Received: 19 February 2016 | Revised: 26 July 2017 | Accepted: 26 July 2017
DOI: 10.1002/stvr.1645

WILEY

RESEARCH ARTICLE

CoopREP: Cooperative record and replay of concurrency bugs

Nuno Machado[1,2] | Paolo Romano[2] | Luís Rodrigues[2]

[1]HASLab—INESC TEC, Universidade do Minho, Braga, Portugal
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal

Correspondence
Nuno Machado, Departamento de Informática, Universidade do Minho, 4710-057 Braga, Portugal.
Email: nuno.a.machado@inesctec.pt

Funding information
Fundação para a Ciência e a Tecnologia (FCT), Grant/Award Number: UID/CEC/50021/2013; project "TEC4Growth—Pervasive Intelligence, Enhancers and Proofs of Concept, Grant/Award Number: NORTE-01-0145-FEDER-000020; North Portugal Regional Operational Programme, Grant/Award Number: NORTE 2020; PORTUGAL 2020 Partnership Agreement; European Regional Development Fund (ERDF)

Summary

This paper presents CoopREP, a system that provides support for fault replication of concurrent programs based on cooperative recording and partial log combination. CoopREP uses partial logging to reduce the amount of information that a given program instance is required to store to support deterministic replay. This allows reducing substantially the overhead imposed by the instrumentation of the code, but raises the problem of finding a 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 the partial logs to be combined and needed for the replay phase. CoopREP has been evaluated using both standard benchmarks for multithreaded applications and real-world applications. The results highlight that CoopREP can successfully replay concurrency bugs involving tens of thousands of memory accesses, while reducing recording overhead with respect to state-of-the-art noncooperative logging schemes by up to 13× (and by 2.4× on average).

KEYWORDS

concurrency errors, debugging, partial logging, record and replay

1 | INTRODUCTION

Concurrent programming is of paramount importance to exploit the full potential of the emerging multicore architectures. However, writing and debugging concurrent programs is notoriously difficult. Contrary to most bugs in sequential programs, which usually depend exclusively on the program input and the execution environment (and, therefore, can be more easily reproduced), concurrency bugs depend on nondeterministic interactions among threads. This means that even when re-executing the same code with identical inputs, on the same machine, the outcome of the program may differ from run to run [1].

Deterministic replay (or *record and replay*) addresses this issue by recording nondeterministic events (such as the order of access to shared-memory locations) during a failing execution and, then, use the resulting trace to support the reproduction of the error [2]. Classic approaches, [3-6] also referred to as *order-based*, trace the relative order of all relevant events, thus allowing to replay the bug at the first attempt. Unfortunately, they also come with an excessively high recording cost (10×-100× slowdown), which is impractical for most settings.

Motivated by the observation that the most significant performance constraints are on production runs, more recent solutions have adopted a *search-based* approach [1,7-9]. Search-based solutions reduce the recording overhead at the cost of a longer reproduction time during diagnosis. To this end, they typically log incomplete information at runtime and rely on post-recording exploration techniques to complete the missing data. These techniques explore various trade-offs between recording overhead and replay efficacy (ie, number of replay attempts required to reproduce the bug).

The work in this paper aims at further reducing the overhead achievable using either order-based or search-based techniques, by devising cooperative logging schemes that exploit the coexistence of multiple instances of the same program. The underlying intuition is simple: to share the burden of logging among multiple instances of the same (buggy) program, by having each instance tracking accesses to a random subset of shared variables. The partial logs collected from production runs are then statistically analyzed *offline* to identify those partial logs whose combination maximizes the chances to successfully replay the bug.

This approach is implemented in a system named CoopREP (standing for **Coop**erative **R**ecord and r**EP**lay). In a nutshell, CoopREP is a search-based record and replay framework that leverages cooperative logging performed by multiple user instances* and statistical techniques for effective partial log combination. CoopREP benefits classic order-based solutions by reducing the overhead of capturing the complete execution schedule in a single production run, although potentially sacrificing the guarantee of immediate bug replay. For search-based solutions, which already trace partial information at runtime, CoopREP offers the ability of further reducing the record cost, while still achieving similar bug replay abilities.

An extensive experimental evaluation was performed to demonstrate the effectiveness of CoopREP when used in combination with 2 state-of-the-art noncooperative logging schemes, namely, LEAP [6] (order-based) and PRES [1] (search-based). The results of the experiments on 13 Java programs, comprising both third-party benchmark and real-world concurrency bugs, show that CoopREP versions achieve significant reductions of runtime overhead when compared to the respective noncooperative logging solutions (up to 6× with respect to LEAP [6] and up to 13× with respect to PRES [1]). In terms of space overhead, CoopREP generates logs that are, on average, 3.2× and 4.2× smaller than those produced respectively by LEAP and PRES, while still providing similar bug replay guarantees.

In summary, this paper shows that *cooperative logging is a viable strategy to replicate concurrency bugs with tolerable overhead*. Furthermore, the paper makes the following contributions: (1) a set of novel statistical metrics to detect similarities between partial logs, independently captured by different user instances of the same program; (2) 2 novel heuristics that leverage the similarity metrics above to efficiently combine partial logs to obtain a complete failure-inducing execution schedule; (3) an experimental evaluation of the current CoopREP prototype applied to 2 state-of-the-art record and replay systems, using standard benchmarks for multithreaded applications and bugs from real-world applications.

The rest of this document is structured as follows: Section 2 presents the background concepts related to this work. Section 3 overviews some related work in the deterministic replay topic. Section 4 introduces CoopREP, describing in detail its architecture. Section 5 presents the statistical metrics used to measure similarity between partial logs. Section 6 describes the 2 heuristics, which leverage the similarity metrics to merge partial logs and replay concurrency bugs. Section 7 shows how the heuristics fit in CoopREP's execution flow. Section 8 presents the results of the experimental evaluation and discusses CoopREP's major benefits and limitations. Finally, Section 9 concludes the paper by summarizing its main points and discussing future work.

2 | BACKGROUND

This section overviews some of the background concepts most related to this work. The section starts by introducing the deterministic replay technique and, then, discusses the main sources of nondeterminism in concurrent programs. Finally, the section describes the most common types of concurrency bugs.

2.1 | Deterministic replay

Deterministic replay (or *record and replay*) aims to overcome the problems associated with the reproduction of concurrency bugs. The purpose of this technique is to allow re-executing a multithreaded program, obtaining a behavior similar to the original failing execution. This is possible as long as all nondeterministic factors that have an impact on the program's execution are replayed in the same way [1].

In general, deterministic replay operates in 2 phases: (1) *record phase*, during which information regarding nondeterministic events is captured into a trace file, and (2) *replay phase*, during which the application is re-executed using the trace file to ensure that the order of execution of nondeterministic events is compliant with that of the original execution.

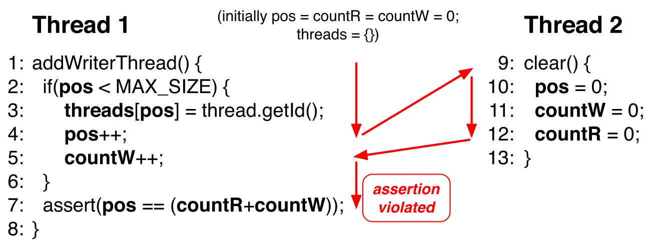
Deterministic replay targets mainly debugging (which is also the purpose of this work). However, the ability to deterministically reproduce a given execution of a program can be also useful for several other purposes, namely, fault tolerance (to efficiently replicate the application state across different replicas [10]), testing (to force the execution of a program along different thread interleavings in an attempt to increase the chances of finding errors in existing tests [11]), and security (to examine the state of the system during an attack [12]).

2.2 | Concurrency bugs

Concurrency bugs are faults in the code of a program that incorrectly permit a thread schedule that results in unintended behavior, such as a crash or data corruption[13]. Common examples of concurrency bugs are *atomicity violations*, *order-violations*, and *deadlocks*. An atomicity violation occurs when a pair of accesses to shared state by a given thread is incorrectly interleaved by an access by another thread. An order-violation happens when operations in different threads execute in an order that leads to a failure. Finally, a deadlock is characterized by threads acquiring locks in a way that prevents the execution from making progress.

Figure 1 shows a code snippet of a multithreaded program with a concurrency bug. This program has 2 threads and 4 shared variables (`pos`, `countR`, `countW`, and `threads`). If the program follows the execution depicted by the arrows, it will violate the assertion on line 7.

* For simplicity of description, the terms *user instance* of a program and *instance* are used interchangeably in this paper.



```
1: addWriterThread() {
2:   if(pos < MAX_SIZE) {
3:       threads[pos] = thread.getId();
4:       pos++;
5:       countW++;
6:   }
7:   assert(pos == (countR+countW));
8: }
```

```
 9: clear() {
10:    pos = 0;
11:    countW = 0;
12:    countR = 0;
13: }
```

FIGURE 1 Example of an atomicity violation bug. Accesses to shared variables are depicted in bold

This error is an atomicity violation: Thread 1 increments the position counter after inserting a value in the array (`pos++` at line 4), but before incrementing the counter `countW` (line 5), it is interleaved by thread 2, which sets both variables to 0 (lines 10 and 11). As such, when thread 1 reaches the assertion, the condition will fail because `pos = 0` and `countW = 1`. Section 4 describes how CoopREP operates to reproduce concurrency bugs like the one in Figure 1.

3 | RELATED WORK

Debugging concurrent programs on multiprocessors via deterministic replay has been the subject of extensive research over the past decade. On the basis of how the solutions are implemented, prior work can be divided into 2 main categories: *hardware-based* and *software-based*.

Hardware-based systems rely on hardware extensions to efficiently record the nondeterministic events and, consequently, mitigate the overhead imposed to the production run. Flight Data Recorder [14] and BugNet [15] have provided solutions to achieve this, but at a cost of nontrivial hardware extensions. More recently, DeLorean [16] and Capo [17] have proposed new techniques to reduce the complexity of the extensions. Despite that, they still require changing noncommodity hardware, which can be costly. For this reason, this work concerns only software-based approaches.

Regarding software-based systems, they can be broadly classified as *order-based* or *search-based*. LEAP, [6] Order, [18] and CARE [19] are state-of-the-art order-based solutions that record the exact read-write linkage of shared-memory accesses. They essentially vary in the technique used to minimize contention when recording the order of events: LEAP tracks shared accesses on a per-variable basis, Order traces thread interleavings at the object instance level, and CARE exploits thread local caches to identify the subset of all exact read-write linkages that have to be recorded. CoopREP strives to further minimize the overhead imposed by pure order-based approaches by sharing the burden of recording the execution schedule across different program instances.

Search-based solutions differ from order-based ones in the sense that they capture only partial information at runtime. As such, these systems need to synthesize offline a complete bug-inducing execution through state space exploration. For example, ESD [8] does not record any information at all and attempts to infer the failing schedule by analyzing the program symbolically, with hints from the core dump. PRES [1] logs an execution sketch and relies on an intelligent offline replayer to explore the space of the nonrecorded data race outcomes that also conform with the sketch. ODR [7] logs program's outputs and inputs, as well as the execution path and the locking order, and encodes the search space as a constraint solving problem. CLAP [9] improves ODR by both parallelizing the constraint solving phase and encoding the constraint formula as a problem of ordering shared read and operations instead of finding an execution with similar output to the original run.

