Lightweight Cooperative Logging for Fault Replication in Concurrent Programs

Nuno Machado, Paolo Romano, Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa

***Abstract*—This paper presents CoopREP, a system that provides support for fault replication of concurrent programs, based on cooperative recording and partial log combination. CoopREP employs partial recording to reduce the amount of information that a given program instance is required to store in order to support deterministic replay. This allows to substantially reduce the overhead imposed by the instrumentation of the code, but raises the problem of finding the combination of logs capable of replaying the fault. CoopREP tackles this issue by introducing several innovative statistical analysis techniques aimed at guiding the search of partial logs to be combined and used during the replay phase. CoopREP has been evaluated using both standard benchmarks for multi-threaded applications and a real-world application. The results highlight that CoopREP can successfully replay concurrency bugs involving tens of thousands of memory accesses, reducing logging overhead with respect to state of the art non-cooperative logging schemes by up to 50 times in computationally intensive applications.**

***Keywords*-concurrency errors; deterministic replay; debugging; performance**

I. Introduction

Software bugs continue to hamper the reliability of software. It is estimated that bugs account for 40% of system failures [1]. Unfortunately, despite the progress made on the development of techniques that prevent and correct errors during software production (e.g. formal methods [2]), a significant number of errors still reaches production [3]. This problem is exacerbated by the advent of multi-core systems and the increasing complexity of modern software. Therefore, it is imperative to develop tools that simplify the task of debugging the software, for instance, by providing the means to replay a faulty execution.

A fundamental challenge is that achieving deterministic replay is far from trivial, especially in parallel applications. Contrary to most bugs in sequential program, that usually depend exclusively on the program input and on the execution environment (and therefore can be more easily reproduced), concurrency bugs have an inherently non-deterministic nature. This means that even when re-executing the same code with identical inputs, on the same machine, the program outcome may be differ from run to run [4].

The *deterministic replay* technique addresses this problem by recording the relevant details of the execution [5] (including the order of access to shared memory regions, thread scheduling, program inputs, signals, etc) to support the reproduction of the original run. However, logging all the required information induces a large space and performance overhead during production runs.

In the past decade, a significant amount of research has been performed on techniques to provide deterministic replay (either based on hardware or software). Several of these solutions [6], [7], [8], [9] aim at replaying the bug on the first attempt, but this comes at an excessively high cost on the original run (10x-100x slowdown), making the approach impractical in most settings.

Since the most significant performance constraints are on the production version, it becomes of paramount importance to reduce its instrumentation overhead, even if it results in a slightly longer reproduction time during diagnosis.

In this paper we introduce and evaluate the idea of exploiting the coexistence of multiple instances of the same program to devise cooperative logging schemes. The underlying intuition is very simple: sharing the burden of logging among multiple instances of the same (buggy) program, by having each instance track accesses that only target a subset of the program's shared variables. The partial logs recorded by different instances of the same program are then gathered at the software maintenance side, where they are statistically analyzed in order to identify sets of partial logs whose combination maximizes the chances to successfully replay the bug. We have named the resulting system CoopREP (standing for Cooperative Replay), a deterministic replay system that leverages on cooperative logging performed by multiple clients and on statistical techniques to combine the collected partial logs. One of the main contributions of this paper is to show that *cooperative logging is a viable strategy to replicate concurrency bugs with low overhead*. Additionally, the paper also makes the following contributions:

- A set of novel statistical metrics to detect correlations among partial logs, that have been independently collected by different clients;
- A novel heuristic, named Similarity-Guided Merge, that leverages on these metrics to systematically perform a guided search, among the possible combinations of partial logs. The goal is to find those that generate complete replay drivers capable of reproducing the bug.
- An experimental evaluation of the implemented prototype of CoopREP, based on standard benchmarks for multi-threaded applications and on a real-world application.

The rest of this document is structured as follows: Section II presents the background concepts related to this work.

Section III overviews some deterministic replay and statistical debugging systems. Section IV introduces CoopREP, describing in detail its architecture, the Similarity-Guided Merge heuristic, and the metrics used to capture the similarity among partial logs. Section V presents the results from the experimental evaluation. Finally, Section VI concludes the paper by summarizing its main points and discussing future work.

II. Background

A. Deterministic replay

Deterministic replay (or record/replay) aims to overcome the problems associated with the reproduction of bugs, in particular those raised by non-determinism. The purpose of this technique is to re-execute the program, obtaining the exact same behavior as the original execution. This is possible because almost all instructions and states can be reproduced as long as all possible non-deterministic factors that have an impact on the program execution are replayed in the same way [4]. Thereby, deterministic replay operates in two phases:

1) Record phase – consists of capturing data regarding non-deterministic events, putting that information into a trace file.

2) Replay phase – the application is re-executed consulting the trace file to force the replay of non-deterministic events according to the original execution.

B. Sources of Non-determinism

External factors often interfere with the program execution, preventing the timing and the sequence of instructions executed to be always identical. The sources of these factors can be divided into two types: *input non-determinism* and *memory non-determinism* [10].

Input non-determinism encompasses all the inputs that are received by the system layer being logged but are not originated in that layer (e.g. signals, system calls, hardware interrupts, DMA, keyboard and network inputs, etc). This kind of non-determinism is present in both single-processor and multi-processor machines.

Memory non-determinism in single-processor systems is mainly due to the thread interleaving in the access to shared memory locations, which may vary from run to run (and, with a lesser extent, from reads to un-initialized memory locations). Memory non-determinism can be tackled by using "logic time" [11] to log the events, instead of ordinary physical time. In fact, logical time may be sufficient to support deterministic replay in single-processor systems [12]. However, in multi-processor systems (e.g. SMPs and multi-cores) the scenario is more complex since it is necessary to take into account how threads that execute concurrently on different processors may interleave with each other. Therefore, one needs to capture the global order of shared memory accesses and synchronization points (obviously, this is not a problem when threads are independent from each other).

III. Related Work

There are various approaches to prevent bugs in a program or to optimize the debugging process. In this section we focus on approaches that aim at reproducing the failure or to statistically isolate it, as these are the most relevant to our work. Among the deterministic replay solutions, over the past few years, several solutions have been proposed to cope with the challenges brought by multi-processors. Based on how they are implemented, prior work can be divided in two main categories: *hardware-based* and *software-based*.

Hardware-based solutions rely on hardware extensions to efficiently record the non-deterministic events and, consequently, mitigate the overhead imposed to the production run. Flight Data Recorder [13] and BugNet [14] have provided solutions to achieve this, but at a cost of non-trivial hardware extensions. More recently, DeLorean [15] and Capo [16] have proposed new techniques to reduce the complexity of the extensions. Despite that, they still require changing non-commodity hardware, which can be costly.

