thereby imposing additional overhead. As mentioned earlier, a 4th experiment uses all the optimizations of the first three together. These optimizations are respectively referred to as partially optimized 1, 2, and 3 (PO1, PO2, PO3), and "optimized".

We did not keep many of the optimizations we mentioned earlier due to increased overhead (e.g., using the Java reflection API instead the AspectJ API), or no overhead reduction (e.g., `Hashmap` to uniquely identify objects).

3) Experiment for RQ3

We used the four test cases for Producer-Consumer we mentioned earlier. Except for the value of the delay configuration parameter, all other settings remained the same: the deadline in the producer thread to suspend execution was one second. A test case is considered a failure in this experiment if the instrumented Producer-Consumer losses data (the producer cannot produce); it is a success otherwise. Recall a producer-consumer works well if the consumption rate is greater or equal to the production rate, but produced items will be missed if the consumption rate is smaller than the production rate. The test cases we used ensure that production is slower or equal to consumption in the non-instrumented version. Therefore, all test cases should pass. We measured execution times using the Java `currentTimeInMillis()` method at the start and end of the consumer to compute the delay. We ran three versions of the Producer-Consumer: the non-instrumented one, which we refer to as Base, the original (Light) instrumentation, and the optimized Light instrumentation, which we refer to as Optimized.

4) Executions

Each execution was repeated 100 times, and we will report on averages. The standard deviation of all samples was below 1%, and we do not report on those values. Also, we compared samples with a Student t-test, and differences are always very statistically significant at α=0.05. We confirmed this with the Mann–Whitney *U* non-parametric test.

D. Results

1) Answering RQ1

TABLE III shows execution time values (in seconds) for Base and Light, as well as the contribution (in seconds and percentages) of each characteristic of the Light instrumentation. Writing log items to disk is, as expected contributing significantly to overhead. Unexpectedly, it is by far the largest contributor to overhead (71%). Looking into alternative logging mechanisms to reduce this overhead is therefore paramount. The second highest contributor to overhead (25%) is the advice. Although this is much less than writing to the disk, this is still a considerable amount of overhead compared to Base. Despite our initial intuition, AspectJOverhead amounts to only 3% of total overhead, which suggests the AspectJ interception mechanisms we used are very efficient. The contributions to the overall overhead of object identification (ObjectInterOverhead) and the logger (LoggerOverhead) are negligible. Note

that we consider the ObjectInterOverhead to be representative of the overhead due to object identification. Thus, the inter-typed declarations we used for object identification does not impose much overhead.

2) Answering RQ2

All partially optimized instrumentations indicate overhead reduction (TABLE IV). As we anticipated, optimizing the logging mechanism (PO3) showed the largest amount of overhead reduction (53%). Optimizations on encoding data (PO2) showed the second largest overhead reduction (30%), as optimizing data encoding results in the least data transmission and trace size. Optimizations on collecting data (PO1) showed the smallest amount of overhead reduction. Since there is not much additional optimization we could envision for collecting data, besides collecting fewer data and compensating with additional static analysis, we believe this is another indication of the effectiveness of AspectJ interception mechanisms. We show two execution times for the optimized instrumentation: $time_{SUS}$ is the amount of time to execute Weka_TC and illustrates nearly 74% overhead reduction compared to Light. On the other hand, the client virtual machine has to wait for 285 seconds ($time_{log}$) after the end of the execution of the SUS (Weka) for all the packets to be transferred from the client to the server. The risk, although we have not observed it, is that this may translate into overhead for the SUS if buffers used for network communication get full. The reason for such TCP behaviour is that the TCP congestion control mechanism slows down the rate by which the client packets are sent and consequently creates a bottleneck.

TABLE IV. AVERAGE EXECUTION TIMES (100 EXECUTIONS) FOR NO INSTRUMENTATION, LIGHT, PARTIALLY OPTIMIZED AND OPTIMIZED INSTRUMENTATIONS

Test case		Base	Light	Partially Optimized			Optimized	
				1	*2*	*3*	$time_{SUS}$	$time_{log}$
Weka_TC	Time (sec)	0.37	17.51 100%	14.90 -15%	12.22 -30%	8.34 -53%	4.60 -74%	289.85
	Size (MB)	no trace	1250	1250	460	1250	488	

TABLE V. EFFECT OF INSTRUMENTATION OVERHEAD ON THE PRODUCER-CONSUMER

	Base		Light		Optimized		
	Rate	*Consumer delay*	*Rate*	*Consumer delay*	*Rate*	*Consumer delay*	
TC1	Greater	$1.5*10^{-5}$	Greater	0.229	Greater	0.07_{165}	3.5
TC2	Greater	$4*10^{-5}$	Equal	1.008	Greater	0.32_{36}	14.7
TC3	Greater	$8.5*10^{-5}$	Smaller	2.27	Greater	0.71_{14}	35
TC4	Greater	$77*10^{-5}$	Smaller	23.18	Smaller	5.7_{1}	355

TABLE IV also shows the trace size for each optimization. Among different partial optimizations, PO2 and Optimized show reduced trace size, as expected. The optimized version generates a slightly larger trace due to a mechanism we inserted to monitor transmission of data.

Looking at the size of traces (TABLE IV) and the time spent to write data on disk (DiskOverhead in TABLE III) the Light instrumentation needs, on average, to send data to the Logger at a rate of 250 megabyte/second (mb/s): LightTraceSize/(LightOverhead-DiskOverhead). Our network bandwidth allowed for at most 12.5 mb/s which is far less than 250 mb/s. Therefore, fixing this TCP bottleneck, either by solving the congestion control problem (e.g., using parallel TCP sockets) or using another fast, but reliable protocol (such as SCTP) will increase the network throughput and improve performance, though unlikely to the required rate mentioned above.