CoopREP is also best-effort, but improves on previous search-based solutions by leveraging information logged by multiple clients to ease the inference task. To this end, CoopREP draws on statistical debugging techniques, which analyze information gathered from a large number of users to isolate bugs. Statistical debugging was pioneered by Cooperative Bug Isolation (CBI) [20]. The CBI samples certain predicates of the program at runtime (eg, branches and return values) and uses statistical indicators to pinpoint which predicates are likely to be the source of a sequential bug. However, CBI does not support concurrency bugs. The Cooperative Crug Isolation (CCI) [21] overcomes this limitation by adjusting CBI's principles to support tracking thread schedules. For instance, it implements cross-thread sampling and relies on longer sampling periods to capture multiple thread accesses to shared memory. CoopREP is fundamentally different from CCI though, because it focuses on bug reproduction rather than root cause diagnosis.

4 | COOPREP OVERVIEW

4.1 | System architecture

Figure 2 depicts the overall architecture of CoopREP, which entails 5 components: (1) the logging profile generator, (2) the transformer, (3) the recorder, (4) the statistical analyzer, and (5) the replayer. Each component is described in detail in the following.

Logging profile generator. The logging profile generator receives the target program as input and identifies the points in the code that should be monitored at runtime. The resulting logging profile is then sent to the transformer. Note that this profile is, in fact, a *partial logging profile*, because it contains only a subset of all the relevant events required to replay the concurrent execution.

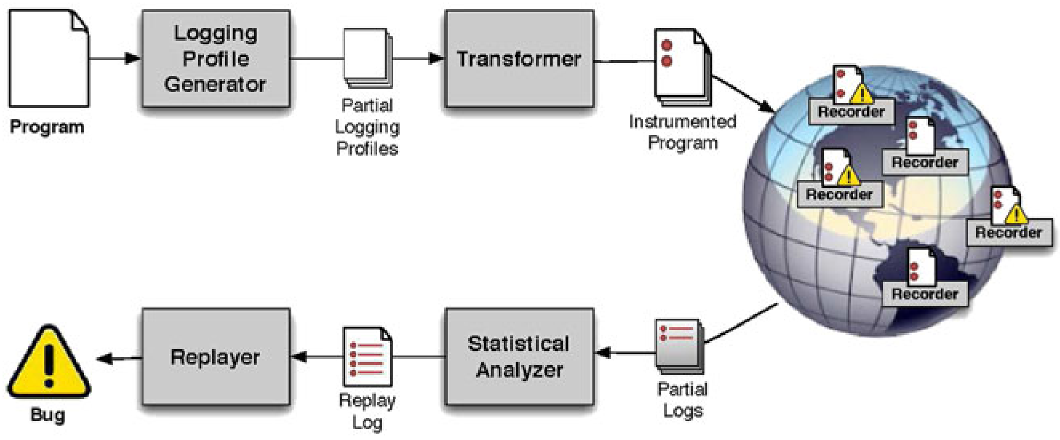


FIGURE 2 Overview of CoopREP's architecture and execution flow

Logging profiles can comprise different kinds of information (eg, function return values, basic blocks, and shared accesses). However, CoopREP focuses on capturing accesses to synchronization variables (eg, locks and Java monitors), as well as class and instance variables that can be manipulated by multiple threads. To locate these *shared program elements* (SPEs), CoopREP uses a static escape analysis denoted *thread-local objects analysis* [22] from the Soot[†] framework. Since accurately identifying shared variables is generally an undecidable problem, this technique computes a sound over-approximation, ie, every shared access to a field is indeed identified, but some nonshared fields may also be classified as shared[6].

Transformer. The transformer is responsible for instrumenting the program. It consults the logging profile to inject event-handling runtime calls to the recorder component. The transformer also instruments the instructions regarding thread creation and end points of the program. User instances will then execute a given instrumented version of the program, instead of the original one.

To ensure consistent thread identification across all executions, CoopREP follows an approach similar to that of jRapture [23]. The underlying idea of this approach is that each thread should create its children threads in the same order, even though there may not be a consistent global order among all threads. Concretely, CoopREP instruments the thread creation points and modifies the new Java thread identifiers by new identifiers on the basis of the parent-children order relationship. To this end, each thread maintains a local counter to store the number of children it has forked so far. Whenever a new child thread is forked, its identifier is defined by a string containing the parent thread's ID concatenated with the value of the local counter at that moment. For instance, if a thread t_i forks its jth child thread, which in turn forks its kth thread, then the latter thread's identifier will be $t_{i:j:k}$.

Recorder. There is a recorder per user instance. The recorder monitors the execution of the instance and captures the relevant information into a log file. Then, when the execution ends or fails, the user sends its *partial log* to the statistical analyzer.

CoopREP can be configured to trace different kinds of information, but the work in this paper uses the approach proposed by LEAP [6] to track the thread interleaving[‡]. In particular, CoopREP associates each SPE with an *access vector* that stores, during the production run, the identifiers of the threads that read/write on that SPE. For instance, for the execution illustrated in Figure 1, a complete log would have the following access vectors: `pos = [1,1,1,1,2,1]`, `countW = [2,1,1,1]`, `countR = [2,1]`, and `threads = [1]`.

Using this technique, one gets several vectors with the order of the thread accesses performed on individual shared variables, instead of a single global-order vector. This provides lightweight recording, but relaxes faithfulness in the replay, given that it allows for reordering the replay of 2 not causally related memory accesses. Still, it has been shown that this approach does not affect the correctness of the replay (a formal proof of the soundness of this statement can be found in Huang et al [6]).

Conversely to LEAP, CoopREP's recorder does not log all access vectors. Instead, each user traces accesses to only a subset of the program's SPEs, as defined in the partial logging profile. A possible partial log for the example in Figure 1 could be {`countW = [2,1,1,1]`; `countR = [2,1]`}. In addition to access vectors, each CoopREP's partial log contains a flag indicating the success or failure of the execution (successful runs can be useful for the statistical analysis, as it will be shown in Section 7).

Assuming that the program is executed by a large population of users, not only does this mechanism allow for gathering access vectors for the whole set of SPEs with high probability but it also reduces the performance overhead incurred by each user instance due to full logging.

Statistical analyzer. The statistical analyzer is responsible for gathering and analyzing the partial logs recorded during production runs (from both successful and failed executions) to produce a complete replay trace, capable of yielding an execution that triggers the bug observed at runtime. The statistical analyzer relies on 2 subcomponents: the *candidate generator* and the *replay oracle*, as depicted in Figure 3.

The candidate generator uses one of 2 novel heuristics (see Section 6) to identify similarities in partial logs and pinpoint those that are more likely to be successfully combined to reproduce the failure. In other words, the candidate generator tries to aggregate partial logs into smaller

[†] http://www.sable.mcgill.ca/soot
[‡] In Section 8, CoopREP is also evaluated using a search-based record-and-replay approach, in addition to LEAP's order-based scheme.

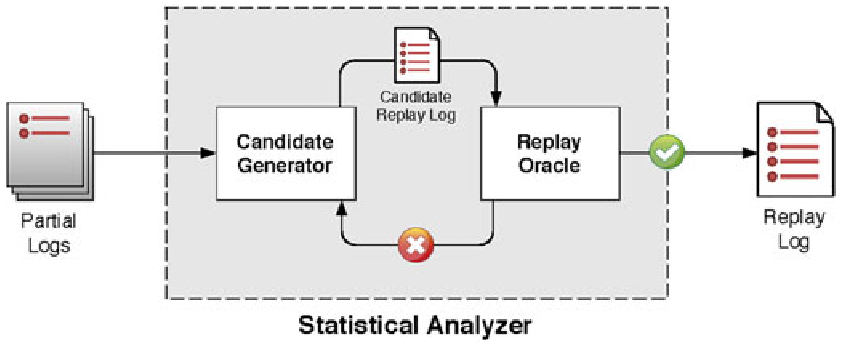


FIGURE 3 Detailed view of the statistical analyzer component

groups, where each group contains logs originated from *similar* production runs—where the similarity between runs is evaluated by means of several statistical metrics (see Section 5). The rationale is that partial logs originated from similar executions are likely provide compatible information with respect to the thread interleaving that leads to the bug.[§]

The resulting candidate replay log (containing the access vectors for all the SPE in the program) is then sent to the *replay oracle*, which checks the effectiveness of the replay log in terms of bug reproducibility. Concretely, the oracle enforces an execution that respects the schedule defined in the replay candidate trace and checks whether the error is triggered.

In case the bug is not reproduced, the replay oracle asks the candidate generator for another candidate trace, and the procedure is repeated until the bug is successfully replayed or until the maximum number of attempts to do so is reached. This search procedure is detailed in Section 7.

Given that access vectors come from independent executions, the resulting candidate replay log may be infeasible, meaning that the replay oracle may fail to enforce the thread execution order specified in the replay log. In this case, the execution will hang, as none of the threads will be allowed to perform its subsequent access on an SPE. CoopREP copes with this issue by using a simple, yet effective, timeout mechanism, which terminates immediately the execution replay as soon as it detects that all threads are prevented from progressing.

Finally, the replay oracle also needs to deal with the case in which the bug is not triggered. For crash failures, CoopREP simply detects the absence of the exception. On the other hand, to detect generic incorrect results, CoopREP requires programmers to provide information regarding the failure symptoms (eg, specifying predicates on the application's state that can be dynamically checked via assertions).

Replayer. Once the statistical analyzer clog, the developers can then use the replayer to re-execute the application and observe the bug being deterministically triggered in every run. This allows developers to re-enable *cyclic debugging*, as they can run the program repeatedly in an effort to incrementally refine the clues regarding the error's root cause.

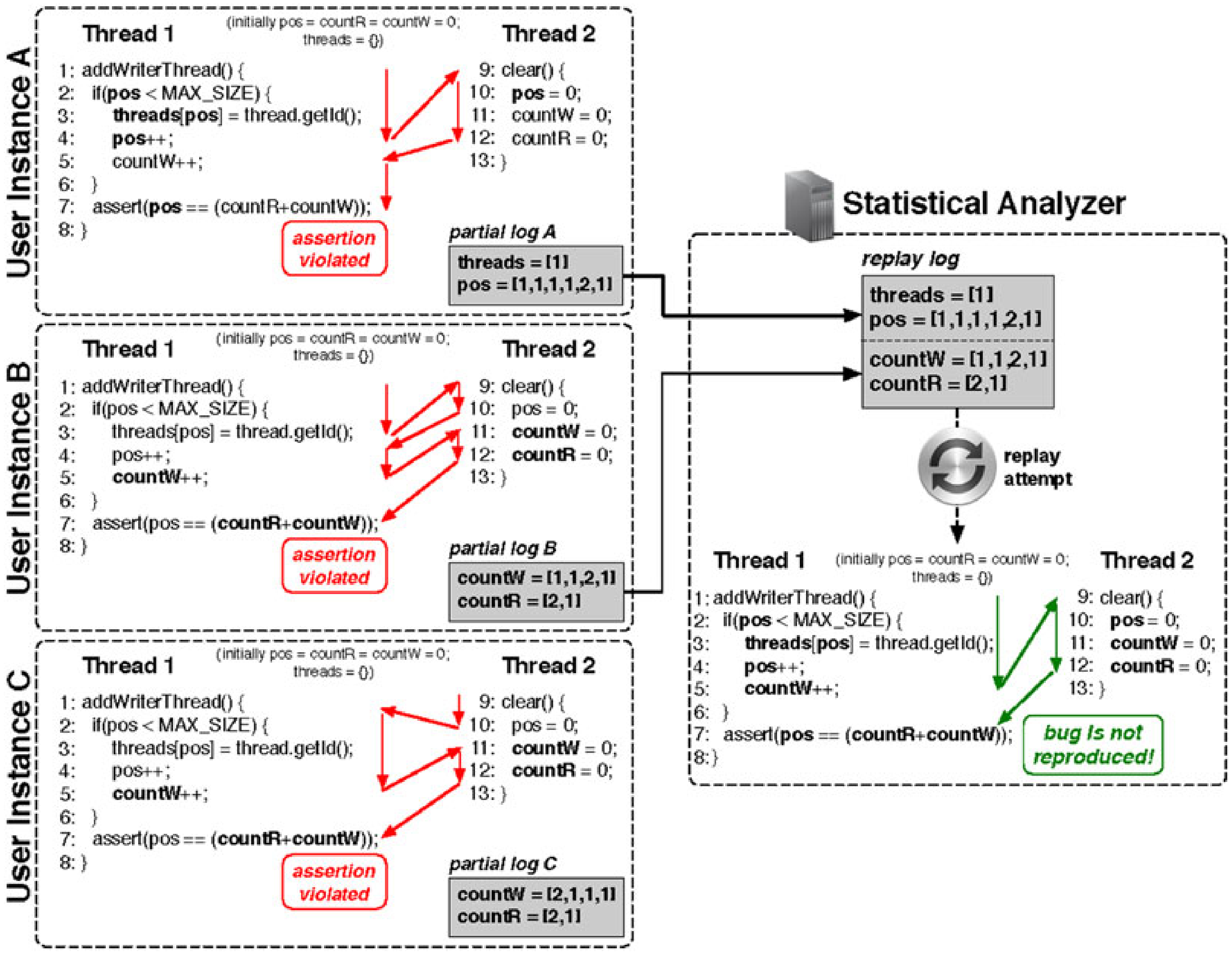
Observing the architecture depicted in Figure 2, one can identify 2 main challenges in the cooperative record and replay approach: (1) *how to devise the partial recording profiles* and (2) *how to combine the partial logs recorded during production runs to generate a feasible fault-inducing replay trace*. The next sections present CoopREP's approach to address these 2 issues, with special emphasis on the techniques devised to merge partial logs.