Regarding software-based approaches, InstantReplay [8] was the first deterministic replay system to support multi-processors. It leverages on an instrumented version of the *Concurrent-Read Exclusive-Write* (*CREW*) protocol [8] to control and log the accesses to shared memory locations. In turn, DejaVu [7] logs the order of thread "critical events" (e.g. synchronization points and accesses to shared variables) and uses global clocks to enforce their execution in total order at replay time. However, this technique incurs high performance overhead and requires large trace files. JaRec [6] reduces the overheads imposed by InstantReplay and DejaVu, by dropping the idea of global ordering and using a *Lamport's clock* [11] to preserve the partial order of threads with respect to synchronization points. However, JaRec requires a program to be data race free in order to guarantee a correct replay, otherwise it only ensures deterministic replay until the first data race. This constraint makes this approach unsuitable for most real world concurrent applications, given that is common the existence of both benign and harmful data races. LEAP [9] addresses JaRec issues by tracking all shared memory accesses, in addition to those performed on monitors. However, to minimize the runtime overheads, LEAP only keeps a local trace for each shared variable, containing the order of the thread accesses to that variable.

All the previous approaches try to reproduce the bug on the first replay run, thus inducing large overheads during production runs, with the drawback of penalizing also bug-free executions, which are much more frequent than the faulty ones [4]. Motivated by this, some recent solutions, such as PRES [4], ODR [17], and ESD [18], relax the constraint of replaying the bug at the first attempt, by only logging partial

information (or even none, in the case of ESD) in order to further minimize the cost of recording the original execution. Later, these solutions apply inference techniques to complete the missing information.

Our solution, denoted CoopREP, is also based on the observation that it is not crucial to achieve deterministic replay at the first attempt, but improves previous work as it leverages on information logged by multiple clients to ease the inference task. For this, CoopREP draws on statistical debugging techniques, which aim at isolating bugs by analyzing information gathered from a large number of users.

Statistical debugging was pioneered by CBI [19]. This system collects feedback reports that contain values recorded for certain predicates of the program (e.g. branches, return values, etc). Then, performs a statistical analysis of the information gathered in order to pinpoint the likely source of the failure. However, CBI does not support concurrency bugs. CCI [20] outstrips this limitation by adjusting CBI's principles to cope with non-deterministic events. For instance, it implements cross-thread sampling and relies on longer sampling periods, because concurrency bugs always involve multiple memory accesses. CoopREP differs from CCI in the sense that we are concerned with the bug reproduction, whereas CCI strives to identify and isolate predictors that can explain the root causes of failures caused by concurrency errors.

Since recording and replaying input non-determinism can be achieved with an overhead less than 10% [4], [5], [12], we only focus on coping with memory non-determinism. In fact, a recent study on the evolution of the types of errors in MySQL database [21] shows a growth trend in the number and proportion of concurrency bugs over the years. Thereby, CoopREP addresses the deterministic replay of this kind of bugs (e.g. atomicity violations, data races), disregarding other sources of non-determinism.

IV. CoopREP System

This section describes CoopREP, a system that provides fault replication of concurrent programs, based in cooperative recording and partial log combination. Given that CoopREP reuses and extends several building blocks that were originally introduced in LEAP [9], we begin by presenting an overview of LEAP.

A. LEAP

LEAP [9] proposes a general technique for the deterministic replay of Java concurrent programs in multi-processors. It is based on the insight that, to achieve deterministic replay, it is sufficient to record the local order of thread accesses to shared variables, instead of enforcing a global order. To track thread accesses, LEAP associates an *access vector* to each different shared variable. During execution, whenever a thread reads or writes in a shared variable, its ID is stored in the access vector. For instance, let us assume a program P with a shared variable x and running with two threads (t_1 and t_2). If, during the execution of P, x is accessed one time by t_1 and, later, two times by t_2, the access vector of x will be `<t1,t2,t2>`.

Using this technique, one gets (local) order vectors of thread accesses performed on individual shared variables, instead of a global-order vector. This provides lightweight recording, but relaxes faithfulness in the replay, allowing thread interleavings that are different from the original execution. However, in [9] the authors claim that this approach does not affect the error reproduction, and they formally prove the soundness of this statement.

To locate the shared program elements (SPEs), LEAP uses a static escape analysis called *thread-local objects analysis* [22] from the Soot[1] framework. Given that accurately identifying shared variables is generally an undecidable problem, this technique computes a sound over-approximation, i.e. every shared access to a field is indeed identified, but some non-shared fields may also be classified as shared [9].

SPEs include variables that serve as monitors (including Java monitors) and other shared field variables (including class and thread escaped instance variables). For each identified SPE, LEAP assigns offline a numerical index in order to be able to consistently identify objects across different runs. Moreover, as access vectors only contain thread IDs tracked during the production run, it is imperative to correctly recognize each thread in both recording and replay phases. LEAP achieves this by maintaining a mapping between the thread name and the thread ID during recording and using the same mapping for replay. In addition, it uses a list with the parent threads' IDs to track the global order in which they create their child threads, thus ensuring the same thread creation order when recording and replaying the execution.

The overall infrastructure of LEAP, depicted in Figure 1, consists of three major components: the *transformer*, the *recorder*, and the *replayer*.

The transformer receives the Java program bytecode and employs two types of instrumentation schemes to produce the *record version* and the *replay version*, respectively. LEAP instruments the following code instructions: *i)* SPE accesses, *ii)* thread creation information, and *iii)* recording end points.

The record version is then executed and when a program end point is reached, the recorder saves both the recorded access vectors and the thread ID map information. In addition, the recorder also creates the replay driver, i.e. a Java file containing the code needed to execute the replay version of the program and initiate both the thread scheduler and the trace loader components.

Finally, the replayer uses the logged information and the

[1] http://www.sable.mcgill.ca/soot

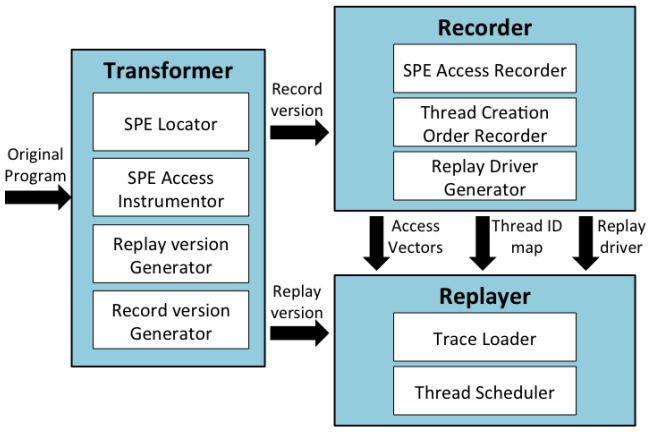

Figure 1. Overview of the LEAP architecture (adapted from [9]).

generated replay driver to replay the program. To control the interleaving of thread execution (and enforce a deterministic replay), LEAP takes control of the thread scheduling and consults the thread ID map information file.

The evaluation study presented in [9] has shown that LEAP incurs a runtime overhead ranging from 7.3% to 626% (for applications with several shared variables accessed in hot loops). In terms of space overhead, the log size in LEAP is still considerable, ranging from 51 to 37760 KB/sec.