We conclude that different criteria must be accounted for and adapted when optimizing the logging mechanism, including network throughput, network bandwidth, overhead due to encoding data for network transmission, system resources (memory and CPU), type of communication protocol, and that these criteria have various, conflicting impacts on a given solution.

3) Answering RQ3

TABLE V shows actual delays due to instrumentation and a coarse grained comparison of relative executions between the consumer and the producer: "Greater" (resp. "Smaller") means that the consumption rate is greater (resp. smaller) than the production rate, i.e., the producer is slower (resp. faster) than the consumer, "Equal" means the two rates are approximately the same (both producer and consumer work at approximately the same pace). The delays in the Base and Light columns indicate the average (over 100 executions) time for the consumer to consume an object. Delay values in the optimized column illustrate two different measures: the first (left) column indicates the average consumption time when the aspects do not have to wait for access to the logging buffer (i.e., the buffer is not full); the subscript indicates the number of queue accesses the consumer performs while this condition holds; the second (right) column indicates the average consumption time when the logging buffer gets full (because of low network transfer rate) and the aspects have to wait for access to the logging buffer. The producer is negligibly affected by the instrumentation as it executes only one method call (to the queue) in case the queue is not full and therefore only one call is instrumented: its execution times are therefore not reported.

For Base (TABLE V), the consumer is faster than the producer in each test case. However, in Light, the consumer is slower than the producer for TC3 and TC4. This is evidence that the instrumentation changes the behaviour of the SUS: the producer cannot produce objects. With our optimization ("Optimized" column), only TC4 shows a change of behaviour, indicating that we reduce the risk of behaviour change due to instrumentation. The amount of delay (second column) for the optimized solution is worse than Light for all four test cases as the instrumentation buffer gets full. The optimized logger is not fast enough at removing log items from the instrumentation buffer and sending them over the network because of the low throughput of the TCP network protocol. This is another incentive for trying to reduce the amount of data to log (and therefore transfer to the server). The number of consumer executions in the first column of optimized (subscript) declines from TC1 to TC4 as the length of consumer executions grows.

VI. CONCLUSION

A hybrid instrumentation strategy is a good idea when reverse engineering object interactions since it benefits from the good characteristics of static and dynamic strategies. Our previous work identified that our hybrid strategy imposed execution time overhead, since it was not designed with performance

in mind but rather with maintainability, understandability objectives (i.e., good design principles), that may become unacceptable if one wants to reverse engineer object interactions from a multi-threaded software. In this paper we therefore tried to precisely characterize and quantify possible sources of overhead. This gave us actionable findings, specifically, which characteristics of our hybrid instrumentation to change (optimize) the instrumentation and to reduce execution time overhead. We implemented a number of such optimizations and experimented on one, industry size case study. We also evaluated to what extent execution time overhead can indeed change the observed behaviour when one reverse-engineers a multi-threaded software. Results indicate that our optimizations can reduce overhead up to 74% compared to our previous hybrid work. However, their benefit diminishes when the system runs long executions because of the low throughput of the TCP protocol. We believe our findings can be used more broadly to reduce instrumentation overhead in other contexts. Other avenues for optimization can be considered, including: a faster, more reliable network protocol (e.g., SCTP), further reducing the amount of data collected at runtime and compensating with more static analysis (e.g., point-to analysis, call graph).

REFERENCES

[1] Ernst M.D., "Static and Dynamic Analysis: Synergy and Duality," *Proc. of ICSE Workshop on Dynamic Analysis*, pp. 24-27, 2003.

[2] Horwitz S., "Precise Flow-Insensitive May-Alias Analysis Is NP-Hard," *ACM TOPLAS*, vol. 19, no. 1, pp. 1-6, 1997.

[3] Labiche Y., Kolbah B. and Mehrfard H., "Combining Static and Dynamic Analyses to Reverse Engineer Scenario Diagrams," *Proc. of IEEE ICSM*, pp. 130-139, 2013.

[4] Laddad R., *AspectJ in Action: Enterprise AOP with Spring Applications*, Manning Publications Co., p. 568, 2009.

[5] Marek L., Zheng Y., Ansaloni D., Bulej L., Sarimbekov A., Binder W. and Tuma P., "Introduction to Dynamic Program Analysis with Disl," *Science of Computer Programming*, pp. 1 - 16, 2014.

[6] Mak G., Long J. and Rubio D., "Spring Aop and AspectJ Support," *Spring Recipes*, Apress, pp. 117-158, 2010.

[7] Pawlak R., Seinturier L. and Retaillé J.-P., "Jboss AOP," *Foundations of AOP for J2ee Development*, Apress, pp. 91-112, 2005.

[8] Golbeck R.M., Davis S., Naseer I., Ostrovsky I. and Kiczales G., "Lightweight Virtual Machine Support for AspectJ," *Proc. of Int. Conference on Aspect-oriented software development*, ACM, pp. 180-190, 2008.

[9] Kumar N., Childers B.R. and Soffa M.L., "Low Overhead Program Monitoring and Profiling," *Proc. of ACM workshop on Program analysis for software tools and engineering*, pp. 28-34, 2005.

[10] Fischmeister S. and Lam P., "Time-Aware Instrumentation of Embedded Software," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 652-663, 2010.

[11] Tartler R., Lohmann D., Scheler F. and Spinczyk O., "Aspectc++: An Integrated Approach for Static and Dynamic Adaptation of System Software," *Knowledge-Based Systems*, vol. 23, no. 7, pp. 704-720, 2010.