4.2 | Partial log recording

CoopREP is based on having each program instance recording accesses to only a fraction of the entire set of SPEs in the program. The subset of SPEs to be traced is defined by the logging profile generator prior to the instrumentation of the target program instances. The logging profile generator relies on a probabilistic scheme that selects a subset of the total SPEs using a uniform random distribution. The cardinality of the set of SPEs logged by CoopREP is a parameter configurable by the developer. This approach has the key benefit of being extremely lightweight and to ensure statistical fairness; ie, on average, all SPEs have the same probability of being traced by a user instance. Thus, each instance has the same probability of logging SPEs that are very frequently/rarely accessed. As it will be shown in Section 8.5, the latter factor has indeed a significant impact on the actual recording overhead of a given instance. Despite that, it would be relatively straightforward to increase the flexibility of the information recorded at runtime by extending the current recording framework to support dynamic instrumentation with hints provided by the developer or by program analysis techniques[24]. Note that dynamic instrumentation would not hinder performance heavily, as the runtime overhead is dominated by the additional synchronization injected to track the thread interleaving.

Long running executions, in turn, still pose a challenge for partial logging approaches. Similarly to full-logging systems, CoopREP has to reenact the original execution from the beginning, which might not be practical in long running applications (because of the long replay time and large log size). A possible way to circumvent this issue is to extend CoopREP to leverage periodic checkpointing. This way, CoopREP could focus on collecting partial logs only recorded after the last checkpoint.

[§]This paper assumes that all partial logs from failed executions refer to the same bug. In practice, one may cope with applications that suffer of multiple bugs by distinguishing them using additional metadata, such as the line of code in which the bug manifested or the type of exception that the bug generated.



```
addWriterThread() {
  if(pos < MAX_SIZE) {
    threads[pos] = thread.getId();
    pos++;
    countW++;
  }
  assert(pos == (countR+countW));
}
```

```
clear() {
  pos = 0;
  countW = 0;
  countR = 0;
}
```

FIGURE 4 Example of a scenario where 2 failing partial logs originate a replay log that does not trigger the bug

4.3 | Merge of partial logs

Another major challenge of cooperative record and replay, and the main focus of this paper, is how to combine the collected partial logs in such a way that the resulting thread interleaving leads to a feasible execution, capable of reproducing the bug during the replay phase.

In general, the following factors can make partial log merging difficult to achieve: (1) The bug can result from the interleaving of multiple threads; (2) the combination of access vectors from partial logs of failing executions may enforce a thread order that leads to a *non-failing* replay execution; (3) the combination of access vectors from partial logs of failing executions may enforce a thread order that leads to an *infeasible* replay execution; and (4) the number of different possible combinations of access vectors grows very quickly (in the worst case factorially) with the number of available partial logs, which renders blind brute force searches in such a vast space impractical.

To better understand challenges (1) and (2), consider the scenario in Figure 4, which depicts 3 user instance runs of the program in Figure 1. Figure 4 shows that all instances record partial logs corresponding to failing executions (instance *A* traces accesses to SPEs `threads` and `pos`, whereas both instances *B* and *C* record `countW` and `countR`). Despite that, a naive combination of partial logs may not be effective. For instance, as shown in the figure, merging partial logs *A* and *B* yields a complete log that enforces a successful execution, which does not reproduce the error. On the other hand, the log resulting from the combination of *A* and *C* would have triggered the failure. Therefore, a cooperative record and replay solution must be able to distinguish, in an efficient fashion, the combinations of partial logs that are likely to replay the bug from those that will be ineffective.

The following sections show how CoopREP addresses the challenges related to both partial log compatibility and vastness of the search space. First, Section 5 proposes 3 metrics to compute similarity between partial logs. Then, Section 6 describes 2 novel heuristics that exploit these metrics to generate a failure-inducing replay log in a small number of attempts.

5 | SIMILARITY METRICS

In CoopREP, the *similarity metrics* are used to quantify the amount of information that different partial logs may have in common. Similarity can hold values in the [0,1] range, and the rationale for the classification of the similarity between 2 partial logs is related to their number of SPEs with similar access vectors (ie, SPEs for which the logs have observed exactly the same or a very alike thread interleaving at runtime). Hence, the more SPEs with equal access vectors the partial logs have, the closer to 1 their similarity is.

CoopREP uses 3 metrics to measure partial log similarity: *Plain Similarity* (*PSIM*), *Dispersion-based Similarity* (*DSIM*), and *Dispersion Hamming Similarity* (*DHS*). Briefly, *PSIM* and *DSIM* differ essentially on the weight given to the SPEs of the program. On the other hand, *DSIM* and *DHS* weigh SPEs in the same way, but *DHS* uses a more accurate method than *DSIM* to compare the access vectors.

Using the formal notation presented in Table 1, the similarity metrics can then be defined as follows:

Plain Similarity. Let l_0 and l_1 be 2 partial logs; their *PSIM* is given by the following equation:

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

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. The first factor of the product computes the fraction of SPEs that have equal access vectors. The second factor *penalizes* the similarity value in case the partial logs have common SPEs with different access vectors (otherwise, $\#Diff_{l_0,l_1}$ will be 0 and the first factor will not be affected).

Note that *PSIM* is equal to 1 only when both logs are complete and identical, ie, when 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 ($l_0.s = l_1.s, \forall s \in \mathcal{S}$). This implies that, for any 2 partial logs, their *PSIM* will always be less than 1. However, the greater this value is, the more similar the 2 partial logs are.

Finally, it should be noted that functions $Equal_{l_0,l_1}$ and $Diff_{l_0,l_1}$ can be implemented very efficiently by comparing pairs of logs using the hashes of their access vectors.

Dispersion-based Similarity. In opposition to *PSIM*, *DSIM* does not assign the same weight to all SPEs. Instead, it takes into account whether a given SPE exhibits many different access vectors across the collected partial logs or not. This property is referred to as *dispersion*. The dispersion of an SPE depends, essentially, on the number of *concurrent and unsynchronized accesses to shared memory* existing in the target program, as well as on the number of threads. The reason is because the more concurrent and unsynchronized accesses and the more threads the program has, the larger the number of different thread interleavings there is. For instance, shared variables `pos`, `countW`, and `countR` in Figure 1 are subject to data races, as their accesses in both threads are not synchronized. As a consequence, their access vectors may easily contain different execution orders.

To capture the dispersion of a particular SPE, comparatively to the other SPEs of the program, CoopREP defines an additional metric denoted *overall-dispersion*. This metric corresponds to the ratio between the number of *different* access vectors logged for a given SPE and the whole universe of *different* access vectors collected for all SPEs (across all instances). More concretely, the overall-dispersion of an SPE *s* is computed as follows:

$$overallDisp(s) = \frac{\#\mathcal{AV}_s^*}{\#\mathcal{AV}^*}. \tag{2}$$

Note that the result of this metric varies within the range [0,1]. Also, note that *overallDisp* assigns larger dispersion to SPEs that tend to exhibit different access vectors across the collected partial logs. As an example, consider 2 SPEs—*x* and *y*—and 2 partial logs containing identical access vectors for SPE *x* (say x_1) and different access vectors for SPE *y* (say y_1 and y_2). For this case, $overallDisp(x) = \frac{1}{3}$ and $overallDisp(y) = \frac{2}{3}$.

TABLE 1 Notation used to define the similarity metrics

Notation	Description
$\mathcal{S}$	Set of all SPEs of the program.
$\mathcal{S}_l$	Subset of SPEs traced by partial log *l*.
$\mathcal{AV}$	Set of all access vectors recorded for all SPEs by all partial logs. ($\mathcal{AV}^*$ indicates the set of *different* access vectors recorded for all SPEs by all partial logs.)
$\mathcal{AV}_s$	Set of all access vectors recorded for SPE *s* by all partial logs. ($\mathcal{AV}_s^*$ indicates the set of *different* access vectors recorded for SPE *s* by all partial logs.)
$l.s$	Access vector recorded for SPE *s* by partial log *l*.
$Common_{l_0,l_1} = \{s \mid s \in \mathcal{S}_{l_0} \cap \mathcal{S}_{l_1}\}$	Intersection of SPEs recorded by both partial logs l_0 and l_1.
$Equal_{l_0,l_1} = \{s \mid s \in Common_{l_0,l_1} \wedge l_0.s = l_1.s\}$	Intersection of SPEs with identical access vectors recorded by both partial logs l_0 and l_1.
$Diff_{l_0,l_1} = \{s \mid s \in Common_{l_0,l_1} \wedge l_0.s \neq l_1.s\}$	Intersection of SPEs with different access vectors recorded by both partial logs l_0 and l_1.

Abbreviation: SPE, shared program element.

Now, let l_0 and l_1 be 2 partial logs. The *DSIM* between l_0 and l_1 is given by the following equation:

$$DSIM(l_0, l_1) = \sum_{x \in Equal_{l_0, l_1}} overallDisp(x) \times \left(1 - \sum_{y \in Diff_{l_0, l_1}} overallDisp(y)\right). \quad (3)$$

Using overall-dispersion as weight in *DSIM*, one can bias the selection towards pairs of logs having similarities in those SPEs that are often subject to different access interleavings. The rationale is that 2 partial logs having in common the same "rare" access vector, for a given SPE, are more likely to have been captured from compatible production runs.

Dispersion Hamming Similarity. Both *PSIM* and *DSIM* metrics rely on a binary comparison of access vectors, ie, the access vectors are either fully identical or not (see Equations 1 and (3)). However, it can happen that 2 access vectors have the majority of their values identical, differing only by a few positions. For instance, let us consider 3 access vectors: $A = [1, 1, 1, 1]$, $B = [1, 1, 1, 0]$, and $C = [0, 0, 0, 0]$. It is obvious that, despite being all different, *A* is more similar to *B* than to *C*.