As we will explain, CoopREP re-uses some key concepts of LEAP, in particular the idea of logging accesses on a per-SPE basis. However, by introducing the notion of cooperative logging, CoopREP allows for achieving significant (up to one order of magnitude) reductions of the logging overhead incurred by LEAP and, more in general, by classic non-cooperative logging schemes.

B. CoopREP Architecture

Figure 2 illustrates the overall architecture of CoopREP. During the instrumentation phase (Figure 2-1), the *transformer* instruments the Java program bytecode to generate both the record version and the replay version, as done in LEAP. Unlike LEAP, however, the record version is instrumented to only log a subset of the SPEs. In the following, we will refer to this version as *partial record version*. The partial record versions are then sent to the clients, whereas the replay version is sent to the replayer.

Figure 2-2 illustrates the record and replay phases. In CoopREP, there is a *recorder* module for each client. Conversely to LEAP, in CoopREP the recorder does not record all SPEs' access vectors. Instead, each user logs accesses only to a part of the program's SPEs, as previously defined by the transformer. Assuming that the program is executed by a large population of users, this mechanism allows to gather access vectors for the whole set of SPEs with high probability. By doing this, CoopREP aims at minimizing the

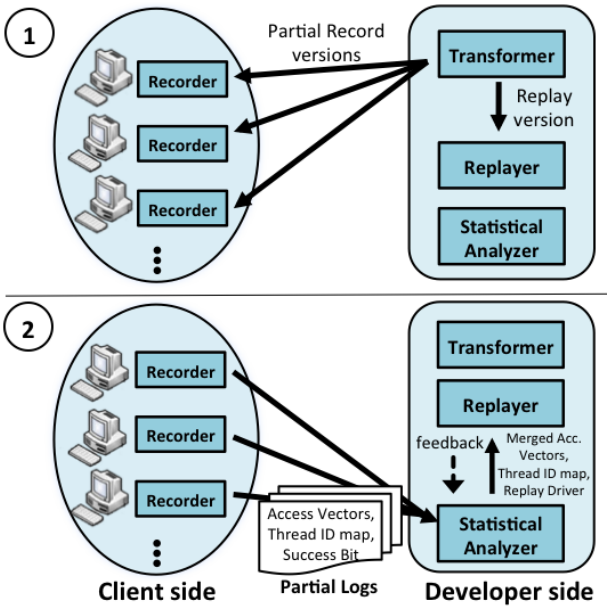

Figure 2. Overview of the CoopREP architecture: (1) Instrumentation phase; (2) Record and Replay phases.

performance overhead that would be required if one had to record all the access vectors at each client.

When the production run ends, each client sends its *partial log* to the developer site to be analyzed. This log consists of the access vectors recorded for a subset of the SPEs, a hash of the log (computed in our prototype using MD5), the thread ID map, and also an additional bit indicating the success or failure of the execution (successful executions can be useful for the statistical analysis).

At the developer site, the *statistical analyzer* employs a novel lightweight statistical methodology (see Section IV-G) that aims at pinpointing which partial logs are more likely to be successfully recombined in order to generate a complete log yielding an equivalent execution that reproduces a given concurrency bug.

Finally, the combination of access vectors determined by the *statistical analyzer* is passed as input to the *replayer* component, along with the thread ID map and the generated replay driver. Note that, given that access vectors come from independent executions, the resulting combined log can be incompatible, meaning that the *replayer* fails to enforce the thread execution order specified by the access vectors of the combined log. In this case, the execution will hang, as none of the threads will be allowed to perform its subsequent access on a SPE. This allows to use a simple, but effective, deadlock detection mechanism that terminates immediately the execution replay as soon as it is detected that all threads are prevented from progressing.

Of course, the replayer needs also to deal with the case in which the bug is not replayed. This is a non-trivial issue,

for which we rely on standard techniques already adopted in other probabilistic fault-replication schemes [4], [17]. For crash failures, it is straightforward, as CoopREP can catch exceptions. For incorrect results, it is more complicated, and it is, in general, required to obtain information from programmers regarding the failure symptoms. These include conditional breakpoints to examine outputs for detecting anomalies with respect to the originally buggy execution to be replayed. CoopREP could also benefit from the integration with bug detection tools, such as [23], which could automate the identification of non-visible bugs.

In case the bug cannot be successfully reproduced, the replayer will send *feedback* to the statistical analyzer communicating the replay failure, so the latter can investigate another access vector combination and produce a new complete log for replay. This process ends when the bug is successfully replayed or when the maximum number of attempts to do it is reached.

C. Partial Log Recording

CoopREP is based on having each instance recording accesses to only a fraction of the entire set of the SPEs of the program. The subset of SPEs to be traced is defined at instrumentation time by the transformer. For this purpose, different criteria may be used, e.g. random selection, load balancing distribution, subset of fixed SPEs, etc. In this paper all experiments use the random selection of SPEs. The research of the viability of different selection criteria is left for future work.

Therefore, in the implemented prototype, whenever a new SPE is identified, CoopREP uses a simple probabilistic scheme to decide whether the SPE is to be instrumented or not by a given instance. This ensures that each instance only tracks a fraction of the total number of SPEs of the program, and statistical fairness. However, it is not guaranteed that two partial logs (acquired at different clients) necessarily overlap. The drawback of this is that, if two partial logs have no SPEs in common, it is impossible to deduce whether these partial logs were traced from equal executions and that are suitable to be combined. This problem could be addressed by increasing the percentage of coverage to ensure SPE overlapping (at the cost of greater overheads), or by defining a smaller fixed subset of SPEs to be logged by all users.

Moreover, one must note that the overhead reduction may not be linear with the decrease of the coverage percentage. This happens because some SPEs may be accessed more frequently than others, therefore, when instrumenting the code, the logging overhead may not be distributed equally among the users. A solution for this could be instrumenting the whole program and profiling it, at the developer side, in order to measure the number of accesses performed on each SPE. Later, when instrumenting the user versions, CoopREP could already take into account the SPEs access distribution.

D. Merge of Partial Logs

The major challenge of using partial recording is how to combine the collected partial logs in such a way that the access vectors used lead to a feasible thread interleaving, capable of reproducing the bug during the replay.

In general, the following facts make the partial log merging difficult: *i)* the bug can be the result of several different thread interleavings; *ii)* the probability of obtaining two identical executions of the same program can be very low (this probability is inversely proportional to the complexity of the program in terms of number of SPEs and the number of thread accesses); *iii)* the combination of access vectors from partial logs of faulty executions may enforce a thread order that leads to a non-faulty replay execution; *iv)* the combination of access vectors from partial logs of faulty executions may enforce a thread order that leads to a non-compatible replay execution.

To address these challenges, and to mitigate the incompatibility of the merged access vectors, CoopREP applies statistical metrics over the universe of collected partial logs to pick those that present more similarity. Thereby, our statistical metrics are divided in two types: *statistical metrics for partial log correlation* and *statistical metrics for bug correlation*.

E. Statistical Metrics for Partial Log Correlation

These metrics measure the amount of information that different partial logs may have in common, so that one can increase the probability of merging compatible access vectors. In particular, the following statistical metrics are used to calculate the partial log correlation: *Similarity* and *Relevance*. Both metrics are described in detail below.