[12] Kuleshov E., "Using the Asm Framework to Implement Common Java Bytecode Transformation Patterns," *Proc. of Aspect-Oriented Software Development*, 2007.

[13] Apache Commons, "BCEL: The Byte Code Engineering Library," 2015; http://commons.apache.org/proper/commons-bcel/.

[14] Dufour B., Goard C., Hendren L., Moor O.d., Sittampalam G. and Verbrugge C., "Measuring the Dynamic Behaviour of AspectJ Programs," *Proc. of OOPSLA*, ACM, pp. 150-169, 2004.

[15] Gradecki J.D. and Lesiecki N., *Mastering AspectJ - Aspect-Oriented Programming in Java*, Wiley, 2003.

[16] Mens T. and Tourwe T., "A Survey of Software Refactoring," *IEEE TSE*, vol. 30, no. 2, pp. 126-139, 2004.

[17] Moreira J.E., Midkiff S.P., Gupta M., Artigas P.V., Snir M. and Lawrence R.D., "Java Programming for High-Performance Numerical Computing," *IBM Systems Journal*, vol. 39, no. 1, pp. 21-56, 2000.

[18] Bloch J., *Effective Java*, Pearson Education, 2008.

[19] Hilsdale E. and Hugunin J., "Advice Weaving in AspectJ," *Proc. of Aspect-oriented Software Development*, ACM, 2004.

[20] Avgustinov P., Christensen A.S., Hendren L., Kuzins S., Lhotak J., Lhotak O., de Moor O., Sereni D., Sittampalam G. and Tibble J., "Optimising AspectJ," *Proc. of ACM on Programming Language Design and Implementation*,, 2005.

[21] Bockisch C., Kanthak S., Haupt M., Arnold M. and Mezini M., "Efficient Control Flow Quantification," *Proc. of OOPSLA*, ACM, pp. 125-138 2006.

[22] Avgustinov P., Bodden E., Hajiyev E., Hendren L., Lhotak O., de Moor O., Ongkingco N., Sereni D., Sittampalam G., Tibble J. and Verbaere M., "Aspects for Trace Monitoring," *Formal Approaches to Software Testing and Runtime Verification*, LNCS 4262, Springer Berlin Heidelberg, pp. 20-39, 2006.

[23] Hundt C., Stohr D. and Glesner S., "Optimizing Aspect-Oriented Mechanisms for Embedded Applications," *Objects, Models, Components, Patterns*, LNCS, Springer, pp. 137-153, 2010.

[24] Eric B., Laurie H. and Ondrej L., "A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring," *Proc. of ECOOP*, Springer-Verlag, pp. 525 - 549, 2007.

[25] Lung C.-H., Ajila S. and Liu W.-L., "Measuring the Performance of Aspect Oriented Software: A Case Study of Leader/Followers and Half-Sync/Half-Async Architectures," *Information Systems Frontiers*, pp. 1-14, 2013.

[26] Setty R.B., Dyer R.E. and Rajan H., *Weave Now or Weave Later: A Test-Driven Development Perspective on Aspect-Oriented Deployment Models*, Technical Report, 2008.

[27] Zaidman A., "Scalability Solutions for Program Comprehension through Dynamic Analysis," University of Antwerp, 2006.

[28] Knüpfer A., Brendel R., Brunst H., Mix H. and Nagel W.E., "Introducing the Open Trace Format (OTF)," *LNCS* 3992, Springer, pp. 526-533, 2006.

[29] Reiss S.P. and Renieris M., "Encoding Program Executions," *Proc. of ACM/IEEE ICSE*, pp. 221-230, 2001.

[30] Hamou-Lhadj A. and Lethbridge T., "A Metamodel for the Compact but Lossless Exchange of Execution Traces," *SoSyM*, vol. 11, no. 1, pp. 77-98, 2012.

[31] Baca D., "Tracing with a Minimal Number of Probes," *Proc. of IEEE SCAM*, pp. 74-83, 2013.

[32] Thomas B. and James R.L., "Efficient Path Profiling," *Proc. of ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 46-57, 1996.

[33] Kolbah B., "Reverse Engineering of Java Programs through Static and Dynamic Analysis to Generate Scenario Diagrams," ECE, Carleton University, Ottawa, 2011.

VII. Appendix

In this appendix, we conduct different experiments to find best optimization activities for: collecting data (A), encoding data (B), and logging data (C). We investigate which optimization alternative worsens and which reduces the overhead. In many of experiments in this appendix, we use *Overhead Emulator* (OE) case study, a small and easy to setup program with four classes, which emulates overhead by making method calls (NMC: four millions) and using loops and conditionals. We intentionally adjusted the NMC in OE close to Weka_TC to be able to generalize optimizations for larger systems. We use Linux `time` command for all experiments in this section. For most of experiments (except Log4J experiment), we report on average execution time of 100 executions and confirm the result similar to the process mentioned in V.C.4 (though, we do not report on statistical values in this technical report).

A. Experiments on Collecting Data

In this section, we conduct different experiments to find best refactoring activities suggested in IV.A. These experiments address optimizations on: variable's data-type and declaration type (section 1), dynamic context collection (section 2), and object identification mechanism (section 3). We use OE case study for the first two optimizations and both Weka_TC and OE case studies for the last optimization. As noted before, different refactoring activities would not be impacted by the differences of the two cases study systems.