To capture the extent of these *dissimilarities*, CoopREP uses another metric, called *DHS*. Let l_0 and l_1 be 2 partial logs; their *DHS* is given by the following equation:

$$DHS(l_0, l_1) = \sum_{s \in Common_{l_0, l_1}} hammingSimilarity(l_0.s, l_1.s) \times overallDisp(s), \quad (4)$$

where $Common_{l_0, l_1}$ is the set of SPEs recorded by both partial logs l_0 and l_1 (see Table 1), $hammingSimilarity(l_0.s, l_1.s)$ gives the value (normalized to range [0,1]) of the *Hamming similarity* of the access vectors logged for SPE *s* in partial logs l_0 and l_1, and $overallDisp(s)$ measures the relative weight of SPE *s* in terms of its overall-dispersion (see Equation (2)).

Hamming similarity is a variation of the *Hamming distance* [25] used in several disciplines, such as telecommunication, cryptography, and information theory. Hamming distance is used to compute distance between strings and can be defined as follows: Given 2 strings s_1 and s_2, their Hamming distance is the number of positions that differ in the corresponding symbols. In other words, it measures the minimum number of substitutions needed to transform s_1 into s_2.

As previously referred, CoopREP applies Hamming similarity to access vectors instead of strings.[¶] As such, it might happen that 2 access vectors being compared have different sizes. CoopREP addresses this issue by augmenting the smaller vector with a suffix of *N* synthesized thread IDs (different from the ones in the larger vector), where *N* is equal to the size difference of the 2 access vectors.

Trade-offs among the metrics. This section compares the 3 partial log similarity metrics in terms of their time complexity and accuracy.[‖] Greater accuracy can be useful when the overall-dispersion weight values are not well balanced between the SPEs. This happens when some SPEs have identical access vectors in almost every partial log, while other SPEs exhibit access vectors that are equal (or very alike) only in a small subset of partial logs. As such, the ability to accurately identify those small subsets of similar partial logs becomes crucial to generate a feasible replay log.

Let *S* be the number of SPEs in a log and *L* the maximum length of an access vector recorded. Beginning with *PSIM*, this metric has to compare the hashes of all overlapping SPEs between the 2 partial logs. As such, it has a time complexity of $O(S^2)$, due to the time required to find the set of overlapping SPEs between the 2 logs. In terms of accuracy, one may argue that *PSIM* provides poor accuracy, since it assumes that every SPE has the same importance. Consequently, *PSIM* can be considered to be a simple and baseline metric to capture partial log similarity.

The *DSIM* also executes in $O(S^2)$ time, because, like *PSIM*, it solely compares hashes.[**] However, *DSIM* provides more accuracy than *PSIM*, as it assigns different weights to SPEs.

Finally, *DHS* is expected to further improve the accuracy provided by *DSIM*, by allowing to compare access vectors at a finer granularity, and not just by means of a binary classification (ie, equal or different). Unfortunately, this increase in accuracy comes at the cost of a larger execution time, because *DHS* requires comparing each position of the access vectors instead of just comparing their hashes, as done in *PSIM* and *DSIM*. As such, *DHS* executes with $O(S^2 \cdot L)$ complexity.

In sum, each one of the 3 metrics represent a particular trade-off between computation time and accuracy. Section 8.2 provides an empirical comparison of the similarity metrics to further assess the benefits and limitations of each one.

6 | HEURISTICS TO MERGE PARTIAL LOGS

As mentioned in Section 4.3, testing all possible combinations of partial logs to find one capable of successfully reproducing the nondeterministic bug is impractical. To cope with this issue, CoopREP leverages 2 novel heuristics, denoted *Similarity-guided Merge (SGM)* and *Dispersion-guided Merge (DGM)*. The goal of the heuristics is to optimize the search among all possible combinations of partial logs (from failed executions) and guide it towards combinations that result in an effective (ie, bug-inducing) replay log.

[¶] Additional metrics based on *edit distance* would be potentially usable, but, for this work, we opted for metrics with more efficient computation.

[‖] This paper considers accuracy to be related to the degree of granularity when measuring similarity between access vectors. For example, measuring similarity at the level of the whole access vector (ie, by comparing their hashes) is more coarse-grained than measuring similarity at the level of each position of the access vector. Therefore, the former is less accurate than the latter.

[**] Since the overall-dispersion of the SPEs is only computed once (at the beginning of the statistical analysis), its computation time is being disregarded in the complexity analysis.

Both heuristics operate by systematically picking a partial log to serve as basis to reconstruct a complete trace of the failing execution. To find this *base partial log*, the heuristics rank the partial logs in terms of *relevance*. As such, the heuristics differ essentially in the strategy used to compute and sort the partial logs according to their relevance.

The *SGM* considers the most relevant partial logs to be the ones that maximize the chances of being completed with information from other similar partial logs. The rationale is that if there is a group of partial logs with high similarity among themselves, then it should probably mean that they were captured from compatible production runs.

On the other hand, *DGM* focuses on finding partial logs whose SPEs account for higher overall-dispersion. The rationale is that, by ensuring the compatibility of the *most disperse* SPEs (because they were traced from the same production run), *DGM* maximizes the probability of completing the missing SPEs with access vectors that tend to be common across the partial logs. The downside of *DGM* is that its most relevant partial logs are less likely to have other potential similar partial logs to be combined with. Exactly because the dispersion of the SPEs in relevant partial log is high, the access vectors for these SPEs are rarely occurring in other logs and, therefore, are not good matching points. This can hinder the effectiveness of the heuristic when the capacity of the partial logs is not sufficient to encompass all the SPEs with high overall-dispersion.

The following sections further describe *SGM* and *DGM* by detailing their 2 operation phases: *computation of relevance* and *generation of the next candidate replay log*. Next, Section 6.4 discusses the advantages and drawbacks of each heuristic. Finally, Section 7 presents the general algorithm used by the statistical analyzer, which can leverage any of the 2 heuristics to successfully replay concurrency bugs.

6.1 | Similarity-guided Merge

Computation of relevance. The *SGM* classifies a partial log as being relevant if it is considered similar to many other partial logs. More formally, this corresponds to computing, for each partial log *l*, the following value of *Relevance*:

$$Relevance(l) = \frac{\sum_{l' \in kNN_l} Similarity(l, l')}{\#kNN_l}, \tag{5}$$

where kNN_l is a set containing *l*'s *k-nearest neighbors* (in terms of similarity) that suffice to fill *l*'s missing SPEs and $Similarity(l, l')$ is one of the 3 possible similarity metrics (*PSIM*, *DSIM*, or *DHS*). Note that kNN_l contains, at most, one unique partial log (the most similar) per missing SPE in *l*, which means that $\#kNN_l$ will always be less or equal than the number of missing SPEs in *l*. Also, if several partial logs have the same similarity with respect to *l* and fill the same missing SPE, kNN_l will pick one of them at random.

Unfortunately, to be able to compute *Relevance*, *SGM* needs to measure the similarity between every pair of partial logs. This happens because, for any partial log *l*, the composition of set kNN_l is only accurately known after measuring the similarity between *l* and all the remaining partial logs. The whole process to compute partial logs' relevance in *SGM* is presented in Algorithm 1.

For a given base partial log *baseLog*, *SGM* starts by computing the set of *k*-nearest neighbors *KNN* (line 5). Then, *SGM* fills the missing SPEs in *baseLog* with the information recorded by the partial logs in *KNN*, while calculating the corresponding relevance of the base partial log (lines 6-11). Note that the relevance of *baseLog* corresponds to the average similarity between *baseLog* and the partial logs contained in *KNN* (line 11). Finally, *SGM* stores the newly complete *baseLog* in *MostRelevant*, which is a set containing the complete logs sorted in descending order of their relevance (line 12).

Generation of the next candidate replay log. The generation of the next candidate replay log in *SGM* is straightforward. Since, during the computation of the relevance values, *SGM* already combines partial logs and generates complete execution traces (line 9 in Algorithm 1), it just needs to iteratively pick replay logs from the set of most relevant logs. Algorithm 2 illustrates this procedure.

6.2 | Dispersion-guided Merge

Computation of relevance. The *DGM* classifies a partial log as being relevant if it contains SPEs with high overall-dispersion. The *Relevance* of a partial log *l* in *DGM* is computed as follows:

$$Relevance(l) = \sum_{s \in S_l} overallDisp(s). \tag{6}$$

By identifying partial logs that have traced the *most disperse* SPEs, *DGM* attempts to decrease the chances of completing the missing SPEs with noncompatible access vectors. Algorithm 3 shows how *DGM* computes relevance for a set of partial logs.

The *DGM* computes the relevance of a given base partial log *baseLog* by summing the overall-dispersion values of the SPEs recorded by *baseLog* (lines 2-6). The *DGM* then stores the base partial log in the sorted set *MostRelevant* according to the computed relevance (line 7).

Algorithm 1: sgm.computeRelevance

```
Input: PLogs: set with partial logs collected during production runs.
Input: simMetric: metric used to measure similarity between partial logs.
Output: MostRelevant: sorted set containing complete logs in descending order of relevance.

MostRelevant = $\emptyset$
for baseLog $\in$ PLogs do
    simSum = 0
    simLogs = 0
    KNN $\leftarrow$ simMetric.computeKNN(baseLog)
    while KNN.hasNext() do
        nLog $\leftarrow$ KNN.next()
        simSum += simMetric.measureSimilarity(baseLog, nLog)
        baseLog.fillMissingSPEs(nLog)
    end
    relevance = simSum/KNN.size()
    MostRelevant.addBasedOnRelevance(baseLog, relevance)
end
return MostRelevant
```

Algorithm 2: sgm.genNextReplayLog

```
Input: MostRelevant: sorted set containing complete logs in descending order of relevance.
Output: replayLog: a complete execution trace ready to be replayed.

replayLog $\leftarrow$ MostRelevant.next()
return replayLog
```

Algorithm 3: dgm.computeRelevance

```
Input: PLogs: set with partial logs collected during production runs.
Output: MostRelevant: set containing partial logs sorted in descending order of their relevance.

MostRelevant = $\emptyset$
for baseLog $\in$ PLogs do
    dispSum = 0
    for spe $\in$ baseLog.getSPEs() do
        dispSum += spe.getOverallDisp()
    end
    MostRelevant.addBasedOnRelevance(baseLog,dispSum)
end
return MostRelevant
```

Algorithm 4: dgm.genNextReplayLog

```
Input: MostRelevant: set containing partial logs sorted in descending order of their relevance.
Input: simMetric: metric used to measure similarity between partial logs.
Output: replayLog: a complete execution trace ready to be replayed.

replayLog $\leftarrow$ MostRelevant.next()
KNN $\leftarrow$ simMetric.computeKNN(replayLog)
while !replayLog.isComplete() $\wedge$ KNN.hasNext() do
    nLog $\leftarrow$ KNN.next()
    replayLog.fillMissingSPEs(nLog)
end
return replayLog
```

Generation of the next candidate replay log. Algorithm 4 describes how *DGM* generates new replay logs. In opposition to *SGM*, *DGM* does not combine partial logs when computing their relevance. As a consequence, after picking the base partial log to build the next candidate replay trace, *DGM* has still to identify its *k*-nearest neighbors (line 2). Once the set of *k*-nearest neighbors *KNN* is computed, *DGM* then fills the missing SPEs in the base partial log with the information recorded by the logs in *KNN* (lines 3-6).

As final remark, note that the similarity metrics are orthogonal to the heuristics. For instance, *DGM*, although using overall-dispersion to calculate the relevance values, can use any similarity metric (*PSIM*, *DSIM*, or *DHS*) to compute the set of nearest neighbors for the base partial log.

6.3 | Example

To better illustrate how the 2 heuristics operate, let us see an example. On the left side of Figure 5, there are 6 failing partial logs collected from user instances running the program in Figure 1. Notice that each partial log recorded only 2 of the 4 SPEs of the program (traced SPEs are depicted in black and the observed but nonrecorded SPEs are represented in gray). The subscripted number in each SPE identifier indicates the hash of its respective access vector (eg, $\texttt{pos}_1 \neq \texttt{pos}_2$).