1) Similarity: The rationale behind the classification of the similarity between two partial logs is related to their number of SPEs with identical access vectors (i.e. that had recorded exactly the same thread interleaving). Hence, the more SPEs with equal access vectors the partial logs have, the better.

The computation of this metric can come in two flavors: *Plain Similarity* (PSIM) and *Dispersion-based Similarity* (DSIM), according to the weight given to the SPEs of the program. To better define these metrics, Table I presents some formal notation. With this notation, we can now define the metrics as follows.

Let l_0 and l_1 be two partial logs, their *Plain Similarity* (PSIM) is given by the following equation:

$$\text{PSIM}(l_0, l_1) = \frac{\#Equal_{l_0,l_1}}{\#\mathcal{S}} \times \left(1 - \frac{\#Diff_{l_0,l_1}}{\#\mathcal{S}}\right) \quad (1)$$

where $\#Equal_{l_0,l_1}$, $\#\mathcal{S}$, and $\#Diff_{l_0,l_1}$ denote the cardinality of the sets $Equal_{l_0,l_1}$, $\mathcal{S}$, and $Diff_{l_0,l_1}$, respectively. Note that since we are using the client-generated hashes of the logs, the functions $Equal_{l_0,l_1}$ and $Diff_{l_0,l_1}$ can be implemented very efficiently.

Notation	Description
$\mathcal{S}$	Set of all the SPE identifiers of the program.
$\mathcal{S}_l$	Set of the SPE identifiers recorded only by the partial log l.
$\mathcal{AV}$	Set of the different hashes of the access vectors recorded by all the partial logs.
$\mathcal{AV}_l$	Set of the hashes of the access vectors recorded only by the partial log l.
$avecs(s) : \mathcal{S} \rightarrow \mathcal{AV}$	Map that, for a given SPE identifier s, returns the set of the hashes of its access vectors across all the partial logs.
$avec_l(s) : \mathcal{S}_l \mapsto \mathcal{AV}_l$	Function that maps a SPE identifier s to the hash of its access vector, recorded by the partial log l.
$\mathit{Equal}_{l_0,l_1} = \{s \mid s \in \mathcal{S}_{l_0} \cap \mathcal{S}_{l_1} \wedge avec_{l_0}(s) = avec_{l_1}(s)\}$	Set of the SPE identifiers, recorded by both partial logs l_0 and l_1, with identical access vectors.
$\mathit{Diff}_{l_0,l_1} = \{s \mid s \in \mathcal{S}_{l_0} \cap \mathcal{S}_{l_1} \wedge avec_{l_0}(s) \neq avec_{l_1}(s)\}$	Set of the SPE identifiers, recorded by both partial logs l_0 and l_1, with different access vectors.
$Sim_{l_0} = \{l_1, l_2, ..., l_k\}$	Set of the k partial logs more similar (according to either the plain or dispersed similarity metric) to l_0 (denoted as *group of similars of* l_0).
$Fill_{l_0,Sim_{l_0}} = \{s \mid s \in \mathcal{S}_{l_0} \cup \mathcal{S}_{l_1} \cup \mathcal{S}_{l_2} \cup ... \cup \mathcal{S}_{l_k} \wedge l_1, l_2, ..., l_k \in Sim_{l_0}\}$	Union of the sets of the SPE identifiers recorded by the partial log l_0 and by the partial logs of its group of similars Sim_{l_0}.

Table I
NOTATION USED TO DEFINE THE STATISTICAL METRICS.

It should also be noted that this metric will only be 1 when both logs are complete and identical, i.e. they have recorded access vectors for all the SPEs of the program ($\mathcal{S}_{l_0} = \mathcal{S}_{l_1} = \mathcal{S}$) and those access vectors are equal for both logs ($avec_{l_0}(s) = avec_{l_1}(s), \forall s \in \mathcal{S}$). This implies that, for every two partial logs, their plain similarity will always be less than 1. However, the greater this value is, the more probable is that the both partial logs are compatible.

Let l_0 and l_1 be two partial logs, their *Dispersion-based Similarity* (DSIM) is given by the following equation:

$$\text{DSIM}(l_0, l_1) = \sum_{x \in \mathit{Equal}_{l_0,l_1}} \mathit{weight}(x) \quad \times \left(1 - \sum_{y \in \mathit{Diff}_{l_0,l_1}} \mathit{weight}(y)\right)$$

where $weight(s)$ is a function of type $\mathcal{S} \rightarrow Double$ that maps each SPE identifier to a double value that captures its relative weight, in terms of *overall-dispersion*. Here, the overall-dispersion of a given SPE corresponds to the ratio between the number of *different* access vectors (based on their hash) logged (across all clients) for that SPE and the total number of *different* access vectors collected for all the SPEs (across all clients). Thereby, the weight function of a SPE identifier s can be calculated as follows:

$$weight(s) = \frac{\#avecs(s)}{\#\mathcal{AV}} \tag{2}$$

Assume for instance that we have two SPEs, x and y, and two partial logs containing identical access vectors for SPE x (say x^*) and different access vectors for SPE y (say y^1 and y^2). In this case we would have $weight(x) = \frac{1}{3}$ and $weight(y) = \frac{2}{3}$. Notice that this definition assigns larger weights to the SPEs whose access vectors are more likely to be different across different partial logs. When used in DSIM equation, this metric allows for biasing the selection towards pairs of logs having similarities in those SPEs that are more likely subject to different access interleavings. The rationale is that if we have that two partial logs having in common a "rare" access vector for a given SPE, then with high probability they were originated from equivalent executions.

Comparing the two metrics, one can see that the Plain Similarity considers that every SPE has the same importance, whilst the Dispersion-based Similarity assigns different weights to the SPEs. In general, both metrics allow to pinpoint the most similar partial logs, but the first is more useful when the overall-dispersion weight values are relatively well distributed for all the SPEs. On the other hand, the Dispersion-based Similarity is more suitable for cases when there are many SPEs whose access vectors are identical in every execution.

2) Relevance: This metric allows to classify each partial log according to its likelihood of being completed with compatible information:

$$Relevance(l_0) = \alpha \times \frac{\#Fill_{l_0,Sim_{l_0}}}{\#\mathcal{S}} + (1-\alpha) \times \frac{\sum_{l_n \in Sim_{l_0}} Similarity(l_0, l_n)}{\#Sim_{l_0}} \tag{3}$$

where $Similarity(l_0, l_n)$ is one of the two possible types of Similarity metrics.

The Relevance metric is the sum of two parcels weighed by the parameter $0 < \alpha < 1$, thus ensuring that $0 \leq Relevance \leq 1$. The first parcel captures the "completeness" of the set of SPEs obtainable by merging the partial logs, i.e. the number of SPEs that is possible to fill joining the access vectors from the partial log l_0 and its group of similars Sim_{l_0}. This follows the rationale that the more

missing SPEs of l_0 that can be filled with access vectors from similar partial logs, the better.

