IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received June 8, 2019, accepted June 22, 2019, date of publication July 1, 2019, date of current version July 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2925855

Field Monitoring With Delayed Saving

OSCAR CORNEJO, DAVIDE GINELLI, DANIELA BRIOLA, DANIELA MICUCCI, AND LEONARDO MARIANI

Department of Informatics, Systems and Communication, University of Milano-Bicocca, 20126 Milan, Italy

Corresponding author: Oscar Cornejo (oscar.cornejo@unimib.it)

This work was supported in part by the ERC Consolidator Grant 2014 Program through the EU H2020 Learn Project under ERC Grant 646867.

ABSTRACT Field monitoring techniques can collect data about the behavior of software applications while running in the field, with real users and real data. Developers can exploit the information extracted from the field to timely improve, tune, and fix their systems, anticipating feedback by users. It is, however, challenging to extract a relevant amount of information from field executions without introducing significant overhead. This paper addresses this challenge by studying how to inexpensively trace data in-memory while postponing save operations to idle time, so that the operations requested by users are exposed to a negligible overhead only. In particular, this paper presents delayed saving, a technique that efficiently traces references to objects, opportunistically saving information only when the monitored application is not serving any user request. Storing references and postponing save operations may introduce inaccuracy in the collected data, that is, a lately saved object might be in a different state compared to the state of the object at the time it was traced. Our evaluation shows that the level of inaccuracy introduced by delayed saving is limited compared to its efficiency and low intrusiveness.

INDEX TERMS Monitoring, tracing, dynamic analysis.

I. INTRODUCTION

Collecting data from the field, that is, while applications run in a real environment with real users, is extremely important because it can reveal how programs are actually used. For instance, field monitoring techniques can be used to extract frequent usage scenarios [1], profiling data [2], popular configurations [3], behavioral information [4], [5], and failure reports [6], [7]. Using field data as part of the development process is increasingly popular, especially with the growing adoption of DevOps practices [8], which systematically exploit automation and monitoring tools to reduce the gap between development and operation [9], [10].

Monitoring the software directly in the field is however challenging. Indeed, the monitoring activity may interfere with the user activity causing annoying slow downs, which may negatively affect the quality of the user experience. It is thus important to be able to collect data non-intrusively, balancing the amount of collected information with its completeness and accuracy. This is particularly true for *interactive applications*, that is, reactive applications designed to continuously interact with users.

Existing techniques reduce the impact of monitoring by limiting the information that is collected from the field [4], [5], [11]–[14]. Although this might preserve the quality of the user experience [15], it significantly reduces the amount of collected data, slowing down the extraction of knowledge from the field while introducing the significant risk of missing rare events in the collected observations. In this paper, we propose to reduce the intrusiveness of field monitoring using a radically different approach: *reducing the accuracy of the collected data instead of reducing the amount of collected data*. This can be particularly beneficial to all those techniques that do not require perfect observations but can work with almost correct data, such as user profiling techniques [2], which focus on statistically relevant evidence rather than perfect evidence, failure reproduction techniques [16], which can work with inaccurate data as long as it is sufficient to reproduce failures, and mining techniques [17], [18], which focus on the generation of accurate, but not necessarily perfect, models.

Following this intuition, this paper presents *Delayed Saving*, a technique that exploits the characteristics of interactive

The associate editor coordinating the review of this manuscript and approving it for publication was Mervat Adib Bamiah.

This work is licensed under a Creative Commons Attribution 3.0 License. For more information, see http://creativecommons.org/licenses/by/3.0/

applications to collect sequences of function calls and the associated objects (e.g., the set of objects that can be reached from the parameters and the reference to the receiving object) introducing negligible overhead, at the cost of a limited loss of accuracy. Delayed Saving collects minimal information while the user is effectively interacting with applications (e.g., it just traces the sequence of methods that are executed and the references to the associated objects), and postpones data recording to idle time.

In particular, Delayed Saving observes the status of an interactive application, which can be either *idle* (i.e., the application is not serving a user request) or *working* (i.e., the application is serving a user request), by capturing user interactions and sampling the CPU consumption. This information is exploited to optimally activate and deactivate data recording, thus minimizing its intrusiveness.

Opportunistically saving the data while the application is idle has the advantage of not affecting the user experience because users do not experience any significant slow down. On the other hand, data is saved asynchronously compared to the time when the recording should actually happen and this may introduce a loss of accuracy. For instance, if the value of a parameter is recorded sometime after a method is invoked, the value of the parameter at the time of the recording could be different than the value at the time the method was called, and thus the resulting trace might be inaccurate.

We evaluated Delayed Saving with two popular open source applications, Eclipse [19] and Open Hospital [20], and discovered that in practice it can introduce a huge gain in performance at the cost of a limited loss of accuracy. We also confirmed the possibility of exploiting the collected data in relevant tasks by applying the Daikon data mining technique [17] to the collected data and demonstrating that the loss of accuracy has minor impact on the resulting models.

In a nutshell, the major contributions of this paper are: (i) Delayed Saving, a novel monitoring technique that can save a significant amount of data about function calls and the associated objects without annoying users, at the cost of a small loss of accuracy; (ii) an evaluation that assesses Delayed Saving in terms of performance and data accuracy; (iii) an investigation of the impact of the inaccuracy in the collected data when exploiting the Daikon mining technique.

The paper is organized as follows. Section II presents a simple study that quantifies the saving that can be achieved by temporary storing values in memory instead of immediately persisting them, as Delayed Saving does. Section III describes the Delayed Saving technique. Section IV discusses how the technique can be configured to address a specific application. Section V presents the empirical validation we conducted to assess our technique. Section VI discusses the results. Finally, Sections VII and VIII discuss related work and provide final remarks, respectively.

II. IN-MEMORY TRACING VS SAVING TO FILES

To quantify the gain that can be obtained by saving data to file during idle time and tracing values in-memory during working time, we designed a small and simple motivating experiment.

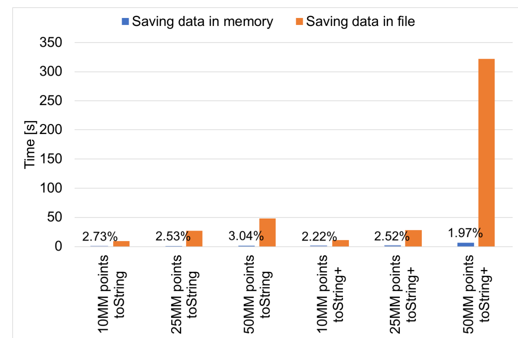


FIGURE 1. Time performance comparison when saving to memory and to file.

We implemented a small program in Java that creates a list of objects of a `Point` class with attributes `x` and `y`, and then saves them either in memory or into a file. We extract data about the points using the `toString` method. To study what happens when a non-trivial amount of data has to be collected, we also produced a modified version of the `toString` method that returns a string with a long message in addition to the x and y coordinates of the point.