As shown in the center of Figure 5, there are 8 different access vectors in total: $\texttt{threads}_1$, $\texttt{pos}_1$, $\texttt{pos}_2$, $\texttt{pos}_3$, $\texttt{countW}_1$, $\texttt{countW}_2$, $\texttt{countR}_1$, and $\texttt{countR}_2$. Therefore, one obtains the following values of dispersion for the 4 SPEs: *overallDisp*(`threads`) = 1/8, *overallDisp*(`pos`) = 3/8, *overallDisp*(`countW`) = 2/8, and *overallDisp*(`countR`) = 2/8.

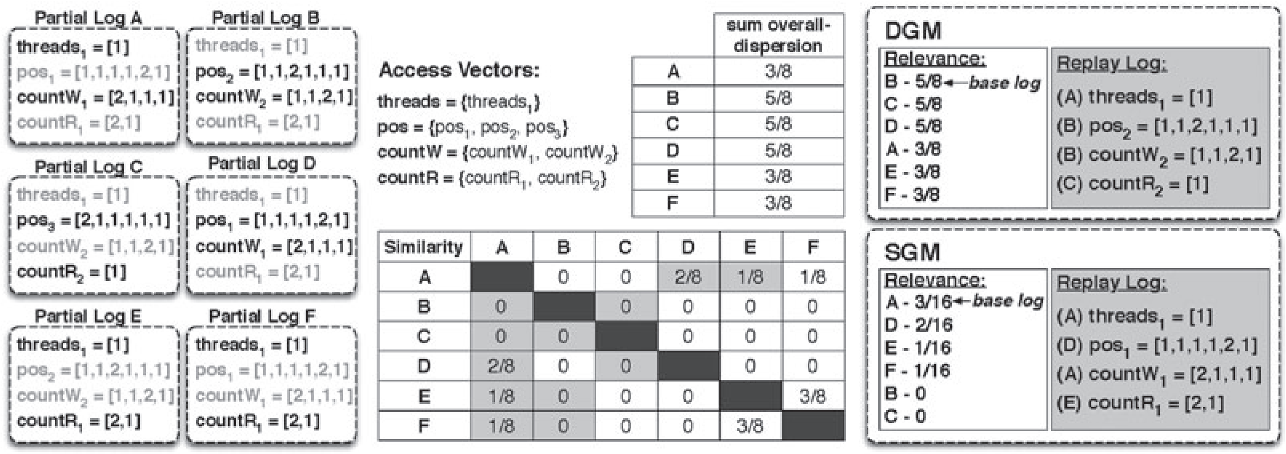
Assuming *DSIM* as similarity metric (see Equation 3), the following describes how *SGM* and *DGM* operate to produce a bug-inducing replay log using the partial logs depicted in Figure 5.

SGM. This heuristic first calculates the similarity between the partial logs (see the similarity matrix at the center-bottom of Figure 5). For example, both logs *A* and *D* traced the same access vector $\texttt{countW}_1$ for their single overlapping SPE `countW`; hence, *DSIM(A,D)* = *DSIM(D,A)* = 2/8.

Next, *SGM* computes the *k*-nearest neighbors for all partial logs and ranks them in terms of relevance. The *k*-nearest neighbors for each log are marked with shaded cells in the similarity matrix at the center-bottom of Figure 5. For instance, the nearest neighbors for partial log *A* are partial logs D and *E*, because they exhibit the highest similarity to *A* and suffice to fill its missing SPEs.

Recall from Section 6.1 that the most relevant log in *SGM* is considered to be the one with highest average similarity to the partial logs in the nearest neighbors set. The box at the bottom-right of Figure 5 shows the logs sorted in descending order of their relevance for *SGM*. As shown in the figure, log *A* is the most relevant log (note that *Relevance(A)* = (2/8 + 1/8)/2 = 3/16) and, therefore, is chosen as the base partial log. The first replay log for *SGM* will, thus, be composed by partial log *A* augmented with access vector $\texttt{pos}_1$ from *D* and $\texttt{countR}_1$ from *E*.

DGM. This heuristic considers the relevance of partial logs to be the sum of their SPEs' overall-dispersion. The table at the center-top of Figure 5 shows the sum of the SPE overall-dispersion for the partial logs. Since there are several partial logs with the highest value of relevance (namely, *B*, *C*, and *D*), one is picked at random. Let us follow the alphabetical order and pick *B* as the base log. Given that *B* does not have any particular similar partial log (the similarity between *B* and every other partial log is 0, as reported in the similarity matrix in Figure 5), the heuristics fills its missing SPEs with other partial logs randomly picked. For this case, let us assume that *B* is filled with data from *A* and *C*. Thereby, the replay log for *DGM* will be composed by partial log *B* augmented with access vector $\texttt{threads}_1$ from *A* and $\texttt{countR}_2$ from *C*.



Partial Log A
- $\text{threads}_1 = [1]$
- $\text{pos}_1 = [1,1,1,1,2,1]$
- $\text{countW}_1 = [2,1,1,1]$
- $\text{countR}_1 = [2,1]$

Partial Log B
- $\text{threads}_1 = [1]$
- $\text{pos}_2 = [1,1,2,1,1,1]$
- $\text{countW}_2 = [1,1,2,1]$
- $\text{countR}_1 = [2,1]$

Partial Log C
- $\text{threads}_1 = [1]$
- $\text{pos}_3 = [2,1,1,1,1,1]$
- $\text{countW}_2 = [1,1,2,1]$
- $\text{countR}_2 = [1]$

Partial Log D
- $\text{threads}_1 = [1]$
- $\text{pos}_1 = [1,1,1,1,2,1]$
- $\text{countW}_1 = [2,1,1,1]$
- $\text{countR}_1 = [2,1]$

Partial Log E
- $\text{threads}_1 = [1]$
- $\text{pos}_2 = [1,1,2,1,1,1]$
- $\text{countW}_2 = [1,1,2,1]$
- $\text{countR}_1 = [2,1]$

Partial Log F
- $\text{threads}_1 = [1]$
- $\text{pos}_1 = [1,1,1,1,2,1]$
- $\text{countW}_1 = [2,1,1,1]$
- $\text{countR}_1 = [2,1]$

Access Vectors:
- threads = $\{\text{threads}_1\}$
- pos = $\{\text{pos}_1, \text{pos}_2, \text{pos}_3\}$
- countW = $\{\text{countW}_1, \text{countW}_2\}$
- countR = $\{\text{countR}_1, \text{countR}_2\}$

	sum overall-dispersion
A	3/8
B	5/8
C	5/8
D	5/8
E	3/8
F	3/8

Similarity	A	B	C	D	E	F
A	■	0	0	2/8	1/8	1/8
B	0	■	0	0	0	0
C	0	0	■	0	0	0
D	2/8	0	0	■	0	0
E	1/8	0	0	0	■	3/8
F	1/8	0	0	0	3/8	■

DGM

Relevance:	Replay Log:
B - 5/8 ← *base log*	(A) $\text{threads}_1 = [1]$
C - 5/8	(B) $\text{pos}_2 = [1,1,2,1,1,1]$
D - 5/8	(B) $\text{countW}_2 = [1,1,2,1]$
A - 3/8	(C) $\text{countR}_2 = [1]$
E - 3/8	
F - 3/8	

SGM

Relevance:	Replay Log:
A - 3/16 ← *base log*	(A) $\text{threads}_1 = [1]$
D - 2/16	(D) $\text{pos}_1 = [1,1,1,1,2,1]$
E - 1/16	(A) $\text{countW}_1 = [2,1,1,1]$
F - 1/16	(E) $\text{countR}_1 = [2,1]$
B - 0	
C - 0	

FIGURE 5 Example of the *modus operandi* of the 2 heuristics (Similarity-guided Merge [*SGM*] and Dispersion-guided Merge [*DGM*]), using Dispersion-based Similarity as similarity metric. The gray lines in the partial logs indicate the access vectors that were observed during the original execution but not traced

6.4 | Trade-offs among the heuristics

This section analyzes the complexity of the heuristics, in terms of the time they take to compute the relevance of partial logs and generate a replay log.

Recall from Section 5 that measuring the similarity between 2 partial logs requires either S^2 steps or $S^2 \cdot L$ steps depending on the similarity metric used (where S is equal to the number of SPEs in a logs and L indicates the maximum number of entries of the access vectors being compared). For the sake of simplicity, let us denote the complexity of computing the similarity between 2 partial logs by *simMetric(m)*, where $m \in \{PSIM, DSIM, DHS\}$ is one of the 3 similarity metrics presented in Section 5. Also, let N be the number of partial logs collected across all user instances.

To compute the relevance of a given partial log, the *SGM* heuristic needs to calculate the similarity between that log and the $n-1$ remaining logs, which has complexity $O(N \cdot simMetric(m))$. Consequently, computing the relevance for all partial logs will have time complexity $O(N^2 \cdot simMetric(m))$. Furthermore, inserting a log in the sorted set with the most relevant logs requires $log(N)$ steps. Therefore, the overall complexity of *SGM* using similarity metric m for sorting partial logs according to their relevance is $O(N^2 \cdot simMetric(m) \cdot log(N))$.

Regarding the generation of a new replay log, *SGM* incurs $O(1)$ complexity as it simply needs to return the next most relevant log from the sorted set, which had already been computed in the previous phase (see Algorithm 1).

On the other hand, the *DGM* heuristic computes each partial log's relevance by simply summing the overall-dispersion of its SPEs, which requires S steps. Considering all partial logs, this procedure is executed N times. Therefore, the complexity of *DGM* to compute the set of most relevant partial logs is $O(N \cdot S \cdot log(n))$. In turn, when generating a replay log, *DGM* has also to measure the similarity between the base partial log and all the other logs. This requires $n-1$ comparisons, which results in an overall complexity of $O(N^2 \cdot simMetric(m))$ in the worst case scenario.

The 2 heuristics are now compared in terms of their benefits and limitations in replaying concurrency bugs via partial log combination.

The *SGM* has the advantage of quickly identifying *clusters* of very similar partial logs. This is useful when the majority of partial logs exhibits SPEs with very different access vectors. In this scenario, finding a couple of partial logs with identical (or very alike) access vectors means that there is a high probability that their combination yields a feasible and effective replay log. The drawback of this heuristic is its additional complexity to rank the partial logs in terms of relevance.

The *DGM*, on the other hand, has the key benefit of quickly computing the relevance of partial logs. Moreover, it is also very effective when the number of SPEs traced by each partial log is sufficient to encompass all SPEs with very high dispersion. This means that, for a given base partial log, the remaining nonrecorded SPEs should exhibit very common access vectors. Hence, the replay log can be trivially produced by simply merging the base log with any other partial log (the example for *DGM* in Figure 5 illustrates this scenario). However, when the number of high-disperse SPEs is greater than the size of the partial log, *DGM* is not as effective as *SGM* because the partial logs with higher SPE overall-dispersion tend to have few similar partial logs (eg, partial log *B* in Figure 5 does not have similar partial logs at all).

7 | STATISTICAL ANALYZER EXECUTION MODE

This section describes how the metrics and the heuristics fit together in CoopREP's execution flow. Algorithm 5 depicts the pseudo-code of the complete algorithm used by the *statistical analyzer* (see Section 4.1) to systematically combine partial logs to find a replay log capable of reproducing the bug. The algorithm receives a similarity metric and a heuristic as input and starts with the candidate generator computing the relevance and ranking the partial logs according to this value (line 2). Next, the candidate generator builds a replay log (line 6), following the strategies described in Sections 6.1 and 6.2, respectively, for *SGM* and *DGM* heuristics.