In turn, the second parcel provides a measure of the partial logs expected compatibility, by computing the average similarity of all the partial logs in the group of similars. This allows to pick, as the base partial log, the one whose group of similars is composed by partial logs with high similarity, thus increasing the probability of merging compatible information.

It should be noted that the value of each parcel is restricted to the range [0,1]. A value of 1 for the first parcel means that the full set of SPEs can be completed by combining the partial logs in Sim_{l_0}, whereas a value of 1 for the second parcel can only be achieved in the extreme case in which all logs in Sim_{l_0} are complete and identical.

Moreover, one should notice that a partial log l_1 can only be part of Sim_{l_0} if $Similarity(l_0, l_1) \geq threshold$. This avoids the group of similars to be composed by partial logs with a very low value of similarity. Also, the maximum size of the group of similars ($\#Sim_{l_0}$), can be defined by the developer.

In our experiments, we found $\alpha = 0.7$, $max(\#Sim_{l_0}) = 5$, $threshold = 0.3$ for Plain Similarity, and $threshold = 0.01$ for Dispersion-based Similarity (because the weight of some SPEs can be very low), to be good values.

F. Statistical Metrics for Bug Correlation

Unlike the previous metrics, the statistical metrics for bug correlation are concerned with the correlation between the bug and each access vector individually. This also leverages information from successful executions and is specially useful when, even after merging the partial logs of all buggy executions, there are still SPEs to be completed.

To compute these metrics, we adapt the scoring method proposed by Liblit et al [19]. Thereby, access vectors are classified based on their *Sensitivity* and *Specificity*, i.e. whether they account for many failed runs and few successful runs. With this information, it is possible to define a third metric, denoted *Importance*, which computes the harmonic mean between the previous two metrics, thus identifying the access vectors that are simultaneously high sensitive and specific.

Let F_{total} be the total number of partial logs resulting from failed executions; for each access vector v, let $F(v)$ be the number of failed partial logs that have recorded v for a given SPE, and $S(v)$ be the number of successful partial logs that have recorded v for a given SPE. The three metrics are then calculated as follows.

$$\textit{Sensitivity}(v) = \frac{F(v)}{F_{total}} \tag{4}$$

$$\textit{Specificity}(v) = \frac{F(v)}{S(v) + F(v)} \tag{5}$$

$$\textit{Importance}(v) = \frac{2}{\frac{1}{\textit{Sensitivity}(v)} + \frac{1}{\textit{Specificity}(v)}} \tag{6}$$

In summary, the higher the Importance value, the more correlated with the bug is the access vector.

G. Similarity-Guided Merge

To merge the partial logs and generate a complete log having high probability to replay the faulty execution, we developed a heuristic denoted *Similarity-Guided Merge*. This heuristic operates in the following five steps:

1. Calculate the degree of similarity between the partial logs: the first step consists of calculating the similarity between each partial log and all the others from the universe of partial logs received. To calculate the similarity, CoopREP applies the Plain Similarity metric or the Dispersion-based Similarity metric, to every possible pair of partial logs.

2. Identify the list of base partial logs: the next step consists of identifying the list of the partial logs that can be a potential good basis to start reconstructing the faulty execution. To build this list, CoopREP first calculates the relevance of each partial log and picks the n most relevant ones (we found $n = 10$ to be a suitable value for our experiments) in a descending order according to their relevance value.

3. Complete the base partial log with information from the group of similars: having already chosen the base partial log, CoopREP identifies the unrecorded SPEs in the base partial log and completes them with the respective access vectors traced by the logs in the group of similars. If all SPEs have been associated with an access vector, the obtained complete log is sent to the replayer, along with the thread ID map and the generated replay driver. On the other hand, if there are still empty SPEs, the heuristic proceeds with the next step.

4. Complete the base partial log with information from partial logs "similar by transitivity": when the access vectors from the group of similars are not sufficient to create a complete replay log, CoopREP tries to fill the missing SPEs with access vectors from the partial logs "similar by transitivity". These partial logs, although not belonging to the group of similars referred in the previous step, are part of the group of similars of those partial logs which are themselves similar to the base partial log. In other words, if $l_1 \in Sim_{l_0} \wedge l_2 \in Sim_{l_1} \Rightarrow l_2 \in Sim^2_{l_0}$, where $Sim^n_{l_0}$ contains the partial logs which are n^{th}-degree similar to l_0 (in this example, l_2 would be second-degree similar to l_0).

5. Complete the base partial log with statistical indicators: if it is still not possible to complete the log for replay (the union of the different groups of similars may not cover all the SPEs of the program), CoopREP applies the metrics described in Section IV-F to the universe of access vectors collected, and picks the ones with greater Importance (see Equation 6) to fill the missing SPEs.

At the end of this process, CoopREP replays the merged log and verifies if the bug is reproduced. If it is, the goal has

been achieved and the process ends. If it is not, CoopREP chooses the next partial log in the list of the most relevant to be the new base partial log and re-executes the Similarity-Guided Merge from the step 3. It should be referred that, in the worst case scenario, where all the most important indicators failed to replay the bug, the heuristic switches to a *brute force* mode. Here, all the possible access vectors are tested for each missing SPE.

V. Evaluation

A. Experimental Setting

All the experiments were conducted with machines Intel Core 2 Duo at 2.26 Ghz, with 4 GB of RAM and running Mac OS X. CoopREP prototype was implemented over a LEAP public version.In order to get comparative figures, this standard version of LEAP was also used in the experiments.

Regarding partial logging, we vary the percentage of the total SPEs logged in each run from 10% to 75%. For each configuration, 500 partial logs from different failed executions were used, plus more 50 of successful runs. To get a fairer comparison of the different recording schemes, the partial logs were generated from 500 complete logs, picking randomly the SPEs to be stored according to the scheme's percentage.

For the Plain Similarity we used a threshold of 0.3 and for the Dispersion-based Similarity we used a threshold of 0.01 (given that the weights of some SPEs may be very low). Regarding the maximum number of attempts of the heuristic to reproduce the bug, it was set to 500.

B. Evaluation Criteria

Three main criteria were used to evaluate CoopREP, namely: *i)* the bug replay capacity (consists of the number of attempts of the heuristic to replay the bug, therefore, the less number of tries, the better); *ii)* the performance overhead; and *iii)* the size of the partial logs produced. It should be noted that the two latter criteria were applied to both CoopREP and LEAP, in order to evaluate the benefits and limitations of our solution.

To assess CoopREP's bug replay capacity, we used some bugs from the IBM ConTest benchmark suite [24], and a real bug from the widely-used Java Application Server Tomcat. In order to measure the overheads imposed, we compared CoopREP against LEAP on the previous two applications and on the Java Grande workload benchmark.

Program	SPEs	Total Accesses	Bug Description
BoundedBuffer	12	1376	Notify instead of NotifyAll
BubbleSort	10	49964	Not-atomic
ProducerConsumer	8	997	Orphaned thread
Piper	6	347	Missing condition for Wait
Manager	4	30240	Not-atomic
TwoStage	4	27103	Two-stage
BufferWriter	3	50417	Wrong or No-Lock

Table II
Description of the ConTest benchmark bugs used in the experiments.