1) The variable's data-type and declaration type

The choice of data-type and declaration type can potentially change the performance of Light instrumentation. We replace local variable of type `LinkedList` (such as line 9 in Fig. 5) and class variable of type `int` (such as lines 1 and 2 in Fig. 3) in `MethodAspect` and `IdentifierAspect` with other local or global Java data-types to understand which data-type with which declaration type performs more efficiently in the Light instrumentation. We conduct experiments with objects of type `LinkedList`, `StringBuilder` and `String` as well as primary Java data-types `short`, `byte` and `String`[1] where each data-type (or object) is declared either as a class (or aspect) variable or method (or advice) variable. Note that except primary Java data-types all other data-types in Java should be initialized as a new object before assigning to a variable.

a) experiment design:

We conduct nine experiments based on OE to evaluate data-type and declaration type alternatives other than local `LinkedList` and global `int`. We look at data-types `String` and `StringBuilder` to replace `LinkedList` and primary data-types `short` (16 bit) and `byte` (8 bit) to replace `int` (32 bit). Data-types `String` and `StringBuilder` are chosen since they are common Java data-types for immutable and mutable string variables respectively (we do not choose `StringBuffer` since Java documentation suggests `StringBuilder` performs faster). Primary data-types `short` and `byte` are chosen since they require less memory for initialization compared to `int`. In the first experiment, we modify the Light instrumentation by replacing the local variable of type `LinkedList` in advices and inter-typed methods (i.e., `className.objectIDgenerator()`) with the `static` variable of type `LinkedList` in aspects (i.e., global variable) to measure the execution time. In another set of experiments, we conduct four experiments where, in the first two, we replace the local variable of type `LinkedList` in the Light instrumentation with the local variable of types `String` and `StringBuilder` and in the second two, similarly to the first experiment, we replace the local variable of type `LinkedList` with the `static` variable of types `String` and `StringBuilder` in aspects. When we create `StringBuilder` object, we assign the length of log item to the constructor of this object. Note that we initialized local `String` variable as a new object where we used the keyword `new` for a single `String` variable creation throughout the advice or inter-typed methods for the whole log item (in case of the global variable, a single `static String` variable is `new`ed in the aspect). Alternatively, we can pass string literals directly to the `String` variable without using the keyword `new`. We think `String` should be performing more efficiently in this case since JVM uses its string pool for variable initialization. In addition, we suspect there is overhead due to concatenation operation since a local `LinkedList` variable in advice uses the concatenation operation 12 times to form a log item. Therefore, we run another experiment (sixth experiment) to measure execution time where we replace the local variable of type `LinkedList` in advices with 17 and in inter-typed methods with three local variables of type `String` (literals assigned directly). In this case the Logger needs to be modified based on advices and methods. Note that multiple local `String` declarations are negligible in this situation since Java initializes `String` data-type (with direct assignment of string literals) similar to its primary data-types with very little initialization overhead (In fact, our

[1] In this case Java simulates the variable of type `String` similar to primary data-types by assigning string literals directly to the variable.

experiment (not reported here) indicates that the difference between single declaration and multiple declarations is a fraction of millisecond). Similarly to the sixth experiment, we conduct the seventh experiment this time based on local `String` objects with no concatenation operation to measure the execution time. This execution time helps us to better understand the impact of concatenation overhead in the Light instrumentation. Finally, we conduct the last two experiments by replacing the `static` variable of type `int` in the `IdentifierAspect` (lines 1, 2, 3, 4 in Fig. 3) with `short` and `byte` to investigate performance improvement due to using less memory.

It is important to note that when `static` mutable data-types (i.e., global `LinkedList` and `StringBuilder`) replace the local `LinkedList`, unlike the local `LinkedList`, they do not cause any initialization overhead in advices (or inter-typed methods). However, `static` mutable data-types impose an additional overhead due to deletion operation at the end of advices (or methods) by emptying the global variable after passing the log item string to the Logger. There is no such need (i.e., deletion operation) in the case of global immutable variable (i.e., global `String` object) since a new immutable object has to be created when the old object is modified. Note that, except the sixth and seventh experiments, new data-types (either local or global) do not add any new concatenation operation to the ones already exist in the local `LinkedList`.

For all nine experiments, we disable calls to the Logger in advices and inter-typed methods (same as the way Light\CallsToLogger is calculated in TBALE II). This way, we remove the logging overhead from the Light instrumentation and able to better monitor the impact of data-type changes in the collecting of information. Note that we limited our choices of data-type only to those mentioned ones since they satisfy simple string operations efficiently: insert string operation, remove operation, and return string operation. Other Java data-types (such as `HashMap` and `TreeSet`) do other operations efficiently rather than performing basic operations on string literals and integer numbers efficiently.

b) experiment results:

TABLE VI shows the execution times of experimentations on OE with different data-types and declaration-types when no call is made from aspects to the Logger. The second column in the table shows the execution time of the Light instrumented OE when no call is made from aspects to the Logger (i.e., Light\CallsToLogger). We use this execution time as a index to compare with other execution times. Though, we verified the equivalency of the generated trace in all experiments with the trace of Light instrumentation before calculating execution times in the table. TABLE VI shows that both global `LinkedList` and `StringBuilder` data-types (third and ninth columns) performed worse than local `LinkedList`. The slight increase (2%) in the execution time of global `LinkedList` compared to execution time of local `LinkedList` shows that the overhead due removing log item at the end of advices is more than the overhead due to `LinkedList` creation. In the case of global `StringBuilder`, the execution time (due to replacing local `LinkedList`) marginally increased (by 7%). Even the local declaration of `StringBuilder` (in the eighth column) did not perform better than local `LinkedList` (by adding 9% to overhead). The `StringBuilder` object is an array based data-type where a variable-length array contains a sequence of characters (of type `CharSequence`). Therefore, the overhead of inserting new strings of a log item and removing the log item in the local `LinkedList` is still better than the type `StringBuilder` (declared either locally or globally). Note that by assigning the number of characters in the log item to the constructor of `StringBuilder`, we do not change the length of array (and consequently do not add more overhead).