At the end of this process, the replay oracle enforces the schedule defined by the candidate replay log and verifies whether the bug is reproduced or not (line 7). If it is, the goal has been achieved and the algorithm ends. If it is not, the candidate generator produces a new replay log and sends it to the replay oracle for a new attempt. This procedure is repeated while the bug has not been replayed (ie, *hasBug* = *false*) and the candidate generator has more replay logs to be attempted.

CoopREP also supports an optional *brute force mode* as fallback to the scenario in which all the replay logs generated by the heuristic fail to trigger the bug. Here, all possible access vectors collected by the partial logs are tested for each missing SPE in the base partial log (lines 10-12), until the error is triggered or the maximum number of attempts to reproduce the bug is reached (ie, *replayOracle*.reachedMaxAttempts() = *true*). According to our experiments though, the brute force mode does not provide additional value to the heuristics in terms of bug replay ability. Given the disappointing results of this procedure, it will not be considered in the remaining of the paper.

Corner Case: Partial logs with 1SPE. A special case in the cooperative record and replay approach occurs when the developer defines recording profiles that result in partial logs tracing information for a single SPE. This particular scenario, although minimizing the runtime overhead, hampers the goal of combining access vectors from potentially compatible partial logs. Since partial logs with 1SPE either do not overlap at all or overlap in the single SPE, it becomes pointless to apply the similarity metrics and the heuristics. CoopREP addresses this issue with an additional metric, called *BugCorrelation*, which gives the correlation between the bug and each access vector individually. This metric is an adaptation of the scoring method proposed by Liblit et al [20] and, in addition to failing executions, considers information from successful executions.

Concretely, *BugCorrelation* ranks access vectors based on the harmonic mean of their *Sensitivity* and *Specificity*. In other words, an access vector is correlated with the bug if it appears on many failed runs and only few successful runs. Let F_{total} be the total number of partial logs, resulting from failed executions, that have traced a given SPE s. For an access vector v, let also $F(v)$ be the number of failed partial logs that have recorded v for s and $S(v)$ be the number of successful partial logs that have recorded v for s. The 3 indicators are then calculated as follows:

$$\mathit{Sensitivity}(v) = \frac{F(v)}{F_{total}} \tag{7}$$

$$\mathit{Specificity}(v) = \frac{F(v)}{S(v) + F(v)} \tag{8}$$

$$\mathit{BugCorrelation}(v) = \frac{2}{\frac{1}{\mathit{Sensitivity}(v)} + \frac{1}{\mathit{Specificity}(v)}}. \tag{9}$$

In summary, the higher the *BugCorrelation* value, the more correlated with the bug the access vector is. Therefore, to generate a complete replay log, the candidate generator computes the statistical indicators for each SPE and builds a complete log by merging the access vectors that exhibit higher *BugCorrelation*. If there is more than one access vector with the highest value of *BugCorrelation* for a given SPE, the several possible combinations are tested. However, if the bug does not manifest after these attempts, the candidate generator may still fallback to the *brute force* mode, where it tests all possible combinations of access vectors as in Algorithm 5.

Algorithm 5: Statistical analyzer's algorithm to produce a complete replay log via partial log combination.

```
Input: PLogs: set with partial logs collected from production runs.
Input: heuristic: heuristic to merge partial logs (it can be either sgm or dgm)
Input: simMetric: metric used to measure similarity between partial logs.
Output: replayLog: a complete execution trace ready to be replayed.

1  // Compute relevance
2  MostRelevant ← candidateGen.computeRelevance(PLogs, heuristic, simMetric)
3  // Generate the next candidate replay log and attempt to replay the bug
4  hasBug = false
5  while ! hasBug ∧ candidateGen.hasNextReplayLog() do
6      replayLog ← candidateGen.genNextReplayLog(MostRelevant, heuristic, simMetric)
7      hasBug ← replayOracle.replay(replayLog)
8  end
9  // Optional – Brute Force attempts
10 while ! hasBug ∧ ! replayOracle.reachedMaxAttempts() do
11     for all partial logs, test all possible combinations of access vectors to fill the missing SPEs and check whether the
       bug is reproduced
12 end
13 return replayLog
```

The CCI [21] also applies similar indicators for cooperative statistical debugging of concurrency bugs. However, conversely to CoopREP, CCI focuses on error diagnosis rather the record and replay. For that reason, CCI only samples low-overhead predicates at runtime (eg, whether 2 consecutive accesses to a shared variable x are by the same thread), which can be useful to capture common types of concurrency bugs, but do not suffice to replay them.

8 | EVALUATION

The main goal of CoopREP is to reproduce concurrency bugs with less overhead than previous approaches, using cooperative logging and partial log combination. To this end, CoopREP's experimental evaluation focuses on answering the following questions:

1. Which similarity metric provides the best trade-offs among accuracy, execution time, and scalability? (Section 8.2)
2. Which CoopREP heuristic replays the bug in a smaller number of attempts? How do the heuristics compare to previous state-of-the-art deterministic replay systems? (Section 8.3)
3. How does CoopREP's effectiveness vary with the number of logs collected? (Section 8.4)
4. How much runtime overhead reduction can CoopREP achieve with respect to other classic, noncooperative logging schemes? (Section 8.5)
5. How much can CoopREP decrease the size of the logs generated? (Section 8.6)

8.1 | Experimental setting

A prototype of CoopREP was implemented in Java, using Soot[††] to instrument the target applications. The prototype also contains 2 other state-of-the-art record and replay systems implemented on top of CoopREP for comparison with their respective original noncooperative versions: (1) LEAP,[6] which uses a classic order-based strategy to record information, and (2) PRES, [1] a search-based system. Low-level instrumentation and optimizations settings are identical for all solutions, to guarantee comparison fairness. However, it should be noted that, as PRES' code is not publicly available, its mechanism was implemented as faithfully as possible, according to the details given in Park et al. [1]

The PRES shares CoopREP's idea of minimizing the recording overhead during production runs at the cost of an increase in the number of attempts to replay the bug during diagnosis. Unlike CoopREP though, PRES does not combine logs from multiple user instances. PRES first traces a *sketch* of the original execution and, then, performs an offline guided search to explore different combinations of (nonrecorded) thread interleavings. To cope with the very large dimension of the space of possible execution schedules, PRES leverages feedback produced from each failed attempt to increase the chances of finding the bug-inducing ordering in the subsequent one.

The authors of PRES have explored 5 different sketching techniques that imply different trade-offs between recording overhead and reproducibility. Starting from a baseline logging profile that traces only input, signals, and thread scheduling, the authors performed experiments with different sketches that incrementally record more information, namely, the global order of synchronization operations, system calls, functions, basic blocks, and shared-memory operations, respectively. This work uses only the PRES-BB recording scheme (which logs the global order of basic blocks containing shared accesses), as it provides a good trade-off between effectiveness and overhead according to the results reported in Park et al. [1] An evaluation of CoopREP with a slightly more relaxed scheme, namely, PRES-SYNC (which only traces the global order of synchronization operations), was also considered. However, since the benchmarks used in the experiments have just a few synchronization variables (0 to 3), applying a cooperative logging technique to this scheme would not provide meaningful insights.

The PRES-BB logs the order in which threads access basic blocks containing operations on shared variables into a single vector. To allow for cooperative logging, the implementation of this recording scheme in CoopREP treats each *shared basic block* as an SPE, which means that a distinct access vector is traced for each shared basic block. In other words, CoopREP$_P$ tracks a partial order that reflects how threads access each shared basic block; this is in contrast to the original PRES-BB approach, which, instead, maintains, in a centralized log, the total order with which *any* thread accesses *any* shared basic block. Unlike PRES-BB, hence, CoopREP$_P$ does not ensure that the order with which threads accessed *different* shared basic blocks during the original execution is preserved during the replay phase. This design choice has implications on the breadth of the search space to be explored by CoopREP$_P$ and PRES-BB, which are evaluated in Section 8.3.

Regarding the size of CoopREP's partial logging profiles, the percentage of the total SPEs logged in each run was varied according to 3 different configurations—25%, 50%, and 75%. In addition, the experiments considered the extreme case where partial logs only have one SPE. For each configuration, CoopREP was configured to collect 500 partial logs from different failing executions, plus 100 additional partial logs from successful runs (used to compute the statistical indicators for the 1SPE case). To get a fairer comparison between the recording configurations, the partial logs were generated from complete logs, randomly picking the SPEs to be stored according to the configuration's percentage. The maximum number of attempts of the heuristics to reproduce the bug was set to 500. Note that, following the 500 tries, one may resort to the brute force approach. Since the outcomes of the experiments showed that brute force partial log combination brings no added value to the heuristics, the paper solely reports results for a threshold of 500 replay attempts (see Section 8.3).

As a final remark, the following sections assume the order-based version of CoopREP (ie, using LEAP on top of CoopREP) for the experiments, unless mentioned otherwise.

All the experiments were conducted in a machine Intel Core 2 Duo at 3.06 GHz, with 4 GB of RAM and running Mac OS X. However, the logs were evenly recorded from 4 different machines: 3 of them with 4 GB of RAM, running Mac OS X, but with processors Intel Core 2 Duo at 2.26, 2.66, and 3.06 GHz, respectively; and the last one equipped with an 8 core processor AMD FX 8350 at 4 GHz, 16 GB of RAM, running Ubuntu 13.04.

8.2 | Similarity metrics comparison

The goal of the first question is to evaluate how effective CoopREP is in replaying concurrency bugs and assess which partial log similarity metric—*PSIM*, *DSIM*, or *DHS*—exhibits the best trade-off between accuracy and execution time. To this end, the experiments are performed on several programs from the IBM ConTest benchmark suite [26] which contain various types of concurrency bugs.[‡‡] Table 2 describes these programs in terms of number of SPEs, lines of code (LOC), total number of shared accesses, failure rate, number of threads (along with the program input that indicates the concurrency level used in the test), and the bug pattern according to Farchi et al. [26]

Accuracy. These experiments are interested in evaluating the impact of the metrics' accuracy on the effectiveness of CoopREP. Table 3 reports the number of attempts required by the *SGM* heuristic, using the 3 metrics, to replay the bug for the configurations previously mentioned (25%, 50%, and 75%). The results for *DGM* heuristic are omitted because they show similar trends.

[††] http://www.sable.mcgill.ca/soot/

[‡‡] The evaluation is restricted to ConTest programs that have at least 4 SPEs.

TABLE 2 ConTest benchmark programs

Program	LOC	#SPE	#Total accesses	#Threads (*Program input*)	Failure rate, %	Bug description
BoundedBuffer	536	15	525	6 *(2)*	1	notify instead of notifyAll
BubbleSort	362	11	52495	3 *(lot)*	2	atomicity violation
BufferWriter	255	6	130597	11 *(lot)*	35	wrong or no-lock
Deadlock	95	4	12	3 *(-)*	12	deadlock
Manager	236	5	2368	3 *(3 15)*	28	atomicity violation
Piper	280	8	580	21 *(10)*	6	missing condition for wait
ProducerConsumer	281	8	576	7 *(little)*	15	orphaned thread
TicketOrder	246	9	6662	11 *(lot)*	2	atomicity violation
TwoStage	123	5	27103	31 *(15 15)*	1	two-stage

Abbreviations: SPE, shared program element; LOC, lines of code.