C. Bug Replay Capacity

1) ConTest Benchmark: The IBM ConTest benchmark suite [24] contains heterogeneous programs affected by several types of concurrency bugs. In order to be able to evaluate the effectiveness of CoopREP when distributing the logging burden across at least 4 clients, we restrict our analysis to the ConTest programs that have at least 4 SPEs. These are described in Table II, in terms of its number of SPEs, the total number of SPE accesses, and the bug-pattern according to [24].

Table III shows the number of attempts of the heuristic (using both Plain Similarity and Dispersion-based Similarity) to replay the ConTest benchmark bugs, when varying the percentage of SPEs recorded at each instance of the program in the range from 10% to 75%.

Analyzing the results, one can verify that the Similarity-Guided Merge heuristic only failed to replay the bug in programs *BoundedBuffer* (with percentage of SPEs recorded smaller than 50%), *Manager*, and *TwoStage* (only in the particular case of Plain-Similarity when logging 75% of the total SPEs). These results highlight that executions involving a high number of total SPE accesses are not necessarily harder to reproduce using partial logging strategies. The concurrency bug affecting the *BubbleSort* program, for instance, was always successfully reproduced at the first recombination of partial log, even though it is the second to entail the higher number of SPE accesses (around 50K). The actual effectiveness of the heuristic depends rather on how the accesses are distributed among the SPEs and, in particular, on how that distribution influences the SPEs' dispersion ratio. Here, the dispersion ratio indicates *how disperse is the SPE*, i.e. whether many different access vectors were recorded for it or not. More precisely, the dispersion ratio for an SPE is computed by dividing the number of *different* access vectors recorded for the SPE by the total number of access vectors recorded for that SPE (notice that this metric differs from the *overall-dispersion*, described in Equation 2). Figure 3 reports the SPE dispersion ratios for the ConTest benchmark programs (for the sake of readability and to ease the comparison, Figure 3 only presents the values for the full logging configuration).

We note that the *BubbleSort* program has only one SPE with a very high dispersion ratio (this SPE also accounts for about 99% of the total accesses), while the remaining SPEs always present the same access vector across all the executions. This allowed the Similarity-Guided Merge heuristic to quickly identify a set of partial logs whose combination yielded to a successful bug reproduction.

Program	Plain Similarity						Dispersion-based Similarity					
	10%	12.5%	16.7%	25%	50%	75%	10%	12.5%	16.7%	25%	50%	75%
BoundedBuffer	*X*	*X*	*X*	*X*	*X*	1	*X*	*X*	*X*	*X*	3	1
BubbleSort	1	1	1	1	1	1	1	1	1	1	1	1
ProducerConsumer	-	1	5	5	2	1	-	1	1	1	1	1
Piper	-	-	1	2	1	1	-	-	1	1	1	1
Manager	-	-	-	*X*	*X*	*X*	-	-	-	*X*	*X*	*X*
TwoStage	-	-	-	34	13	*X*	-	-	-	7	1	1
BufferWriter	-	-	-	11	3	1	-	-	-	11	3	1

Table III
NUMBER OF THE ATTEMPTS REQUIRED BY THE HEURISTIC TO REPLAY BUG IN THE CONTEST BENCHMARK. THE *X* INDICATES THAT THE HEURISTIC FAILED TO REPLAY THE BUG IN THE MAXIMUM NUMBER OF ATTEMPTS STIPULATED. "-" DENOTES THAT IT WAS NOT POSSIBLE TO ACHIEVE THE SPECIFIED PERCENTAGE OF PARTIAL LOGGING, GIVEN THE TOTAL NUMBER OF SPES PRESENT IN THE APPLICATION.

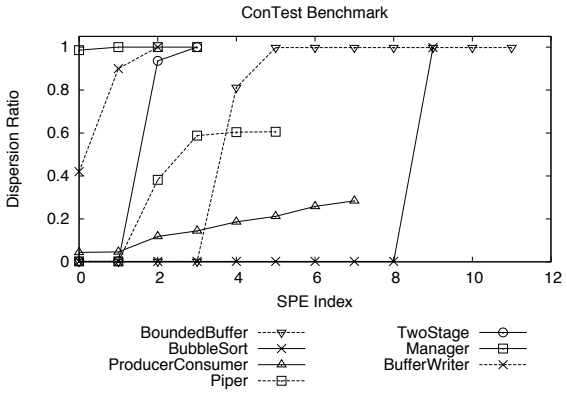

Figure 3. SPE dispersion ratios for the ConTest benchmark programs (sorted in ascending order), when logging all the SPEs of the program.

On the other hand, the *Manager* program has all its SPEs with a dispersion ratio of 1 or closer, which means that almost all the recorded executions had a different thread interleaving. These clearly represent unfavorable conditions for the partial logging approach, which in fact failed to replay the bug, as indicated in Table III. *BoundedBuffer* exhibits a similar outcome, however, as it has some SPEs with dispersion ratio less than 0.8, it was still possible to reproduce the error even when logging 50%, using Dispersion-based Similarity.

Regarding the *TwoStage* application, it presents unusual results when using Plain Similarity, since the bug was not replayed when the partial logs recorded more information. The explanation for this is related to the SPE dispersion ratios. As highlighted by Figure 3, out of the four program's SPEs, two were always identical (SPE 0 and 1), one had very few equal access vectors (SPE 2), and the last one was always different (SPE 3). Let us further discuss the three partial logging scenarios when using Plain Similarity:

75% of SPEs: with this configuration, each partial log was composed by three SPEs. Hence, the list of base partial logs ended being composed by the partial logs whose group of similars contained only other partial logs matching in the common SPEs. As a consequence, the access vectors combined for filling either SPE 2 or 3 were incompatible.

50% of SPEs: with this configuration, each partial log was composed by two SPEs. Here, the list of base partial logs was filled with the partial logs that have other ones matching in the SPE with very few identical access vectors (SPE 2). This because all the partial logs containing only SPEs 0 and 1, albeit having many other similar partial logs, could not generate a complete replay log just by combining information from their group of similars. Therefore, their relevance was lower (see Equation 3). The same did not happen for the partial logs containing SPE 2, which ended up composing the list of base partial logs. The bug was then successfully replayed by trying different access vectors for filling SPE 3.

25% of SPEs: with this configuration, each partial log was composed by a single SPE. Since there were no intersection points between the partial logs, the Similarity-guided Merge heuristic picked random partial logs to act as base to generate the replay log. Then, it tried to replay the error by successively filling the missing SPEs with the access vectors indicated by the statistical indicators. Nevertheless, the bug was successful replayed at the 34th attempt.

In fact, the *TwoStage* program is a good example to understand the differences between the two Similarity metrics. When using Dispersion-based Similarity, the heuristic could easily reproduce the bug, because the SPEs had different importance. Thus, the partial logs with the same access vector for SPE 2 were identified as the best base partial logs and used to generate a complete replay log.