When we replaced local `LinkedList` with a single `String` object, declared either locally in advices (second experiment) or globally in aspects (fourth experiment), execution times (fourth and fifth columns) increased by more than 50 percent. When we removed all concatenation operations and declared 17 local `String` objects in advices (seventh experiment), the execution time (sixth column) reduced by 40 percent. This difference between execution times (when a single or multiple `String` objects declared) indicates the heavy cost of using concatenation operation due to the immutable creation of `String` instances: each time a `String` object is appended with new characters, a new `String` instance is created. As we expected, when string literals are directly assigned to the `String` without using any concatenation operation (sixth experiment), the execution time (seventh column) cut nearly into half of the local `LinkedList` execution time. When we replaced `static int` with `static short` or `static byte` (eighth and ninth experiments), execution times (tenth and eleventh columns) slightly reduced. Though, variables of type `short` and `byte` need to be cast to `int` again before returning the value when calls to the logger are activated. Therefore, it does not worth to change the `int` variables in the Light instrumentation.

TABLE VI. MEAN EXECUTION TIMES (100 EXECUTIONS) FOR DIFFERENT DATA TYPES WITH NO CALL TO THE LOGGER

Time (second) / Case Study	LinkedList (local)	LinkedList (global)	String (local object)	String (global object)	String (distinct local object)	String (local, passing literals)	StringBuilder (local)	StringBuilder (global)	short (IdentifierAspect)	byte (IdentifierAspect)
OE	3.72 (100%)	3.80 (+2%)	5.81 (+52%)	6.02 (+62%)	2.22 (-40%)	2.02 (-46%)	4.05 (+9%)	3.99 (+7%)	3.66 (-1.5%)	3.66 (-1.5%)

2) Modifying dynamic context collection in aspects

In this section, we want to understand the effect of modifying dynamic context collection on the performance of Light instrumentation. More precisely, we ask the following RQ: How does reducing the NMC within aspect, using AspectJ APIs, and using Java reflection APIs change the overhead in the Light instrumentation?

a) experiment design:

We design three experiments. In the first experiment we hardcode all helper methods within `MethodAspect` (lines 78 to 94 in Fig. 2) in advice bodies. In the second experiment, we use AspectJ APIs to replace helper methods within `MethodAspect` and hard code helper methods in advice body if there is no AspectJ API for our purpose. We use the following AspectJ APIs: `getName()`, `getSignature()`, `getFileName()`, `getLine()`. In the third experiment, we replace the AspectJ API we used for capturing object identifiers (e.g., lines 11 to 16 in Fig. 5) with the Java equivalent using Java reflection API (e.g., Fig. 8).

We evaluate experiments based on two criteria: NMC and execution time. Similarly to the previous experiments, after verifying the equivalency of the trace generated from each experiment with the trace from the Light instrumentation, we disable calls to the Logger in advices and inter-typed methods when we measure execution times. This provides better observation over the effect of modifying dynamic context collection. In addition, we are concerned with method calls due to the `instrumentation` package, that is method calls originating from the `instrumentation` package. NMC is the number of places (callers) in the instrumentation package where an invocation (or a series of invocations) is made to another method in the instrumentation package: e.g., advice to logger. We report NMC thanks to a modified version of the Light instrumentation (i.e., manual alteration of its advices) that, in addition to reporting on object interactions in the SUS, reports on calls that take place within the instrumentation package, i.e, the aspects and helper classes. The information we collect for each call includes: the call type (static/non-static/constructor), the caller's name and location, and the callee's name. Note that this heavier (from an execution time point of view) version of the Light instrumentation has the same instrumentation behavior as the Light

```
1           try {
2
3                   Method method =
   ((ObjectID)thisJoinPoint.getThis()).getClass()
   .getMethod("getObjectID");
4                   thisID =
   (method.invoke((thisJoinPoint.getThis())).toString());
5
6           } catch (IllegalArgumentException e) {
7                   e.printStackTrace();
8           } catch (IllegalAccessException e) {
9                   e.printStackTrace();
10          } catch (InvocationTargetException e) {
11                  e.printStackTrace();
12          } catch (Exception e) {
13                  e.printStackTrace();
14          } catch (ClassCastException e) {
15                  thisID = "Caught NonInstrumentedCaller";
16          }
```

Fig. 8. `IdGenerator` class

TABLE VII. MEAN EXECUTION TIMES (100 EXECUTIONS) FOR DIFFERENT DYNAMIC CONTEXT COLLECTION MECHANISMS

Case Study		Light	NoHelper	NoHelper WithAsjAPI	LightWith JavaAPI
OE	Time	3.72 (100%)	3.68 (-1%)	2.22 (-40%)	6.96 (+74%)
	NMC	36,000,018 (100%)	20,000,013 (-45%)	20,000,013 (-45%)	28,000,022 (-22%)

instrumentation (measurement is accurate) and is only used for measuring NMC; it is not used to measure execution time.

b) experiment results:

TABLE VII shows execution times and NMCs in OE for different experiments. The second column (index column) shows the NMC and execution time for the Light instrumentation. The result of the first experiment (third column) shows that even though the NMC was reduced nearly by half, the execution was reduced slightly. This indicates that AspectJ compiler does many of optimizations regarding local method calls that we manually hardcoded in the advice body. However, when we used AspectJ APIs and not used any helper methods (fourth column), the execution time reduced largely (by 40%). This shows the efficiency of AspectJ APIs over the plain Java implementation. When we replaced AspectJ API with Java reflection API in the third experiment (fifth column), the execution time largely increased. Therefore, according to the way we used AspectJ and Java APIs, we can conclude that AspectJ performs more efficiently compared to pain Java or reflective Java APIs implementations.