Figure 1 shows the time necessary to save a variable number of objects and variable amount of information in memory and into file (`toString+` indicates the modified `toString` method). The number on top of each bar representing the cost for saving data in memory indicates the percentage of time needed to save to memory with respect to the time needed to save to file. As expected, the reduction is impressive, ranging from 97% to 98%, and is independent on the amount of information that is recorded. This example further confirms the intuition behind Delayed Saving, that is, since the difference between writing to file and saving in memory is definitely large, we could exploit the idle phase to save the data collected by the monitoring activity, while exploiting the in-memory tracing during the working phase, drastically reducing the overhead introduced by monitoring.

III. DELAYED SAVING

The concept of Delayed Saving is exemplified in Figure 2. The figure shows the time evolution of a typical execution in an interactive system, distinguishing the working (grey area) and idle (white area) states. The solid line represents an execution without monitoring: the CPU usage level significantly increases when the application is working and significantly decreases when the application is idle. When we introduce a monitoring approach that directly saves the observed data, a significant overhead can be experienced during the working state since data is saved synchronously with the execution, while almost no overhead is experienced in the idle state. The resulting behavior is represented by the thin dotted line shown in Figure 2. Delayed Saving completely changes this scenario by collecting minimal amount of information during the working state, to postpone the time consuming saving

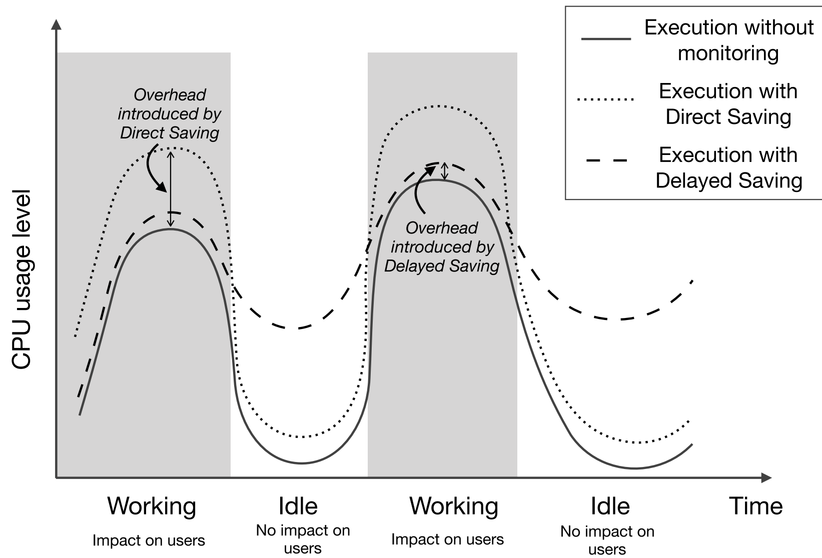


FIGURE 2. Comparison of performances for different monitoring approaches.

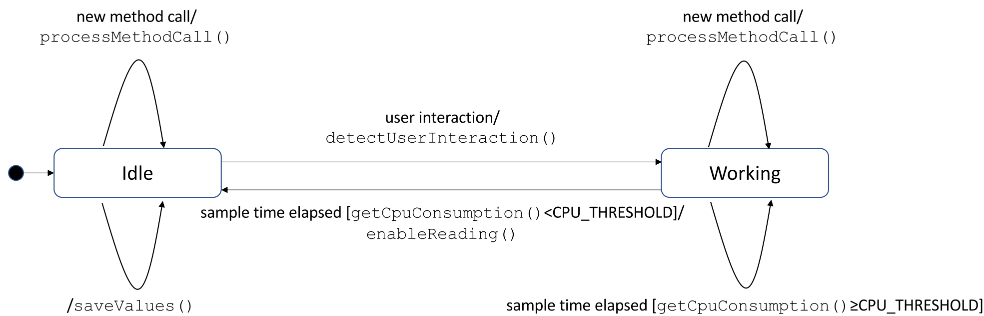


FIGURE 3. Statechart capturing the behavior of delayed saving.

operations to the idle state, when they do not interfere with the user activity. The resulting behavior is represented by the thick dotted line shown in Figure 2.

In the working state, Delayed Saving maintains a collection of references to the data structures and objects that will be later saved (*In-memory tracing*). Since the collection is hosted in memory and the traced data does not require any manipulation beyond saving references, the collection can be populated with a large number of references in a computationally inexpensive way.

When the monitored application enters the idle time, the data traced in memory is *saved* into a persistent storage. This operation is typically expensive because both a significant amount of data has to be written and the access to persistent storage is orders of magnitude slower than access to memory. However, since this operation is performed during idle time, the user of interactive applications do not experience any slowdown.

Although the concept behind Delayed Saving is valid for different kinds of events and data that can be monitored, let us specifically consider the case of a monitor that must collect the sequence of method calls that are executed by an application, to later reconstruct and analyze its internal behavior. Tracing method calls is common to many techniques and applications (e.g., reproducing failures [16], detecting malicious behaviors [21], debugging applications [22], profiling software [2], optimizing applications [23], and mining models [18], [24], [25] which is also the case we studied empirically in our evaluation). In such a case, Delayed Saving collects the sequence of methods that are executed and stores the references to the objects used as parameter during the working state.

Listing 1 shows the pseudocode of the algorithm. It is composed of five main functions: `processMethodCall`, which is executed every time a method call is intercepted; `getCpuConsumption`, which is executed according to the sampling rate to detect changes in the CPU consumption; `enableReading`, which is executed to enable the access to the buffer when the application is detected to be idle; `detectUserInteraction`, which is executed every time the user performs an action; and `saveValues`, which keeps running to save values to a file. The behavior of these functions is illustrated in the statechart reported in Figure 3, which visually shows the functions activated in each state and the events that cause the transitions.

```
1 Buffer buffer = new Buffer();  /* Allows
      concurrent access */
2 Sample lastSample;
3
4 /* Executed when a method call is
      intercepted */
5 void processMethodCall(MethodCall call) {
6   Event event = new Event();
7   event.addSignature(call.getSignature());
8   event.addRefToArgs(call.getArgs());
9   buffer.addEvent(event);
10 }
11
12 /* Executed according to sampling rate */
13 double getCpuConsumption() {
14   Sample currSample = getSample();
15   double cpuConsumption =
16     (currSample.cputime-lastSample.cputime)
      /(currSample.time-lastSample.time);
17   lastSample = currSample;
18   return cpuConsumption;
19 }
20
21 /* Executed when cpuConsumption<
      CPU_THRESHOLD */
22 void enableReading() {
23     buffer.setReadable(true);
24 }
25
26 /* Executed when the user performs an
      action */
27 void detectUserInteraction() {
28   buffer.setReadable(false);
29 }
30
31 /* Saves values from buffer to file */
32 void saveValues() {
33   while(true) {
34     /* Blocking read: it reads a value only
       if the buffer is readable and not
      empty */
35     Event event = buffer.readEvent();
36
37     /* It retrieves parameter values using
      the refs in the event */
38     FileWriter.write(event);
39   }
40 }
```

Listing 1. **Delayed Saving pseudocode.**

The Idle state inexpensively traces in-memory the method calls that may occur by running the `processMethodCall` function, which stores the signatures of the invoked methods and the references to parameter values. In the Idle state, the function `saveValues` is continuously executed to read values from the buffer and write them in a file. The reading operation is blocking, that is, the call to the function does not return until the buffer is readable and includes values to be written. A user interaction causes the execution of the function `detectUserInteraction`, which in turns sets the buffer as not readable, since the monitored application is now in the Working state.