As final remark, it should be noted that the addition of successful logs did not impact the results. The reason is due to the fact that when it was necessary to fill missing SPEs, there were always many different access vectors with the same degree of correlation to the bug.

2) Tomcat: Tomcat is a widely-used complex server application. The bug replay capacity of the Similarity-Guided

Plain Similiarity				Dispersion-based Similarity			
10%	**25%**	**50%**	**75%**	**10%**	**25%**	**50%**	**75%**
10	2	1	1	8	1	1	1

Table IV
NUMBER OF THE ATTEMPTS REQUIRED BY THE HEURISTIC TO REPLAY TOMCAT#37458 BUG.

Merge heuristic was tested with bug `#37458` [2] of Tomcat v5.5. This error consists of a `NullPointerException`, resulting from a data race, and was already used in [9] to test LEAP.

In this case only 15 SPEs (out of the total 32 SPEs of Tomcat) were logged by CoopREP. This depends on the fact that the available unit test designed to trigger the bug only exercises a subset of Tomcat's SPEs. Interestingly, even despite the availability of an aimed unit test, triggering the bug was not trivial, as the bug only manifested itself on average after 112 attempts.

Table IV shows the number of attempts of the Similarity-Guided Merge heuristic (using both Plain Similarity and Dispersion-based Similarity) to replay the Tomcat#37458 bug, when logging from 10% to 75% of the SPEs. The data confirms the effectiveness of the proposed statistical analysis techniques, and in particular of the one based on Dispersion-based similarity, which was able to identify compatible combinations of partial logs and to replay successfully the bug in less than 10 attempts even when logging accesses to only 10% of the SPEs.

D. Performance Overhead

In this section we analyze the performance benefits achievable via the CoopREP scheme. For space constraints, we focus here on the ConTest and the Java Grande Forum benchmarks, as the results related Tomcat show very similar trends.

1) ConTest Benchmark: Figure 4 reports the performance overhead on the tested programs. By using partial recording, CoopREP achieved always lower runtime degradation than LEAP. The results highlight that the overhead reductions are not necessarily linear. This is due to the fact that some SPEs are accessed significantly more frequently than others. Given that the instrumentation of the code is performed statically, the load balance in terms of thread accesses may not be equally distributed among the users, as previously referred in Section IV-C.

Clearly, the advantage of using partial logging is higher in scenarios where the usage of full logging has a higher negative impact on performance. The most notorious case are *BubbleSort* and *BufferWriter* , which are the applications with the highest number of SPE accesses (see Table II). For example, in *BubbleSort*, LEAP imposed a performance overhead of 26%, while CoopREP imposed only a 7% overhead when recording 10% of the total SPEs (which was sufficient to successfully replay the bug).

[2]https://issues.apache.org/bugzilla/show_bug.cgi?id=37458

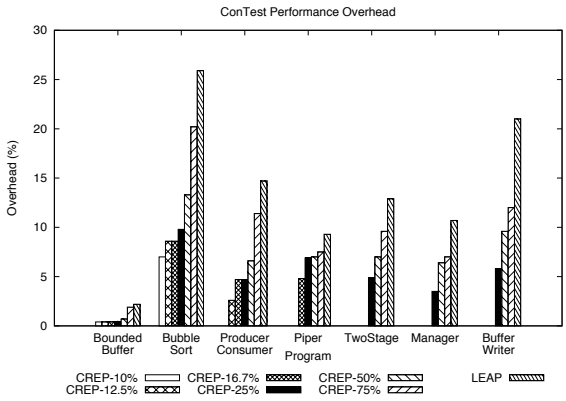

Figure 4. Performance overheads for the ConTest benchmark programs. LEAP corresponds to the recording configuration of 100%.

Program	SPEs	Total Accesses
Raytracer	16	2.56×10^9
SparseMatmult	8	5.08×10^7
SOR	8	1.99×10^6
Montecarlo	15	1.50×10^5
Series	8	2.00×10^4

Table V
DESCRIPTION OF THE JAVA GRANDE FORUM BENCHMARK PROGRAMS USED IN THE EXPERIMENTS.

Overall, the results in Figure 4 highlight that, at least for these benchmarks, the choice of logging at most 25% of the SPEs lead to a runtime penalty that is consistently lower than 10%: a threshold typically regarded as the maximum acceptable overhead for real world applications [4], [20].

2) Java Grande Forum: The Java Grande Forum benchmark contains computationally intensive science and engineering applications that require high-performance computers. Given that Java Grande Forum Benchmark does not have known bugs, it was only used in our experiments to assess the benefits and limitations of CoopREP when compared to LEAP, on demanding computing environments. Table V describes the benchmark programs used in terms of number of SPEs and the overall number of times that they are accessed. For the sake of readability, the results of the tests performed are presented in tables, since the values obtained vary within a large scale.

Table VI contains the experiments with respect to the performance overhead measured when tracing the SPEs with the previous logging configurations.

The results show that the logging overhead can be dramatically abated by CoopREP, especially for memory intensive applications, such as *RayTracer* or *SparseMatmult*. In the former case, the average logging overhead drops by a factor around 50x when configuring the CoopREP to log 10% of

the application's SPEs. Analogous trends can be observed also for the other benchmarks, providing an additional experimental evidence at support of the significant performance gains achievable via cooperative logging schemes.

E. Log Sizes

We now quantify the performance benefits achievable by CoopREP from an alternative perspective, namely the amount of logs generated with respect to a non-cooperative logging scheme, such as LEAP.

1) ConTest Benchmark: Figure 5 reports the results obtained for the ConTest benchmark, showing the ratio between the size of the logs generated by various partial recording configurations and the size of the logs generated by LEAP. Unsurprisingly, the log size ratios follow a trend that is analogous to that observed in the performance overhead plots reported in Figure 4. The *Manager* benchmark resulted to be the program for which CoopREP show a reduction ratio more similar to the one expected by decreasing proportionally the recording percentage. In turn, *BubbleSort* was the program where the log sizes decreased faster even for small reduction of the total number of logged SPEs (the log size ratios obtained, with respect to LEAP, were 0.36 and 0.62, when logging 50% and 75% of the SPEs, respectively). However, the reduction obtained when logging less than 50% of the SPEs was not very significant. This is due to the fact that the largest fraction of accesses is confined to a single SPE, which was recorded the same number of times for the ten logs measured for these configurations.

On the other hand, *Piper* was the application where decreasing the percentage of logged SPEs led to smaller reductions of the log sizes. This is again explainable by the heterogeneity in the size of the access vectors associated with the various SPEs. In this application, recording 25% of the SPEs only led to a log size ratio of 0.55.

2) Java Grande Forum: In Figure 6 we report the results concerning the log size ratios with respect to LEAP when using the Java Grande Forum.