3) Modifying object identification mechanism

In this section, we examine different modifications in the object identification mechanism for the possibility of overhead reduction. We answer the following RQ with experiments in this section: How does the overhead of Light instrumentation change when a) the callees's object identifier is captured with a new advice instead of inter-type declaration, b) a `HashMap` structure is used instead inter-typed methods?

a) experiment design:

We design two experiments where in the first experiment we replace the lines in `objectIDgenerator()` method in the `IdentifierAspect` (lines 6, 8, 9 in Fig 3) with a new `after():execution(constructor)` advice (Fig. 9) in the `MethodAspect`. In this case, no call to the logger will happen during inter-type declaration. However, such an advice would need to collect class name information dynamically (line 12 in Fig. 9), whereas the class name information is retrieved statically in the missing inter-typed method (line 8 in Fig. 6). In the second experiment, we modify Light instrumentation in that we remove all inter-typed `objectIDgenerator()` methods (Fig. 3) as well as calls to the logger in the `IdentifierAspect` aspect. In this case, we can identify object instances, again with the `after(): execution(constructor)` advice (Fig. 9), and use a Java data-structure that counts, stores, and looks up object instances. For this experiment, we choose the `HashMap` data-structure

(Fig. 10). The IdGen class uses class name as the key to assign values (i.e., objectID) to the HashMap.

We modify IdentifierAspect aspect according to the IdGen class (Fig. 11). With new modifications, every time a call is made to getObjectID() method in the IdentifierAspect, getObjectID() method checks whether the object has yet identified or not (line 4 in Fig. 11). If the object has not identified before (i.e., the value of objectID is null), this method calls getObjectId() method from the IdGen class and passes the class name as a key to it. The method looks up in the HashMap based on class name to check if any value has assigned to the class name (line 19 in Fig. 10). If the value has not been assigned to a class name, getObjectId() method adds class name as a new key to the HashMap and assigns a new value to the new class name (lines 23 to 27 in Fig. 10). Otherwise, that particular class has more than one object instance and the method should return a new identifier for the object of that class (lines 20 to 22 in Fig. 10).

For the first and second experiments, we measure the execution time based on OE and Weka_TC case studies respectively.

b) experiment results:

The third column in the TABLE VIII shows that when we captured callee's object identifier with the new advice in the first experiment, the overhead reduction was negligible in OE. Similarly, use of HashMap and after():execution() advice together in the second experiment (third column in the TABLE IX), slightly reduced the overhead. This confirms our earlier results (section V.D.1) that the object identification overhead is a small fraction of the total overhead in the Light instrumentation.

```
1  pointcut executeConstructor() : execution (PackageName..new(..)) && !within
   (instrument..*);

2  after(): executeConstructor () {
3          String thisID = new String();
4          LinkedList log = new LinkedList();

5                  try{
6                  thisID = ((ObjectID) thisJoinPoint.getThis()).getObjectID();
7                   }
8                   catch (ClassCastException e) {
9                   thisID = "Cought NonInstrumented Constructor";
10                 }

11         log.add("<Lifeline className=\" "
12                 + MethodAspect.getNewBindToClassName(thisJoinPoint.toString())
13                 + "\" name=\" " + thisID + "\">");
14         Logger.getLoggingClient().instrument(log);
15 }
```

Fig. 9. after():excuteConstructor() advice

```
1  package instrumentation;
2  public final class IdGen {
3          private static IdGen singletonInstance;
4          Map<String, Integer> idmap = new HashMap<String, Integer>();
5
6          private IdGen() {
7          }
8
9          public static IdGen getSingletonInstance() {
10                 if (null == singletonInstance) {
11                         singletonInstance = new IdGen();
12                 }
13                 return singletonInstance;
14         }
15
16         public String getObjectId(String key) {
17                 String id;
18                 Integer value = idmap.get(key);
19                 if (value != null) {
20                         value++;
21                         idmap.put(key, value);
22                         id = key + value.toString();
23                 } else {
24                         value = 1;
25                         idmap.put(key, value);
26                         id = key + value.toString();
27                 }
28                 return id;
29         }
30 }
```

Fig. 10. IdGen class

```
1  private String PackageName.ClassName.objectID;
2  declare parents : PackageName.ClassName implements ObjectID;
3  public String PackageName.ClassName.getObjectID() {
4          if( objectID == null )
5                  objectID =
   IdGen.getSingletonInstance().getObjectId("ClassName_");
6          }
7          return objectID;
8  }
```

Fig. 11. Excerpt of modified IdentifierAspect aspect

TABLE VIII. MEAN EXECUTION TIMES (100 EXECUTIONS) FOR LIGHT WITH NEW EXECUTION ADVICE

Case Study	Light	4AdviceLight
OE	15.44 (100%)	15.28 (-1%)

TABLE IX. MEAN EXECUTION TIMES (100 EXECUTIONS) FOR LIGHT WITH HASHMAP

Case Study	Light	HashMap
Weka_TC	17.51 (100%)	17.29499 (-1%)

B. Experiments on Encoding Data

In this section, we investigate different optimization activities suggested for encoding data and their impact of on overhead. We want to answer the following RQs in this section: RQ1: do different Java character encodings change the overhead and if so to what extent? RQ2: how does the use of raw format instead of Light format can change the overhead? RQ3: does the use of a logger with a specialized method for

TABLE X. MEAN EXECUTION TIMES (100 EXECUTIONS) OF DIFFERENT PRACTICES FOR ENCODING DATA

Case study	ASCII	ISO-8859-1	UTF-8 (default)	UTF-16	RawFormat	LargeClassName	AllArgs Logger	FewerArgs Logger
OE	15.05 (-2%)	15.12 (-1.3%)	15.31 (100%)	15.32 (+1%)	11.46 (-25%)	16.47 (+7.5%)	11.40 (100%)	11.28 (-1%)

each advice reduce the overhead? RQ4: How does long class names increase the overhead?