In the Working state, events are still inexpensively traced in-memory by the `processMethodCall` function. In addition, the CPU consumption is regularly sampled by activating the function `getCpuConsumption`, and when a value below the threshold is detected, the buffer is marked as readable by the `enableReading` function and the current state changes into the Idle state.

Depending on the size, number, and complexity of objects, writing values to a file may require some time and computational resources, but it does not interfere anyway with the user activity. Thus compared to directly saving the collected data to file, Delayed Saving still collects the required data (in this example the sequence of method calls together with the values of the objects passed as parameters) while having negligible impact on the overhead, as reported in our evaluation.

Of course, postponing data saving may imply recording values that are not the same than at the time they were traced, which is at the time a method was invoked in the example. We studied this aspect in our evaluation and demonstrated that the degree of inaccuracy introduced by Delayed Saving is still compatible with relevant usage scenarios of field data.

Delayed Saving has to automatically detect when an interactive application switches between the idle and working states. To do this, it exploits two information: the generation of user events and the value of the CPU consumption. In particular, an application switches from idle state to working state as soon as a user event (e.g., a click, a keyboard input, a scroll, etc.) is detected. An application switches from a working state to an idle state when the CPU consumption, which is sampled at a given rate, decreases below a given threshold value. We use the CPU time to compute CPU consumption to avoid any interference with other processes and I/O operations [26].

Both the value of the CPU consumption threshold and the sampling rate are configurable parameters of the technique. These parameters may determine the early, optimal, or late activation and deactivation of data saving. We discuss in the next section how the CPU consumption threshold is empirically determined for a specific application so as to minimize the interference between the monitoring system and the application. We also empirically study different CPU sampling rates to determine their impact on the performance of the technique and accuracy of the data.

IV. CALIBRATION OF THE CPU CONSUMPTION THRESHOLD

Delayed Saving requires the definition of a threshold on CPU consumption that can be used to determine when the monitored application has completed serving a request and is thus moving from the working to the idle state. To calibrate this value and to avoid both early activations, which may cause the monitor to interfere with the user activity, or late activations, which may cause the monitor to loose time that can be exploited to save values, we defined a calibration procedure.

The calibration procedure is defined as an optimization problem that takes a set of perfectly classified samples in input and experiments different threshold values until it identifies the optimal value that will be used in the field. To obtain perfectly classified values, a subset of the application functionalities are manually instrumented to collect when the computation exactly starts and ends. Since this procedure must be performed manually, it can be performed only for a selection of the functionalities and not for the full application.

Thus it cannot be cost-efficiently used as general mechanism to detect the transitions between the working and the idle states.

The samples collected for the calibration have a timestamp, a CPU usage level, and a ground truth classification derived by the knowledge of when the user operation started and then finished its computation. We actually run Delayed Saving to associate a label that represents the status of the monitor (*saving* or *not saving* data) with each sample. The ideal behavior corresponds to a monitor that only saves data when the sample belongs to an idle state, and does not save data when the sample belongs to a working state. The accuracy of the behavior of the monitor can thus be measured as the combination of its accuracy with the idle samples and its accuracy with the working samples.

More formally, given a threshold t and a set of samples S, if *idlePos* is the number of idle samples associated with active saving, *idleNeg* is the number of idle samples associated with inactive saving, *wkPos* is the number of working samples associated with inactive saving, and *wkNeg* is the number of working samples associated with active saving, the accuracy resulting from the application of the threshold t to the set of samples S is:

$$calib(t, S) = 0.5 * \frac{idlePos}{idlePos + idleNeg} + 0.5 * \frac{wkPos}{wkPos + wkNeg}$$

The calibration procedure returns the value of t that maximizes *calib(t,S)*.

V. EMPIRICAL VALIDATION

This section presents the design of our empirical evaluation: we briefly describe the prototype tool that we used for the assessment (Section V-A), we then discuss the research questions (Section V-B), the experimental subjects (Section V-C), the calibration procedure (Section V-D), and finally the design of the experiment (Section V-E). We discuss the results in the next section.

In this evaluation we focus on the collection of method calls and parameters values, which is an information commonly recorded by many techniques [2], [16], [18], [21]–[25].

A. PROTOTYPE TOOL

Our prototype tool is implemented in Java and uses JNativeHook [27] to intercept user interactions (function `detectUserInteraction` listed in Listing 1 is executed when an interaction is intercepted) and OSHI [28] to sample CPU consumption values.

We use AspectJ [29] to design the monitor that collects method calls and parameter values (function `processMethodCall` listed in Listing 1 is executed every time a method call to be traced is intercepted). These values are written in-memory and then saved by Delayed Saving. They are otherwise directly written to file exploiting the normal Java flushing policies by Direct Saving.

B. RESEARCH QUESTIONS

With this experimental evaluation, we aim to answer the following research questions:

- **RQ1: How do the overhead and memory consumption introduced by Delayed Saving compare to Direct Saving?** This research question compares Delayed Saving, which delays save operations to idle time, to Direct Saving, which does not delay operations, both in terms of overhead and memory consumption.
- **RQ2: Is the data collected by Delayed Saving accurate?** This research question quantifies the degree of accuracy of the traces collected by Delayed Saving.
- **RQ3: Can Daikon generate accurate models from the data collected by Delayed Saving?** This research question studies if the traces collected by Delayed Saving are accurate enough to use machine learning techniques, such as Daikon, to mine information about the behavior of the software.

C. EXPERIMENTAL SUBJECTS

To answer these research questions, we applied Delayed and Direct Saving to two very diverse projects: the Eclipse IDE [19] and the Open Hospital application [20]. In the case of Eclipse, we monitored the execution of methods within the *JDT Plugin*, which provides a number of views, editors, wizards, builders, and code merging and refactoring tools for Java development. In the case of Open Hospital, we monitored the execution of the methods within the packages managing patients, hospital admissions, and accounting.

To run the applications we implemented two automatically executable Sikulix [30] test cases exercising the monitored functionalities. In the case of Eclipse, we designed a test case that creates a new Java Project with three classes, generates getters, setters, constructor, `toString`, and `main` method, performs multiple search operations, refactors the code, cleans the project, and finally generates the documentation of the project, for a total of 48 user operations. In the case of Open Hospital, the test case adds three new patients to the system, changes their information, creates a bill with the exams to be performed for every patient, and finally deletes all the patients and the related bills, for a total of 36 user operations.

D. CALIBRATION

Since Delayed Saving is a technique whose performance depends on the selected threshold level for CPU consumption, we ran the test cases and applied the calibration procedure presented in Section IV to both subject applications. Figure 4 shows the boxplot of the CPU consumption values observed during the idle and working states for Eclipse and Open Hospital. We can notice that the populations of values are significantly different depending of the state of the application, confirming the intuition that the CPU consumption value can be reliably used to deactivate the monitor. The calibration procedure finally identifies a threshold value equals to 2.75% in Eclipse, and equals to 7.26% in Open Hospital.