TABLE 3 Number of attempts required by the *SGM* heuristic, using the 3 similarity metrics, to replay the ConTest benchmark bugs

Program	25%			50%			75%		
	PSIM	*DSIM*	*DHS*	*PSIM*	*DSIM*	*DHS*	*PSIM*	*DSIM*	*DHS*
BoundedBuffer	233	1	1	1	1	1	1	1	1
BubbleSort	482	482	482	372	372	372	256	256	256
BufferWriter	*X*	*X*	*X*	*X*	*X*	*X*	17	13	13
Deadlock	—	—	—	1	1	1	1	1	1
Manager	—	—	—	*X*	*X*	*X*	*X*	*X*	*X*
Piper	*X*	*X*	*X*	133	1	1	1	1	1
ProducerConsumer	*X*	*X*	*X*	287	19	19	6	6	6
TicketOrder	*X*	*X*	*X*	93	43	43	47	25	25
TwoStage	—	—	—	466	15	15	62	10	10

Abbreviations: DHS, Dispersion Hamming Similarity; DSIM, Dispersion-based Similarity; PSIM, Plain Similarity; SGM, Similarity-guided Merge. The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts.

The "—" indicates that the corresponding configuration resulted in a 1SPE case and, therefore, it is not applicable for metric comparison.

Table 3 shows that, apart from program Manager, all the 3 metrics allow the heuristic to replay the bugs for at least one recording scheme. Despite that, *PSIM* is generally less effective than *DSIM* and *DHS*, as shown by the results for Piper, ProducerConsumer, TicketOrder, and TwoStage (especially for the 50% configuration). These results confirm the insight about the positive effect of taking into account SPE dispersion when measuring partial log similarity.

Another interesting observation is concerned with the variation of the bug replay ability of the *SGM* heuristic across the benchmarks, regardless of the similarity metric. For instance, even when partial logs contain 75% of the total number of SPEs, the bug in program BubbleSort was only reproduced at the 256th attempt. On the other hand, the error in BoundedBuffer was replayed almost always at the first attempt. This is due to the dispersion of the SPEs that, if very high, heavily hampers the combination of compatible partial logs. Section 8.3 goes deeper into how SPE dispersion affects the bug replay ability.

Finally, note that the number of attempts to replay the bug increases with the decrease of the number of SPEs per partial log, as expected.

Execution time. To understand which metric provides the best trade-off between accuracy and execution time, the time required by the heuristic to compute the relevance of all 500 collected partial logs was also measured. The outcomes of these experiments are presented in Table 4. Recall that, for *SGM*, calculating the relevance requires computing the similarity between every pair of partial logs. Hence, the results in Table 4 also account for these computations.

The results demonstrate that *DHS* exhibits, as expected, the worst performance, being up to 5.2× slower than *PSIM* for TwoStage (when logging 75% of the SPEs). In turn, *DSIM* typically achieves almost the same execution time as *PSIM*, which can be explained by the fact that the computation of overall-dispersion for each SPE is performed only once, at the beginning of the analysis. For this reason, *DSIM is considered to be the most cost-effective metric to measure similarity between partial logs.*

Table 4 also shows that computation time is greatly affected by the number of shared accesses in the program (which are indicated in Table 2). For instance, BufferWriter is simultaneously the program with the highest number of accesses (>130K) and highest computation time (almost 1 h for the 75% scheme using *DHS*). However, it should be noted that the values reported in Table 4 encompass the time required to load the logs into memory, create the objects and data structures used by CoopREP, and compute the similarity between the partial logs. Given that the task of computing

TABLE 4 Execution time (in seconds) to compute the relevance of 500 logs in *SGM*, for each similarity metric

	25%			50%			75%		
Program	***PSIM***	***DSIM***	***DHS***	***PSIM***	***DSIM***	***DHS***	***PSIM***	***DSIM***	***DHS***
BoundedBuffer	2.24	2.27	3.45	2.54	2.6	3.79	2.62	2.77	4.04
BubbleSort	31.78	32.70	53.59	64.9	69.16	155.99	88.84	92.43	252.19
BufferWriter	584.68	604.41	702.56	1371.95	1605.48	1937.79	1516.41	2082.03	3532.38
Deadlock	—	—	—	1.85	1.89	2.02	2.40	2.47	2.52
Manager	—	—	—	4.87	4.75	20.44	8.85	9.31	30.17
Piper	2.04	2.53	3.26	2.74	3.11	5.98	2.95	3.52	9.79
ProducerConsumer	2.1	2.24	3.79	2.18	2.53	4.84	2.45	2.73	8.33
TicketOrder	5.99	6.64	12.57	10.16	10.68	28.6	11.77	11.86	47.60
TwoStage	—	—	—	7.50	7.45	31.08	8.64	9.11	44.93

Abbreviations: DHS, Dispersion Hamming Similarity; DSIM, Dispersion-based Similarity; PSIM, Plain Similarity; SGM, Similarity-guided Merge.

TABLE 5 Time (in seconds) required by the 3 metrics to measure similarity between 2 partial logs

	Execution time, s			
Log size	**Bootstrapping**	***PSIM***	***DSIM***	***DHS***
1 KB	0.031	0.001	0.001	0.002
10 KB	0.139	0.001	0.001	0.010
100 KB	0.477	0.003	0.004	0.014
1 MB	1.712	0.004	0.007	0.032
10 MB	17.037	0.005	0.008	0.489
100 MB	119.413	0.016	0.022	4.121
1 GB	1202.054	0.024	0.032	40.078

Abbreviations: DHS, Dispersion Hamming Similarity; DSIM, Dispersion-based Similarity; PSIM, Plain Similarity.

similarity can be easily performed in parallel or even in the cloud (which does not happen in our current prototype), the following section presents some additional experiments to further assess the computational cost and the scalability of the 3 metrics.

Scalability. To assess the scalability of the metrics, additional experiments were performed to measure the amount of time required to compare 2 partial logs using each one of the 3 similarity metrics. Table 5 reports these results for log sizes ranging from 1 KB to 1 GB, as well as the corresponding bootstrapping time (ie, the time required to load the logs into memory and create the data structures used by CoopREP). The logs were obtained using a micro-benchmark, which allows controlling the size of the resulting log size by tuning the number of shared accesses. The experiments were conducted in a machine Intel Core 2 Duo at 3.06 GHz, with 4 GB of RAM and running Mac OS X.

The results in Table 5 show that the great majority of the execution time in CoopREP's statistical analysis is devoted to bootstrapping, which increases linearly with the size of the logs being analyzed. In turn, the cost to measure similarity using either *PSIM* or *DSIM* metrics is practically unaffected by the size of the logs being compared, because these metrics use hashes of the access vectors to compute the similarity value. Conversely, *DHS* needs to compare every single position of the access vector; thus, its computation time is heavily affected by the log size.

In summary, the results in Table 5 provide further evidence that *DSIM* is the similarity metric that exhibits better trade-off between effectiveness and efficiency. For instance, for a scenario of 500 logs of 1 GB and *DSIM* as similarity metric, the heuristic would take around 2 hours to determine the logs to use. This is because the search for the best partial logs to use has a quadratic cost in the number of logs, and measuring similarity between any 2 logs takes around 32 milliseconds for a size of 1 GB. Since each computation is independent, the task of measuring similarity between all partial logs can be easily parallelized, thus allowing to significantly improve CoopREP's efficiency. Nevertheless, one can argue that a latency of a few hours before replaying the bug is, in practice, acceptable in the software maintenance domain.

8.3 | Heuristic comparison

This section assesses the bug replay ability of the 2 merging heuristics with respect to state-of-the-art order-based and search-based techniques. CoopREP$_L$ denotes the version of CoopREP that applies cooperative record and replay to LEAP [6] and CoopREP$_P$ the version of CoopREP that implements PRES's approach.

To a smaller extent, the experiments are also intended at comparing the *SGM* heuristic against the *DGM* heuristic to assess which one replays bugs in fewer attempts. Finally, the replay ability of the *1SPE* case, where CoopREP uses statistical indicators to merge partial logs that have traced only one SPE, is evaluated.

TABLE 6 Real-world concurrency bugs used in the experiments

Program	LOC	#SPE	#Total accesses	#Threads (*Program input*)	Failure rate, %	Bug description
Cache4j	2.3K	4	129	4 *(-)*	15	atomicity violation
Derby#2861	1.5M	14	2509	11 *(10 10)*	5	atomicity violation
Tomcat#37458	535K	22	89	3 *(-)*	16	atomicity violation
Weblech	35K	5	54	6 *(-)*	1	atomicity violation

Abbreviation: SPE, shared program element.

TABLE 7 Number of replay attempts required by CoopREP$_L$ with 1SPE, *SGM*, and *DGM* (both heuristics using *DSIM* metric)

		SGM + DSIM			*DGM + DSIM*		
Program	1SPE	25%	50%	75%	25%	50%	75%
BoundedBuffer	*X*	▨ 1	1	1	3	1	1
BubbleSort	*X*	482	372	256	▨ 1	1	1
BufferWriter	*X*	*X*	*X*	▨ 13	*X*	*X*	97
Deadlock	▨ 1	—	1	1	—	1	1
Manager	*X*	—	*X*	*X*	—	*X*	*X*
Piper	*X*	*X*	▨ 1	1	*X*	32	1
ProducerConsumer	*X*	*X*	19	7	*X*	▨ 1	1
TicketOrder	*X*	*X*	43	25	*X*	▨ 9	1
TwoStage	*X*	—	15	10	—	▨ 1	1
Tomcat#37458	▨ 1	1	1	1	1	1	1
Derby#2861	*X*	*X*	*X*	59	*X*	*X*	▨ 1
Weblech	▨ 1	—	1	1	—	1	1
Cache4j	▨ 1	—	1	1	—	1	1

Abbreviation: DGM, Dispersion-guided Merge; DSIM, Dispersion-based Similarity; SGM, Similarity-guided Merge; SPE, shared program element.

Note. The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts. The "—" indicates that the corresponding configuration resulted in a 1SPE case and, therefore, it is not applicable to the heuristics. Shaded cells highlight the CoopREP$_L$ scheme with best trade-off between effectiveness and amount of information recorded.

Besides the programs in the ConTest benchmark suite (see Table 2), the experiments of this section include 4 concurrency bugs from real-world applications, which are also used in previous work [6,27-29]. These applications are described in Table 6 in terms of number of SPEs, lines of code, number of shared accesses, number of threads, failure rate, and bug pattern. Cache4j is a fast thread-safe implementation of a cache for Java objects; Derby is a widely used open-source Java Relational Database Management System from Apache (bug no. 2861 from Derby's bug repository was used for the tests); Tomcat is a widely used JavaServer Pages and servlet container (bug no. 37458 from Tomcat's bug repository was used for the tests); and Weblech is a multithreaded web site download and mirror tool. Notice that, albeit these applications have thousands of lines of codes and many shared variables, Table 6 reports solely the SPEs experienced by each application's bug test driver. Similarly to previous works, [6,27-29] test drivers were used to facilitate the manifestation of the failure. Despite that, some bugs are still very hard to experience (eg, Weblech).

CoopREP$_L$ vs LEAP (order-based). Table 7 reports the number of attempts, required by each heuristic, to replay the benchmark bugs. For each benchmark, Table 7 highlights the most cost-effective CoopREP$_L$ scheme, defined as the one that reproduces the bug successfully with the least logging. As LEAP is a pure order-based system, it replays all bugs at the first attempt.

An overall analysis of Table 7 shows that CoopREP$_L$ is able to replay all bugs except Manager. Moreover, in 11 of the 12 successful cases, there is at least one recording scheme for which the bug was reproduced at the first attempt, which proves the efficacy of the heuristics in combining compatible partial logs.

Comparing now *SGM* and *DGM* against each other, Table 7 shows that *DGM* tends to be more effective than *SGM*, especially when partial logs record a higher percentage of SPEs. To better understand why this happens, let us observe the benchmarks' SPE *individual-dispersion* depicted in Figure 6.[§§] Here, individual-dispersion corresponds to the ratio between the number of *different* access vectors logged for a given SPE and the

[§§]For the sake of readability and to ease the comparison, Figure 6 presents only the individual-dispersion values for the full-logging configuration.