1) experiment design:

We design four experiments to answer each RQ: First, we examine different Java character encodings (US-ASCII (7-bit encoding), ISO-8859-1 (8-bit encoding), UTF-16 (16-bit encoding)) on the Light instrumentation and compare them with default character encoding (UTF-8) of the Light instrumentation based on their execution time. We try different encodings since using a Java encoding with a minimal required space at compile time can reduce the amount of memory being used by aspects. We pass the encoding type as a JVM parameter at runtime, however, we note that SUS should be compatible with the chosen type of encoding.

Second, we change the Light instrumentation in that use raw format instead of Light format. In other words, aspects only pass the required dynamic information to the logger without using any specific format. Processing this information to any specific format is left for after execution. Therefore, we expect reduction on the trace size and consequently on the overhead. The local `LinkedList` variable in `callMethod`, `callStaticMethod`, `callConstructor` advices and `objectIDgenerator` method contains seven, six, six, and two pieces of information respectively.

Third, in order to understand whether modifying the logger to pass fewer arguments from aspects to the logger reduces the overhead, we design two versions of modified Light instrumentation: one with specialized logger and one without specialized logger. In the instrumentation with specialized logger, we modify the `instrument()` method in the logger based on the type of each advice and method. In this case, aspects pass fewer arguments to the logger. We change the Light instrumentation in that `callMethod`, `callStaticMethod`, `callConstructor` advices and `objectIDgenerator` method pass six, five, five, and two variables of type `String`, when literals are assigned directly, respectively to the logger. In the instrumentation without specialized logger, aspects and inter-typed method pass 17 and 3 `String` variables respectively to a logger that does not use of specialized methods (this instrumentation is the same as instrumentation mentioned in sixth experiment of VII.A.1). Then, the comparison between execution times of these two instrumentations should show the effect of passing fewer arguments. Note that generated trace from both experiments should be equal.

Fourth, we examine the impact of long class names on overhead by changing the name of each class OE from a single character name to a 20-character name. This should disclose the whether there is any impact on overhead if SUS contains long class names and to what extent the overhead can change.

We compare the execution times of all experiments based on OE case study. Note that except the second experiment, all other experiments generate trace according to the Light format.

2) experiment results:

Columns two to five in TABLE X (RQ1: first experiment) show no substantial differences in overheads of different Java character encodings, only US-ASCII encoding performed slightly better. However, we note that combining the most efficient character encoding and Java data-type would considerably reduce the overhead for this type of optimization (as shown for the efficient data-type in VII.A.1). sixth column (RQ2: second experiment) shows 25% overhead reduction when dynamic data is only passed to the logger (raw format). In addition, we observed the 66% shorter trace (trace size: 337.6 MB) compared to the Light trace (trace size: 989.6 MB). Eighth and ninth columns (RQ3: third experiment) indicate that specialized logger can slightly reduce the overhead compared to the instrumentation without specialized logger (eighth column). However, this reduction is not considerable in our case study. Seventh column (RQ4: fourth experiment) shows that in the modified OE case study with larger class names indeed increased the trace size (trace size: 1320.8) as well as the overhead. However, despite our initial intuition, class name was not a big source of overhead.

C. Experiments on Logging Data

In this section, we study different optimization activities when logging on a remote machine or the same local machine.

1) Remote logger

We want to understand: RQ1: how does the Log4J framework can change the overhead? RQ2: how does a customized logger based on TCP protocol can change the overhead? and RQ3: how does a customized logger based on UDP protocol can change the overhead?

a) experiment design:

We design three experiments based on OE case study to examine different logging mechanisms suggested by RQs. Note that we do not report on the detailed implementation of each logging mechanism, rather we report on the design of each logging mechanism. In the first experiment, we modify Light instrumentation to log both on client and server using Log4J library. We change `MethodAspect` by adding a new advice to start a new thread on client and establish a client-server connection before the start of SUS. We configure this logger for asynchronous communication (`AsyncAppender`) and increase the buffer size to five gigabytes on the client side. In the second experiment, we replace the generic logger of Light instrumentation with a TCP-based logger on client and server. Similarly to the previous experiment, we change

`MethodAspect` to start a new thread before SUS starts and open a TCP socket on the client. In addition, we use queue as a buffering mechanism on both client and server. Using queue makes aspects, TCP connections, and writes to disk operation (on server) work asynchronously from each other. In the third experiment, we replace the generic logger in the Light instrumentation with a UDP-based logger. Similarly to the second experiment, we start client socket (`DatagramSocket()`) before the beginning of SUS's execution, and use a queue both on client and server.