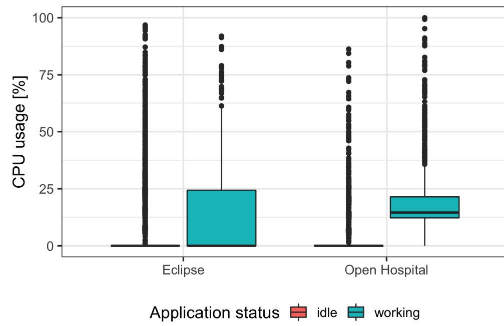


FIGURE 4. CPU consumption values for Eclipse and Open Hospital in the idle and working states.

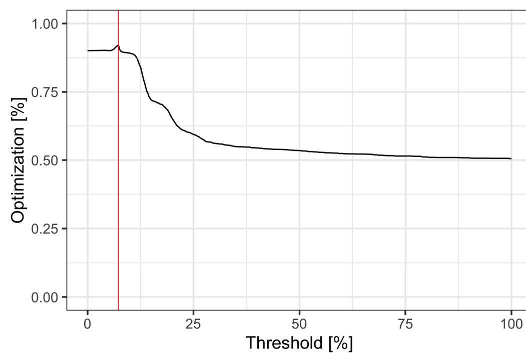


FIGURE 5. Threshold selection for Open Hospital.

Delayed Saving is relatively sensitive to the choice of the threshold. For instance, Figure 5 shows how the accuracy of the technique changes for different values of the threshold for Open Hospital. The vertical line corresponds to the value returned by the calibration procedure that we also used in our evaluation. We can notice that values up to 15% all produce similar accuracy levels.

E. EXPERIMENT DESIGN

To answer RQ1, we executed the automatic test cases in multiple settings: (1) without having any monitoring technique in place, (2) with Direct Saving, and (3) with Delayed Saving. Each execution has been repeated five times. We computed the overhead that Direct Saving and Delayed Saving introduce in each single user operation (seen as any operation starting with a request by the user to the application using the GUI or using the keyboard, and ending when the application giving a feedback to the user) with respect to executions without monitoring. We expected Delayed Saving to introduce a definitely lower overhead than Direct Saving, but still some overhead might be observed also for Delayed Saving due to in-memory tracing and possible delays in switching off saving when the application moves from the idle state to the working state.

To investigate the performance of the approaches for a different amount of recorded data, we considered three cases (called recording levels): *Objects Recording*, which records the values of the attributes in a monitored object without exploring the referred objects recursively, *Objects + Attributes Recording*, which is the same than Objects Recording but it also records the values of the attributes in the object recursively for one level, and *Objects + Attributes Recursively Recording*, which records the attributes in the referred objects recursively for two levels. For instance, a same object of type *Person* is recorded as *Person = [age: 39; car = Car]* by *Objects Recording*, as *Person = [age: 39; car = [brand = Fiat; color = white; trans = Transmission]]* by *Objects + Attributes Recording*, and as *Person = [age: 39; car = [brand = Fiat; color = white; trans = [type = manual; nrGears = 5]]]* by *Objects + Attributes Recursively Recording*.

For Delayed Saving we investigated the performance also considering three CPU sampling rates (20ms, 200ms, 1s). In total we studied 9 configurations for Delayed Saving (3 recording levels for 3 sampling rates), and 3 configurations for Direct Saving (3 recording levels, since this technique does not use CPU sampling).

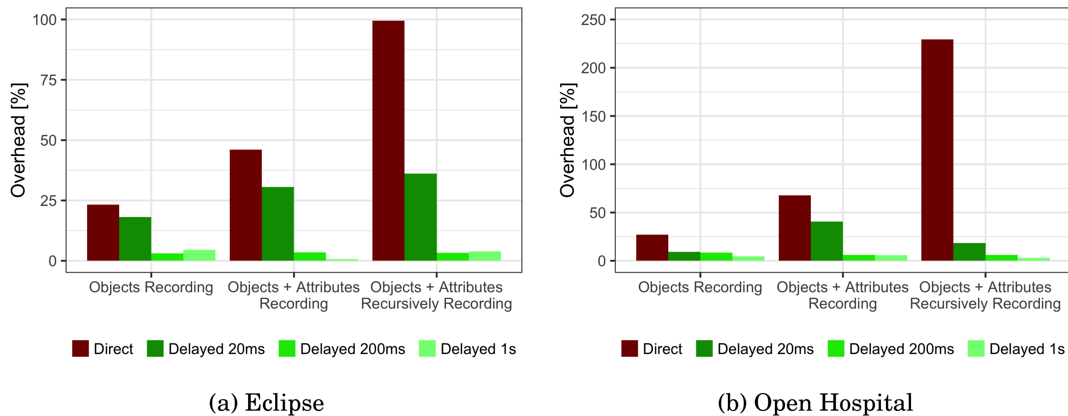
To investigate memory consumption, we sampled the amount of consumed memory for the same configurations considered for performance analysis, and we compared memory consumption to the case no monitoring is in place.

To answer RQ2, we computed the accuracy of the data recorded by Delayed Saving in comparison to the data collected by Direct Saving, which is accurate by definition. To this end, we defined three quality metrics: Recorded, Missing, and Unsound. *Recorded* indicates the percentage of data collected by Delayed Saving that is in common with the data collected by Direct Saving. *Missing* indicates the percentage of data that Delayed Saving was not able to capture. The sum of Recorded and Missing is 100%. We also compute *Unsound*, that is, we measure the percentage of data reported by Delayed Saving that has no counterpart in the traces collected by Direct Saving. For instance, an object that changes the values of its attributes due to postponed data recording causes the generation of both some Missing values, because the original values are not present in the trace recorded by Delayed Saving, and Unsound values, because new values with no counterpart in the original file recorded by Direct Saving are present.

Also in this case, we considered 9 configurations for Delayed Saving (3 recording levels for 3 sampling rates).

To answer RQ3, we merged the data collected for the execution of the same method and run Daikon to mine method pre- and post-conditions.[1] We did this for both traces collected by Direct Saving and by Delayed Saving considering the three recording levels and we computed the percentage of shared pre-conditions.

[1]Daikon is a mining technique [17] that can mine likely properties, including invariants and method pre- and post-conditions, from execution traces. Examples of expressions that Daikon can learn are $x.field > abs(y)$, $z > 0$, where x is a reference to an object, and y and z are numeric variables.



(a) Eclipse (b) Open Hospital

FIGURE 6. Performance results for all levels.

VI. RESULTS

In this Section we both present the results obtained for the investigated research questions and discuss the threats to validity. Since the monitored applications are normally installed and executed in the end-user environment, we executed all the experiments in a machine running Windows 10 Pro with a 3.20 GHz Intel Xeon E5 processor and 8 GB of RAM.