From the figure analysis, the benefits of partial logging are clear. The most evident case is *SOR*, where the log sizes when logging less than 25% of the SPEs account for at most 0.1% of LEAP's log size. In fact, for all the benchmark

Program	Performance Overhead			
	10%	**25%**	**50%**	**LEAP**
Raytracer	1757.5%	9566.7%	17452.1%	92908.4%
SparseMatmult	571.6%	598.2%	1725.5%	2606.7%
SOR	0.8%	1.1%	2.0%	2.7%
Montecarlo	1.1%	1.5%	2.3%	7.3%
Series	0.08%	0.1%	0.4%	6.5%

Table VI
PERFORMANCE OVERHEADS FOR THE JAVA GRANDE FORUM BENCHMARK PROGRAMS.

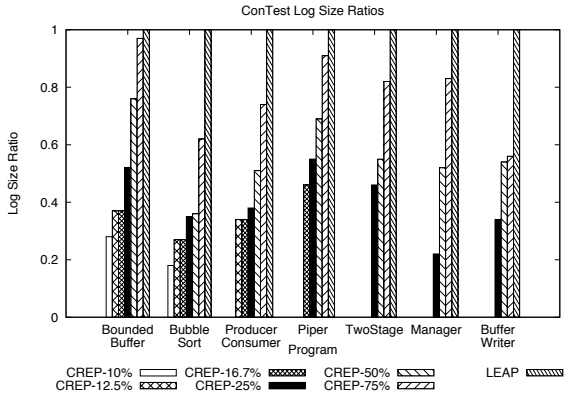

Figure 5. Log size ratios for the ConTest benchmark programs.

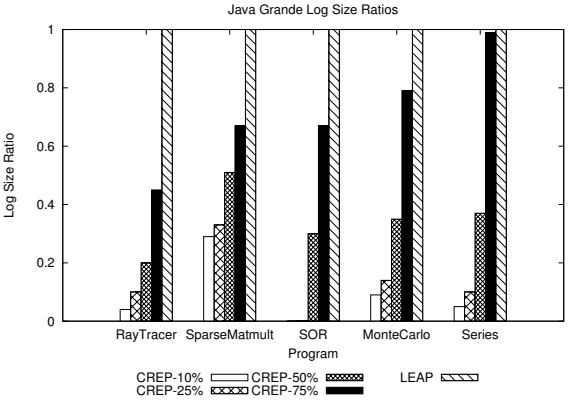

Figure 6. Log size ratios for the Java Grande benchmark programs.

programs, there was a high heterogeneity in the size of the access vectors of the program SPEs, which significantly influenced the actual reduction in the log sizes. In other words, the decreases are just not completely linear because some SPEs are accessed more times than others. Given that the instrumentation of the code is performed using a purely random approach, the load in terms of logged SPE accesses may not be equally distributed among the different runs, as previously referred in Section IV-C. This implies that the impact of logging $x\%$ of the SPEs will not necessarily mean a reduction of $x\%$ in both performance overhead and log size. In fact, sometimes the average reduction may be greater than expected (as in *Raytracer* and *SparseMatmult*), but other times may be lower (as in *Series* when logging 75% of the SPEs). This motivates future research in how one can equally distribute the information to be recorded among the different clients.

VI. CONCLUSIONS

This paper introduced CoopREP, a system that provides fault replication of concurrent programs, through cooperative recording and partial log combination. CoopREP achieves

remarkable reductions of the overhead incurred in by conventional deterministic execution replayer by letting each instance of a program trace only a subset of its shared programming elements (e.g. variables or synchronization primitives). CoopREP relies on several innovative statistical analysis techniques aimed at guiding the search of partial logs to combine and use during the replay phase.

The evaluation study, performed with third-party benchmarks and a real-world application, highlighted both the effectiveness of the proposed technique, in terms of its capability to successfully replay non-trivial concurrency bugs, as well as its performance advantages with respect to non-cooperative logging schemes.

We believe that this work opens a number of challenging research directions, including the design of additional partial logging schemes (e.g. taking into account load balancing or locality of SPEs, or maximizing the probability of logging overlapping SPEs) and new similarity metrics (e.g. that use euclidean or edit distances between access vectors).

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for the valuable feedback and suggestions. The authors also wish to thank Jeff Huang for his availability to answer the countless questions about LEAP. This work was partially supported by FCT (INESC-ID multi-annual funding) through the PIDDAC program funds, and by the European project "FastFix" (FP7-ICT-2009-5).

REFERENCES

[1] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *ACM ASID*, 2006, pp. 25–33.

[2] A. Hall, "Realising the benefits of formal methods," *Journal of Universal Computer Science*, vol. 13, no. 5, pp. 669–678, 2007.

[3] D. Parnas, "Really rethinking 'formal methods'," *Computer*, vol. 43, pp. 28–34, 2010.

[4] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu, "Pres: probabilistic replay with execution sketching on multiprocessors," in *ACM SOSP*, 2009, pp. 177–192.

[5] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen, "Execution replay of multiprocessor virtual machines," in *ACM VEE*, 2008, pp. 121–130.

[6] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere, "Jarec: a portable record/replay environment for multithreaded java applications," *Software Practice and Experience*, vol. 40, pp. 523–547, May 2004.

[7] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *ACM SPDT*, 1998, pp. 48–59.

[8] T. LeBlanc and J. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.*, vol. 36, pp. 471–482, April 1987.

[9] J. Huang, P. Liu, and C. Zhang, "Leap: lightweight deterministic multi-processor replay of concurrent java programs," in *ACM FSE*, 2010, pp. 385–386.

[10] G. Pokam, C. Pereira, K. Danne, L. Yang, and J. Torrellas, "Hardware and software approaches for deterministic multiprocessor replay of concurrent programs," *Intel Technology Journal*, vol. 13, pp. 20–41, 2009.

[11] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[12] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference*, 2004, pp. 29–44.

[13] M. Xu, R. Bodik, and M. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *ISCA*. ACM, 2003, pp. 122–135.

[14] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *IEEE ISCA*, 2005, pp. 284–295.

[15] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *IEEE ISCA*, 2008, pp. 123–134.

[16] P. Montesinos, M. Hicks, S. King, and J. Torrellas, "Capo: a software-hardware interface for practical deterministic multiprocessor replay," in *ACM ASPLOS*, 2009, pp. 73–84.

[17] G. Altekar and I. Stoica, "Odr: output-deterministic replay for multicore debugging," in *ACM SOSP*, 2009, pp. 193–206.

[18] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *ACM EuroSys*, 2010, pp. 321–334.

[19] B. Liblit, A. Aiken, A. Zheng, and M. Jordan, "Bug isolation via remote program sampling," in *ACM PLDI*, 2003, pp. 141–154.

[20] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *ACM OOPSLA*, 2010, pp. 241–255.

[21] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *IEEE DSN*, 2010, pp. 221–230.

[22] R. Halpert, C. Pickett, and C. Verbrugge, "Component-based lock allocation," in *IEEE PACT*, 2007.

[23] M. Xu, R. Bodík, and M. Hill, "A serializability violation detector for shared-memory server programs," in *ACM PLDI*, 2005, pp. 1–14.

[24] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *IEEE IPDPS*, 2003, pp. 286–293.