TABLE XI. MEAN EXECUTION TIMES (10 AND 100 EXECUTIONS) FOR LOGGING MECHANISMS

Case Study		Log4J		TCP		UDP	
		$time_{SUS}$	$time_{log}$	$Time_{SUS}$	$time_{log}$	$Time_{SUS}$	$time_{log}$
OE	Time (sec)	5357	5357	6.67 (-57%)	345.80	6.75 (-56%)	98.55
	Size (MB)	989.6		989.6		974.2	

b) experiment results:

TABLE XI shows the execution time and recovered trace size on the server side based on OE for each experiment. We report on two types of execution times for each experiment: $time_{SUS}$ and $time_{log}$. The $time_{SUS}$ indicates the amount of time spent to execute SUS on the client whereas the $time_{log}$ shows the time span that the client virtual machine has to wait from after the end of the execution of the SUS until all the packets has transferred from the client to the server. Note that we calculated the average execution time from 10 executions in the first experiment and 100 executions for the second and third experiments.

The second column (RQ1: first experiment) shows that using Log4J library not only did not improve the Light instrumentation's overhead, but it worsened the Light's overhead by a very large margin. In addition, the $time_{SUS}$ and the $time_{log}$ were the same in the Log4J experiment. This indicates that even though we enabled asynchronous network communication and allocated a large buffer size, the Log4J framework did not perform well based on requirements of our instrumentation. The trace size shows that Log4J is lossless for transferring data.

The $time_{log}$ in the third and fourth columns (second and third experiments) shows that both TCP and UDP based loggers were able to largely reduce the overhead of Light instrumentation. However, as we noted in V.D.2, $time_{log}$ columns show a large waiting time for the next execution task of the SUS on the client. The trace size of these loggers in the table indicates that the TCP-based logger is a lossless logger, while the UDP-based logger loses on average 2% of total trace size. Therefore, since we are looking for a lossless transfer of information, UDP-based logger would not be a proper logger for our instrumentation.

2) Local logger

We want to understand the performance gain when replacing the generic logger in Light, which writes each log items on file as soon as they are generated, with another customized local logger (so called *CacheLogger*), which buffers logs in memory and decouples the log generation from the logging.

a) experiment design:

We design an experiment based on OE case study to evaluate the performance of a CacheLogger based design. We implement CacheLogger (Fig. 12) based on a producer-consumer design: a log generator thread (i.e., aspects) generates log items, while CacheLogger thread stores log items on disk. The producer thread puts log items, received from the aspects, in a queue as in a memory buffer. We used a `LinkedBlockingQueue` data structure in Java since it is thread safe and maintains the order of logging data. Each log in `LinkedBlockingQueue` is a string representing a "log item". Simultaneously, CacheLogger thread (the consumer thread) removes the `<String>` of log item from the queue and adds the log item to a `LinkedList<String>` *Cache* (the second buffer). If the Cache size reaches the Cache limit, the CacheLogger thread flushes the log items from the `LinkedList` Cache to the disk. The CacheLogger thread uses a variable of type `BufferedWriter` to take log items off the Cache and to flush them to the disk. Depending on a system configuration and resources, different Cache sizes may perform differently. The user can change the Cache size to understand with which Cache size the CacheLogger thread has the best performance. With our configuration, our empirical analysis shows that setting the Cache size to 10^4 bytes results in the best performance in the CacheLogger. When aspects (the producer thread) finish generating log items, which indicates the end of execution of the Light instrumentation of the SUS, the aspect code calls the `forceFlush()` method in the CacheLogger. The `forceFlush()` method removes the remaining items from the queue, adds them to the Cache, and flushes them to the disk. It is imperative to synchronize both cache and queue during the logging process to not lose any log item when the consumer thread removes all log items from the queue.

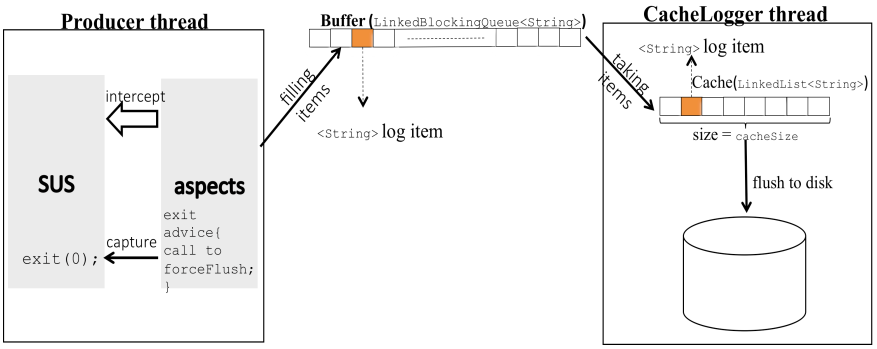

Fig. 12 CacheLogger design

TABLE XII. MEAN EXECUTION TIMES (100 EXECUTIONS) FOR CACHELOGGER AND LIGHT

Test case	Disk techn.	Base	Light	CacheLogger (cacheSize=10k)	
				$Time_{SUS}$	$Time_{Log}$
OE_TC2	SSD	0.085	15.44 (%100)	6.88 (%-55)	0.18

b) experiment results:

TABLE XII shows execution time without Instrumentation (third column), Light instrumentation with the generic Logger (fourth column), and Light instrumentation with CacheLogger (fifth column). Similar to past experiments, $Time_{SUS}$ shows the time span from the beginning until the end test case OE_TC2 and $Time_{Log}$ shows the time required for logging after the completion of the test case. The %55 performance improvement of CacheLogger (i.e., $Time_{SUS}$) over the Light indicates that simple logging practices (e.g., leaving the file open when logging, buffering to reduce accesses to disk) can make a big difference in reducing the total overhead. It is worth noting that we used SSD technology our experiment, we do not expect that HDD technology can perform as good as SSD technology. To this date, a large capacity SSD disk remains an expensive storage device compared to HDD disk.