A. RQ1: HOW DO THE OVERHEAD AND MEMORY CONSUMPTION INTRODUCED BY DELAYED SAVING COMPARE TO DIRECT SAVING?

The results about the overhead introduced by Delayed Saving and Direct Saving for all the considered configurations and subject applications are shown in Figure 6. We can notice that in all the cases Delayed Saving introduces an overhead that is significantly lower than Direct Saving, and for some configurations the reduction reaches an order of magnitude. The reduced overhead, which is already remarkable when recording little information about parameters (*Objects Recording* level), is more and more significant with an increasing amount of recorded data. Indeed, *Objects + Attributes Recursively Recording* level is almost infeasible for Eclipse, where the slow down reaches 99.5% with Direct Saving, and even more for Open Hospital, where the slow down can reach 229.39%. On the contrary, Delayed Saving (with a CPU sampling rate of 200ms and 1s) introduced an overhead that is always lower than 9%. Even for the monitoring configurations that cannot be feasibly addressed with Direct Saving the overhead introduced by Delayed Saving with the two best configurations is lower than 7%.

The sampling frequency has not a major impact on the performance of the monitor, although it is possible to notice that a 20ms sampling rate introduces a higher overhead compared to the other two configurations, due to an excessively fine grained sampling of the CPU consumption. In practice, the ability to quickly react once the applications become inactive caused an observable slow down during the working phase. The results obtained for the 200ms and 1s sampling rates are on the contrary extremely effective and comparable across all configurations and subject applications.

In a nutshell, we can conclude that Delayed Saving can reduce the overhead introduced by Direct Saving by one order of magnitude depending on the application and configuration, with a recommended sampling rate between 200ms and 1s.

Figures 8 and 7 present the results about memory consumption. The collected data show that in both applications and in all configurations there is no significant difference in memory consumption between the idle and working states. This is quite obvious, since applications keep consuming the same amount of memory even if no operation is executed.

Naturally, monitoring always implies consuming additional memory. However, Direct Saving requires less memory than Delayed Saving since it does not need to temporarily store data in memory as Delayed Saving does. Indeed, Direct Saving consumes about 0.1GB in Eclipse and between 0.2GB and 0.5GB in Open Hospital. In contrast, Delayed Saving requires between 0.3GB and 0.4GB in Eclipse and between 0.3GB and 1GB in Open Hospital.

Overall, Delayed Saving is more expensive than Direct Saving, but the total amount of memory consumed can be acceptable for modern execution environments, as long as the monitored software is not a data-intensive application. Delayed Saving thus offers the opportunity of seamlessly collecting data in the field, as long as enough memory can be allocated to the buffering mechanisms implemented in Delayed Saving.

B. RQ2: IS THE DATA COLLECTED BY DELAYED SAVING ACCURATE?

Figures 9 and 10 show the accuracy of the data collected for Eclipse and Open Hospital for all the considered configurations.

Considering the results obtained with Eclipse, we can notice that Delayed Saving performs extremely well with the *Objects Recording* level. Indeed, more than 95% of the data

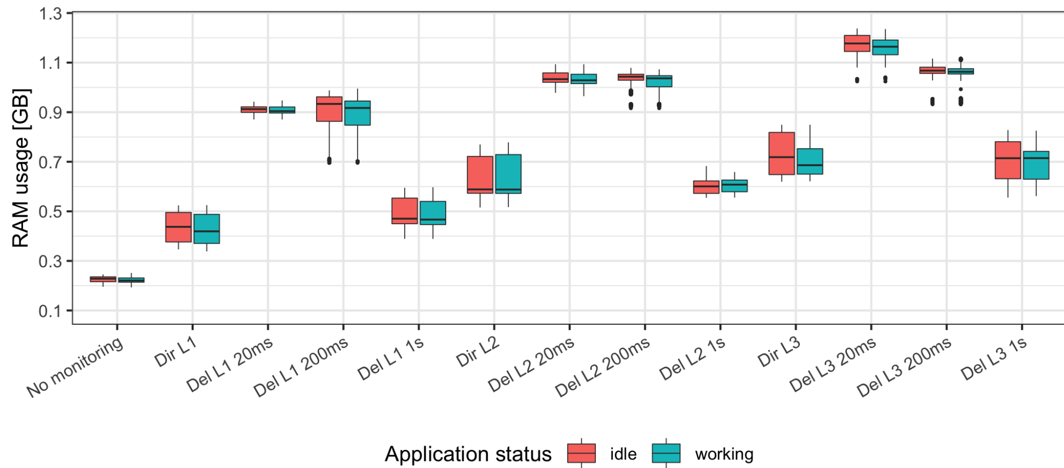


FIGURE 7. Open Hospital: Memory consumption results for all levels.

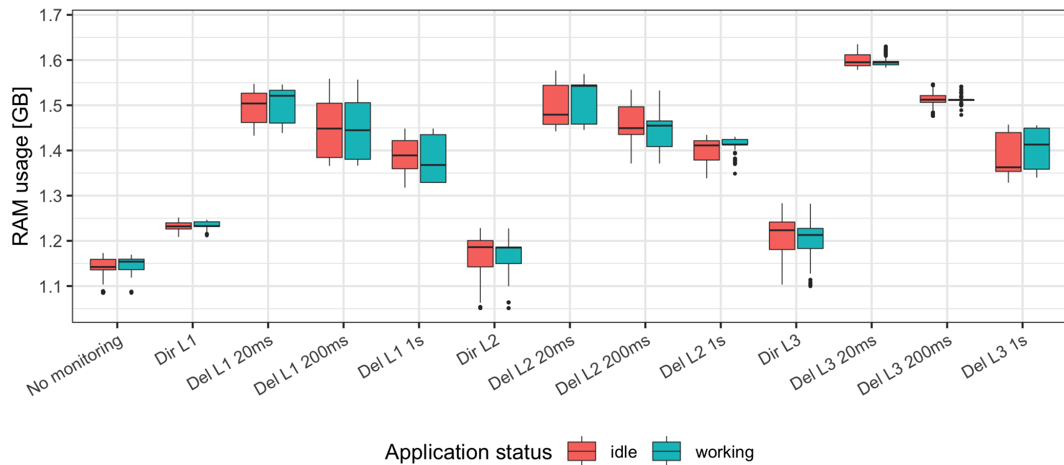


FIGURE 8. Eclipse: Memory consumption results for all levels.

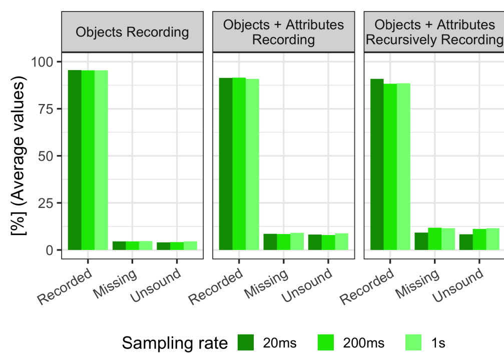


FIGURE 9. Eclipse: Quality comparison for all levels.

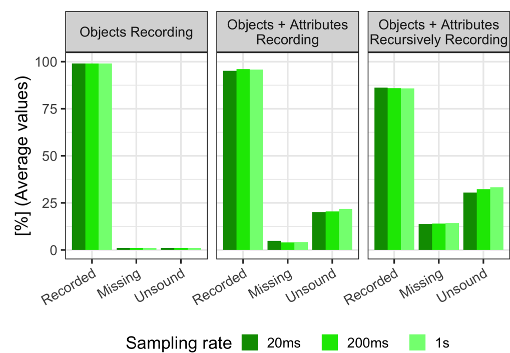


FIGURE 10. Open Hospital: Quality comparison for all levels.

has been recorded accurately (Recorded), and less than 5% of the data is missing (Missing) from the traces. Also, Unsound values are very limited. The accuracy of the Delayed Saving slightly decreases with the *Objects + Attributes Recording* and *Objects + Attributes Recursively Recording* levels: the correctly recorded values (Recorded) amount, in both cases,

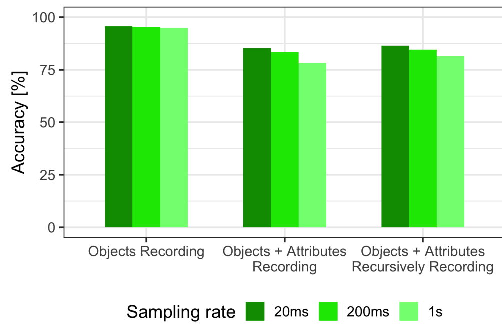


FIGURE 11. Eclipse: Accuracy comparison for all levels.

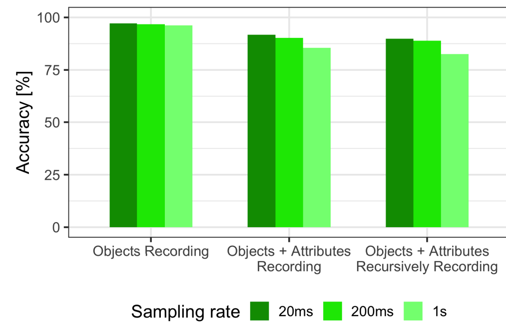


FIGURE 12. Open Hospital: Accuracy comparison for all levels.

to about 90% of the saved data, with a 10% of values either missing (Missing) or replaced with Unsound values. Although collecting more data may challenge the accuracy of the results, still the large majority of the Recorded data is accurate, and this still enables the application of different techniques, such as machine learning techniques (as studied in the next research question).

Open Hospital shows similar results, with a small decrease of the accuracy of the Recorded data while the amount of saved data increases. Coherently, the amount of Missing values remains small across configurations. With respect to Eclipse, in particular when considering the deepest level of data collection, the Unsound data increases. This might be due to the structure of the collected objects, which exploits object references more than Eclipse, and consequently are more exposed to changes, especially when multiple levels of references are navigated to save data into files.

For both Eclipse and Open Hospital, the choice of the sampling rate has a marginal impact on the accuracy of the collected data, with a lower rate producing slightly better results compared to a high rate, since they imply start recording the date earlier when the monitored application enters the idle state. Combining this evidence with the impact of the sampling rate on the performance, a choice of 200ms seems to well balance the performance of the technique and the accuracy of the data.

In a nutshell, Delayed Saving achieved high accuracy with the *Objects Recording* level, and good accuracy with higher recording levels. The sampling rate demonstrated to have a marginal impact on the accuracy of the data, with 200ms representing a good trade-off between performance and accuracy.

C. RQ3: CAN DAIKON GENERATE ACCURATE MODELS FROM THE DATA COLLECTED BY DELAYED SAVING?

Figures 11 and 12 show the accuracy of the models produced by Delayed Saving in comparison with those produced by Direct Saving for Eclipse and Open Hospital for all the used configurations.

Results for Eclipse show that the Daikon pre-conditions mined from the traces collected by Delayed Saving are in average 95% accurate with respect to the pre-conditions generated from traces collected by Direct Saving, when considering the *Objects Recording* level. The accuracy of the pre-conditions decreases to 80% for the *Objects + Attributes Recording* and *Objects + Attributes Recursively Recording* levels, which is still a good result considering that the traces obtained for these two levels are 2.4 times larger than the traces obtained with the *Objects Recording* level.

For Open Hospital, the accuracy of the pre-conditions obtained with Delayed Saving with respect to Direct Saving is 96% for the *Objects Recording* level. Similarly to Eclipse, the accuracy decreases to 88% for the other two levels.

During the experimentation with Eclipse and Open Hospital, we observed that the sampling rate had a little impact on the accuracy of the produced models, with smaller sampling rates producing better results.

In a nutshell, in line with the results obtained for RQ2 and RQ3, a sampling rate of 200ms offers a good balance between performance and accuracy of the data.

D. DISCUSSION

The empirical investigation shows clear complementarity between Direct Saving and Delayed Saving. When applicable, Direct Saving should be preferred, since it produces perfectly accurate data. However, it may introduce significant overhead already when a limited amount of information is recorded (we observed a 25% slow down for the *Objects Recording* level in our evaluation), and this overhead quickly increases when more data is collected (up to 100%-200% slow down in our evaluation for the *Objects + Attributes Recursively Recording* level). This is indeed hardly acceptable in the majority of the settings.

Delayed Saving provides a different trade-off. It reduces by one order of magnitude the overhead, keeping it below 9% for the 200ms and 1s sampling rate, even when recording a large amount of data (e.g., we observed a 99.5% and a 229.39% slow down for the *Objects + Attributes Recursively Recording* level). This capability is traded for some additional memory consumed and a degree of inaccuracy introduced in the recorded data. The additional memory consumption ranged between 0.25GB and 0.9GB in our evaluation, and

it is likely acceptable in the many settings where memory-intensive applications are not executed. The data inaccuracy is limited, as reported empirically. Although the presence of inaccuracy is not acceptable by all the techniques, it is indeed tolerable by several others, such as user profiling [2], failure reproduction [16], and mining techniques [17], [18]. Our investigation based on the Daikon mining technique confirmed a limited impact of data inaccuracy in the mined properties.

E. THREATS TO VALIDITY

The main threats to internal validity concern with the measurement of CPU consumption values and overhead. We mitigated this issue by using OSHI [28], which allows to precisely capture CPU consumption values, and repeating each execution several times to reduce the impact of noise on our results.

Another threat concerns with the instrumentation of the program to collect ground truth data for the calibration procedure. This was mainly manual effort. To mitigate any risk to incorrectly collect data, we tested the instrumented application multiple times before using the data for the experiment.

The main threat to external validity concerns with the generalization of our findings. Although we cannot claim that our results are valid for any software system, the usage of two different end-user applications and the similarity in the results, suggest that the reported empirical evidence can be also valid for many other interactive applications that run on client machines.

VII. RELATED WORK

In this section, we frame our work in the context of related areas. We discuss how it relates to existing studies on efficient field monitoring techniques, and how it relates to other works about adaptive monitoring techniques.

Efficient field monitoring techniques implement different strategies to limit the impact of monitoring on the monitored system. Current approaches dealing with efficient field monitoring can be classified in two main groups: *distributive monitoring* and *probabilistic monitoring*.

Distributive monitoring assigns different monitoring tasks to multiple instances of the same application running on different client machines. This strategy decreases monitoring overhead because it limits the amount of information collected for each location. This strategy has been mainly developed by Bowring *et al.* [11] and Orso *et al.* [31] through the *Gamma System*, which divides a monitoring task into a set of subtasks and assigns them to individual instances of the software to be monitored, in order to minimize runtime overhead. The data gathered from the several instances are however independent from the other locations. This may make the task of reconstructing the behavior of the whole application extremely hard if data are not fully independent. For instance, it is hard to collect and combine segments of executions collected for different application instances under different situations. Delayed Saving is not affected by this challenge since data is still collected from a same instance.

Briola *et al.* [32], [33] and Ancona *et al.* [34] developed a framework for distributed runtime verification of protocols for Multi-Agent Systems (MAS). Basically, the approach splits the global interaction protocol into sub protocols and then monitors the sub protocols by transforming agent messages into Prolog predicates suitable for runtime verification. In the same line, Ferrando *et al.* [35] proposed to distribute and simplify the load of monitoring by partitioning the agents in several groups with one monitor per partition. The partitioning system guarantees that distributed monitoring detects all the protocol violations such as a centralized monitoring system would. Differently from Delayed Saving, runtime protocol verification aims at checking properties at runtime, while Delayed Saving aims at collecting large amount of behavioral data that can be later used to support several tasks.

Probabilistic monitoring accounts for lowering the impact of monitoring by collecting runtime information within a certain probability. Liblit *et al.* [36] have exploited this strategy to isolate bugs by profiling a large, distributed user community and using logistic regression to find the important program predicates that could be faulty. Similarly, Jin *et al.* [13] presented a monitoring framework called Cooperative Crug (Concurrency Bug) Isolation to diagnose production run failures caused by concurrency bugs. This technique uses sampling to monitor different types of predicates while keeping the monitoring overhead low. In the same way, Hirzel and Chilimbi [14] developed Bursty Monitors, which collect subsequences of events with ad-hoc strategies to construct a temporal program profile. In general, Probabilistic Monitoring is designed to collect partial information about the execution, that is, the monitor is not always active and the traces might be incomplete. On the contrary, Delayed Saving aims at collecting traces that are complete, although they may contain some inaccuracies.

Since monitoring requirements might change over time, it is useful to implement *Adaptive monitoring techniques*, that is, techniques with a monitoring logic that can adapt dynamically to runtime events. Recently, Zavala *et al.* [37] worked on a systematic mapping of adaptive monitoring techniques to identify trends and approaches available in the literature. Bartocci *et al.* [12] presented Adaptive Runtime Verification, a monitoring technique that controls the overhead by enabling and disabling runtime verification of events according to certain overhead target levels. This framework determines statistically the probability that an application property is violated, and based on this number a higher or a lower level of overhead for that property is assigned. When is not possible to check a certain property (e.g., due to a high overhead level) the framework estimates the probability that a property will be satisfied, instead of monitoring the actual property, and thus reducing the load of runtime verification.

Maurel *et al.* [38] defined an adaptive monitoring approach for smart-home systems, where the monitoring system can be dynamically activated and tuned based on the current

CPU usage: the tuning process consists of automatically selecting the amount of behaviors to be monitored at runtime.

Adaptation in adaptive monitoring techniques can be triggered not only by the amount of resources available but also by human feedback. For instance, Nainar and Liblit [39] and Whittle *et al.* [40] defined approaches that use human feedback to adapt the system's instrumentation and the behavior of the monitor. In the domain of service-based systems, Contreras and Zisman [41] studied how to adapt the monitor to the characteristics of the user context.

In real-time systems, timing requirements such as response time are essential and violations to these requirements are not accepted. Moghadam *et al.* [42] proposed a self-adaptive monitor technique that uses reinforcement learning to adapt its behavior based on requirements about the response time. Unlike these approaches, Delayed Saving focuses on automatically adapting to the running environment so that the introduced overhead cannot significantly affect users.

Finally, the idea of exploiting the unused resources to perform additional tasks has been already investigated in other contexts, especially to build distributed volunteer computing platforms. For instance, the project Folding@Home exploits users' unused resources to perform computations useful to determine a cure to diseases [43], while SETI@home analyzes radio signals looking for evidence of extraterrestrial life [44]. Differently from these platforms, Delayed Saving originally exploits the idle time to opportunistically delay data recording, offering a new trade-off between overhead and data accuracy.

VIII. CONCLUSIONS

Field monitoring techniques can be extremely useful to extract data about the behavior of software applications when interacting with real users and processing real data. However, collecting significant amount of data is in conflict with the need of introducing a small overhead, so that users are not annoyed by the presence of monitoring features.

This paper presents Delayed Saving, a novel approach that offers a complementary opportunity to field monitoring, compared to approaches that collect partial data or distribute monitoring features. The idea developed in Delayed Saving is to significantly reduce the monitoring overhead by postponing the time when values are saved in a persistent memory to the idle phase of the monitored application. In this way, only inexpensive in-memory value tracing is performed while the monitored application is active, reducing overhead by orders of magnitudes compared to directly saving values, as demonstrated in our evaluation. On the other hand, Delayed Saving may produce inaccurate data. Thus monitoring techniques exploiting the Delayed Saving principle must be able to tolerate it.

As part of future work, we are currently investigating how to further elaborate Delayed Saving and manage to discriminate between the data that must be immediately saved and the one that can be saved during the idle phase, to reduce or even eliminate data inaccuracy and make the approach applicable also to those techniques that require perfectly accurate data extracted from the field.

REFERENCES

[1] M. El-Ramly, E. Stroulia, and P. Sorenson, "From run-time behavior to usage scenarios: An interaction-pattern mining approach," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2002, pp. 315–324.

[2] S. Elbaum and M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 312–327, Apr. 2005.

[3] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in Android testing: They matter," in *Proc. Int. Workshop Adv. Mobile App Anal. (A-Mobile)*, 2018, pp. 1–6.

[4] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit, "Optimizing customized program coverage," in *Proc. Int. Conf. Automated Softw. Eng. (ASE)*, 2016, pp. 27–38.

[5] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, May 1999, pp. 277–284.

[6] L. Gazzola, "Field testing of software applications," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE)*, May 2017, pp. 429–432.

[7] L. Gazzola, L. Mariani, F. Pastore, and M. Pezzè, "An exploratory study of field failures," in *Proc. Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2017, pp. 67–77.

[8] S. Sharma, *The DevOps Adoption Playbook: A Guide to Adopting DevOps in a Multi-Speed IT Enterprise*. Hoboken, NJ, USA: Wiley, 2017.

[9] T. Schlossnagle, "Monitoring in a DevOps world," *Commun. ACM*, vol. 61, no. 3, pp. 58–61, Mar. 2018.

[10] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res. (FoSER)*, 2010, pp. 17–22.

[11] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2002, pp. 2–9.

[12] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster, "Adaptive runtime verification," in *Proc. Int. Conf. Runtime Verification (RV)*, 2012, pp. 168–182.

[13] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 241–255, 2010.

[14] M. Hirzel and T. Chilimbi, "Bursty tracing: A framework for low-overhead temporal profiling," in *Proc. ACM Workshop Feedback-Directed Dyn. Optim. (FDDO)*, 2001, pp. 117–126.

[15] O. Cornejo, D. Briola, D. Micucci, and L. Mariani, "In the field monitoring of interactive application," in *Proc. IEEE/ACM Int. Conf. Softw. Eng., New Ideas Emerg. Technol. Results Track (ICSE-NIER)*, May 2017, pp. 55–58.

[16] W. Jin and A. Orso, "BugRedux: Reproducing field failures for in-house debugging," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 474–484.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[18] L. Mariani, M. Pezzè, and M. Santoro, "GK-tail+ an efficient approach to learn software models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 8, pp. 715–738, Aug. 2017.

[19] Eclipse. (2019). *Eclipse Community*. [Online]. Available: http://www.eclipse.org

[20] (2019). *Open Hospital*. [Online]. Available: https://www.open-hospital.org/en/

[21] A. Gorji and M. Abadi, "Detecting obfuscated javascript malware using sequences of internal function calls," in *Proc. ACM Southeast Regional Conf. (SE)*, 2014, p. 64.

[22] S. S. Murtaza, A. Hamou-Lhadj, N. H. Madhavji, and M. Gittens, "Towards an emerging theory for the diagnosis of faulty functions in function-call traces," in *Proc. SEMAT Workshop Gen. Theory Softw. Eng. (GTSE)*, 2015, pp. 59–68.

[23] Z. Zhao, B. Wu, M. Zhou, Y. Ding, J. Sun, X. Shen, and Y. Wu, "Call sequence prediction through probabilistic calling automata," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2014, pp. 745–762.

[24] L. Mariani, A. Marchetto, C. D. Nguyen, P. Tonella, and A. I. Baars, "Revolution: Automatic evolution of mined specifications," in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2012, pp. 241–250.

[25] F. Pastore, D. Micucci, and L. Mariani, "Timed k-tail: Automatic inference of timed automata," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 401–411.
[26] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Boston, MA, USA: Newnes, 2013.
[27] (2019). *JNativeHook*. [Online]. Available: https://github.com/kwhat/jnativehook
[28] (2019). *OSHI*. [Online]. Available: https://github.com/oshi/oshi
[29] Eclipse Foundation. (2019). *Aspectj*. [Online]. Available: https://www.eclipse.org/aspectj/
[30] R. Hocke. (2019). *Sikulix Capture and Replay Tool*. [Online]. Available: http://sikulix.com
[31] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: Continuous evolution of software after deployment," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2002, pp. 65–69.
[32] D. Briola, V. Mascardi, and D. Ancona, "Distributed runtime verification of JADE multiagent systems," in *Proc. Intell. Distrib. Comput. (IDC)*, 2015, pp. 81–91.
[33] D. Briola, V. Mascardi, and D. Ancona, "Distributed runtime verification of JADE and Jason multiagent systems with prolog," in *Proc. Italian Conf. Comput. Logic (CILC)*, 2014, pp. 319–323.
[34] D. Ancona, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert, "Efficient verification of MASs with projections," in *Engineering Multi-Agent Systems* (Lecture Notes in Computer Science), vol. 8758. Cham, Switzerland: Springer, 2014, pp. 246–270.
[35] A. Ferrando, D. Ancona, and V. Mascardi, "Decentralizing MAS monitoring with DecAMon," in *Proc. 16th Conf. Auton. Agents Multiagent Syst. (AAMAS)*, 2017, pp. 239–248.
[36] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 141–154, 2003.
[37] E. Zavala, X. Franch, and J. Marco, "Adaptive monitoring: A systematic mapping," *Inf. Softw. Technol.*, vol. 105, pp. 161–189, Jan. 2019.
[38] Y. Maurel, A. Bottaro, R. Kopetz, and K. Attouchi, "Adaptive monitoring of end-user OSGI-based home boxes," in *Proc. 15th ACM SIGSOFT Symp. Compon. Based Softw. Eng. (CBSE)*, 2012, pp. 157–166.
[39] P. A. Nainar and B. Liblit, "Adaptive bug isolation," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng. (ICSE)*, May 2010, pp. 255–264.
[40] J. Whittle, W. Simm, and M.-A. Ferrario, "On the role of the user in monitoring the environment in self-adaptive systems: A position paper," in *Proc. ICSE Workshop Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, 2010, pp. 69–74.
[41] R. Contreras and A. Zisman, "A pattern-based approach for monitor adaptation," in *Proc. IEEE Int. Conf. Softw. Sci., Technol. Eng. (ICSTE)*, Jun. 2010, pp. 30–37.
[42] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "Adaptive runtime response time control in PLC-based real-time systems using reinforcement learning," in *Proc. IEEE/ACM 13th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, May/Jun. 2018, pp. 217–223.
[43] Pande Lab, Find Home. (2019). [Online]. Available: https://foldingathome.org/
[44] Seti Home. (2019). [Online]. Available: https://setiathome.berkeley.edu/

OSCAR CORNEJO received the M.Sc. degree in computer science engineering from the Federico Santa María Technical University, in 2014, and the Ph.D. degree in computer science from the University of Milano-Bicocca, in 2019, where he is currently a Postdoctoral Researcher with the LTA Group.

He has been working mainly on software engineering, studying how to develop non-intrusive monitoring techniques, always in the sight of supporting software testing and analysis processes.

DAVIDE GINELLI received the M.Sc. degree in computer science from the University of Milano-Bicocca, in 2017, where he is currently pursuing the Ph.D. degree in computer science.

His research interests include software engineering, in particular self-healing and self-repairing systems, real-time monitoring of interactive applications without interfering with the user experience, and software architectures. He is also interested in the mobile and web development.

DANIELA BRIOLA received the Ph.D. degree in computer science from the University of Genoa, in 2009. She is currently an Assistant Professor with the University of Milano-Bicocca.

Her research interests include software engineering, with a focus on real-time monitoring of interactive applications, and artificial intelligence, in particular multiagent systems and knowledge representation. She is regularly involved in the program committees of many international conferences in her areas of interest.

DANIELA MICUCCI received the Ph.D. degree in mathematics, statistics, computational sciences and computer science from the University of Milano, in 2004. She is currently an Assistant Professor with the University of Milano-Bicocca.

Her research interest includes software engineering, in particular software architectures, real-time systems, and self-healing and self-repairing systems. She is currently active in several European and National projects. She is also regularly involved in the program committees of workshops and conferences in her areas of interest.

LEONARDO MARIANI received the Ph.D. degree in computer science from the University of Milano-Bicocca, in 2005, where he is currently a Full Professor.

His research interests include software engineering, in particular software testing, program analysis, automated debugging, and self-healing and self-repairing systems. He has authored more than 100 papers appeared at top software engineering conferences and journals.

He has been awarded with the ERC Consolidator Grant, in 2015, and an ERC Proof of Concept Grant, in 2018, and he is currently active in several European and National projects. He is regularly involved in organizing and program committees of major software engineering conferences.