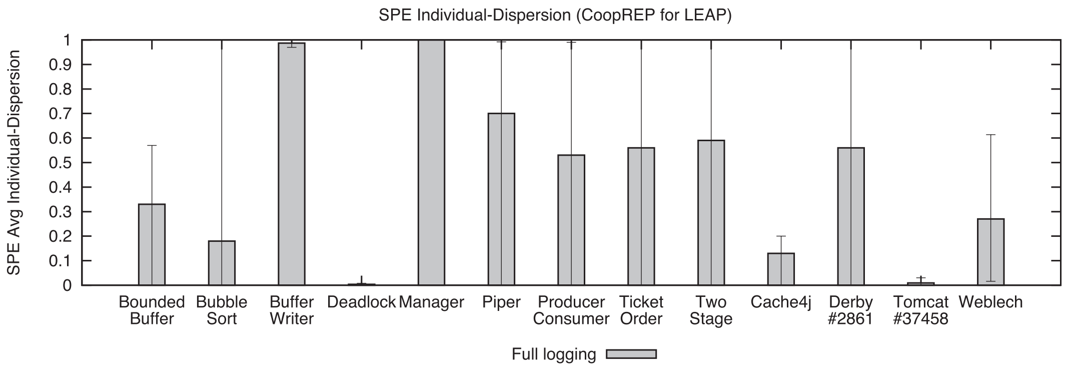


FIGURE 6 Average shared program element (SPE) individual-dispersion value for the benchmarks, with a full LEAP logging scheme. The error bars indicate the minimum and maximum values observed

total number of access vectors collected for that SPE (across all instances).[¶¶] It is computed as follows:

$$individualDisp(s) = \frac{\#\mathcal{AV}_s^*}{\#\mathcal{AV}_s}. \tag{10}$$

The first conclusion that can be drawn from the figure is that programs with higher average SPE individual-dispersion—BufferWriter and Manager—are also the ones for which replaying the bug was harder. In particular, the logs collected for Manager contained different access vectors for all SPEs of the program, which clearly represents unfavorable conditions for a partial log combination approach (in fact, none of the heuristics was able to reproduce the bug, as indicated in Table 7). On the other hand, programs with very low SPE dispersion (such as Deadlock and Tomcat) were easily replayed by both heuristics, as well as when partial logs contained only a single SPE (see the "1SPE" column in Table 7). This confirms the insight that CoopREP's bug replay ability depends on programs' SPE dispersion.

Figure 6 also shows that the SPE individual-dispersion is highly variant in most test cases (Deadlock, Cache4j, and Tomcat are the exceptions). This means that these programs have, simultaneously, SPEs for which the partial logs have recorded always the same access vector and SPEs for which the partial logs have traced (almost) always different access vectors. Under these circumstances, *DGM* was clearly more effective than *SGM*, because it was able to pick, as the most relevant partial log, the one that had traced all the most disperse SPEs. This fact is particularly evident for programs BubbleSort, TicketOrder, and Derby, for the 75% recording scheme. These results support the claim that tracing all SPEs with very high dispersion in the same partial log increases the likelihood of filling the remaining nonrecorded SPEs with very common (and, thus, compatible) access vectors.

However, *DGM* did not always outperform *SGM*: For the BufferWriter and Piper programs, *SGM* exhibited better effectiveness. This can be explained by the fact that, for these cases, the capacity of the partial logs was not sufficient to encompass all SPEs with high dispersion. As a consequence, *DGM* ended up having several partial logs with the same relevance value and simply picked one at random for each attempt (moreover, the majority of these most relevant logs did not have any similarity with the remaining partial logs). In opposition, *SGM* was able to quickly identify subsets of similar partial logs and, thus, generate a feasible replay log in fewer attempts.

CoopREP$_P$ vs PRES (search-based). To further assess the benefits and limitations of cooperative logging, the current prototype of CoopREP also implements a version of the approach proposed by PRES, a state-of-the-art search-based record and replay system for C/C++ programs.[‖‖] This version is denoted CoopREP$_P$. As mentioned in Section 8.1, the original PRES solution maintains a single log that totally orders accesses of threads to different shared basic blocks. In order to allow for cooperative logging, the recording scheme was changed to treat each shared basic block as an SPE. During the replay phase, CoopREP$_P$ first generates a complete log using the heuristics and then applies PRES's original replay scheme. This means that CoopREP$_P$ traces a distinct access vector for each shared basic block, yielding a partial order that allows *different* shared basic blocks for being accessed in different orders during the original and replay phase.

Table 8 compares CoopREP$_P$'s and PRES's post-recording exploration techniques against each other in terms of the number of replay attempts performed by each technique.

Similarly to the previous section, CoopREP$_P$ exhibits very good effectiveness. In particular, CoopREP$_P$ was able to replay all bugs with the same number of attempts as PRES-BB, apart from program Manager (though achieving an average reduction of log size of 4.2×, as it shall be discussed in Section 8.6). Besides Manager, CoopREP$_P$ was also not able to reproduce the bug in Derby, but this error was not replayed by PRES-BB either. These experimental results suggest that the choice of tracking solely the partial order of access to individual shared basic blocks (in contrast to PRES-BB's, which totally orders these events) does not impair the ability to replay bugs, at least for the cases considered in this study.

Comparing now the 2 heuristics against each other, Table 8 shows once again that DGM tends to achieve better results than SGM (the former performed equally to or better than the latter in 11 of the 13 cases).

[¶¶]Note that individual-dispersion differs from overall-dispersion (see Equation (2)) because it accounts for different access vectors recorded for a particular SPE, rather than for the whole set of SPEs.
[‖‖]A version of PRES was implemented in Java for the experiments.

TABLE 8 Number of replay attempts required by PRES-BB, CoopREP$_P$ with *1SPE*, *SGM*, and *DGM* (both heuristics using *DSIM* metric)

Program	#SPEs	PRES-BB	1SPE	*SGM + DSIM*			*DGM + DSIM*		
				25%	50%	75%	25%	50%	75%
BoundedBuffer	22	1	1	1	1	1	1	1	1
BubbleSort	14	1	1	1	1	1	1	1	1
BufferWriter	9	1	*X*	*X*	4	4	10	2	1
Deadlock	3	1	1	—	1	1	—	1	1
Manager	13	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Piper	12	1	*X*	4	1	1	6	3	1
ProducerConsumer	20	1	*X*	1	1	1	6	2	1
TicketOrder	14	2	*X*	*X*	316	218	*X*	6	2
TwoStage	9	1	*X*	*X*	*X*	44	*X*	*X*	1
Tomcat#37458	32	3	*X*	*X*	*X*	6	*X*	*X*	3
Derby#2861	15	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Weblech	7	1	*X*	1	1	1	1	1	1
Cache4j	6	1	*X*	1	1	1	1	1	1

Abbreviation: DGM, Dispersion-guided Merge; DSIM, Dispersion-based Similarity; SGM, Similarity-guided Merge; SPE, shared program element. Column *#SPEs* indicates the number of basic blocks that contain accesses to shared variables.

Note. The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts. The "—" indicates that the corresponding configuration resulted in a 1SPE case and, therefore, it is not applicable to the heuristics. Shaded cells indicate the CoopREP$_P$ scheme with best trade-off between effectiveness and amount of information recorded.

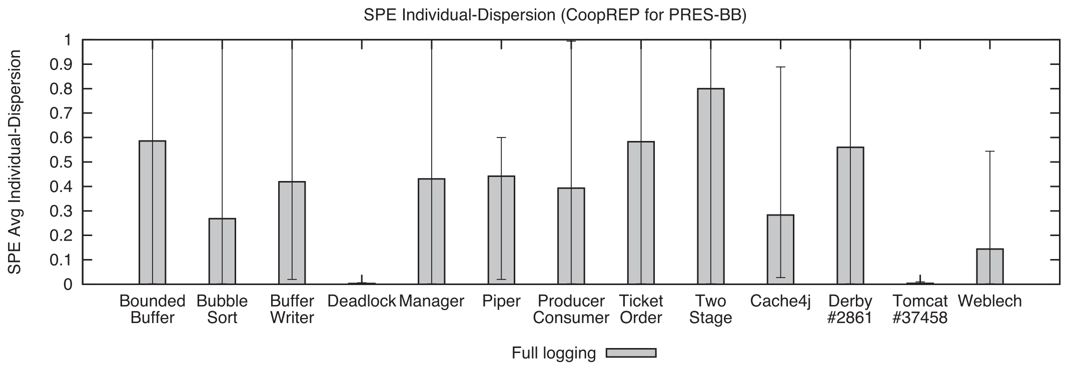


FIGURE 7 Average shared program element (SPE) individual-dispersion value for the benchmarks, with a full PRES-BB logging scheme. The error bars indicate the minimum and maximum values observed

Another interesting observation is that CoopREP$_P$ required fewer replay attempts than CoopREP$_L$ for some test cases, namely, BufferWriter, BubbleSort, and Piper. Observing the SPE individual-dispersion of PRES-BB for these programs (depicted in Figure 7), it is possible to see that the corresponding values are smaller than those of Figure 6. This fact is likely the result of considering SPEs as basic blocks with shared accesses instead of shared variables. Here, access vectors that previously encompassed all accesses to a given shared variable now become scattered across different basic blocks. As a consequence, if a given basic block contains only accesses to a single shared variable, it will have less accesses for PRES-BB than its corresponding shared variable for LEAP and a smaller individual-dispersion.

However, the opposite scenario is also possible to occur. If a basic block encompasses accesses to multiple shared variables, its resulting SPE individual-dispersion is expected to increase. For instance, this occurs with programs TwoStage and TicketOrder, which exhibited higher individual-dispersion in Figure 7 and required more attempts to replay the bug in Table 8 than in Table 7.

Partial log combination vs bug correlation. Comparing the heuristics for partial log combination against the approach of tracing only one SPE per partial log and using the statistical indicators to measure correlation between the error and the access vectors (see Section 7), it is possible to conclude that the former approach is clearly more effective for most cases. In fact, the *1SPE* approach was rarely able to replay the bug in our experiments, apart from the programs with very low average SPE individual-dispersion, as indicated by Tables 7 and 8. Nevertheless, for this kind of programs, *1SPE* becomes an appealing approach to adopt, as it provides the smallest runtime overhead (see Section 8.5).

8.4 | Variation of effectiveness with the number of logs collected

The third research question regards the impact of varying the number of logs collected on the number of attempts required to replay the bug.

The benchmarks were selected to obtain a heterogeneous test-bed, so to be representative of a broad range of programs with diverse characteristics. In fact, the considered set of benchmarks includes a low-failure rate concurrency bug affecting a large real-world software artifact (Weblech), as well as 3 synthetic bugs from the ConTest suite. Further, the benchmarks have diverse average individual-dispersion values, which, as already discussed, represent a key factor for the effectiveness of CoopREP.

For each one of these programs, both *SGM* and *DGM* were run with a number N of (randomly chosen) partial logs, where $N \in \{16, 32, 64, 128, 256, 512\}$. Once again, the percentage of SPEs logged was varied as in the previous section.

Table 9 reports the outcomes of the experiments. As expected, the results show that bugs of programs with low average individual-dispersion (such as Deadlock or Weblech) can be easily reproduced even when gathering a small number of logs. On the other hand, as the average individual-dispersion increases, the reproducibility of the program through partial logging schemes becomes much more challenging, thus requiring a greater number of partial logs to deterministically trigger the bug. Program Piper reflects this scenario well, as it was only possible to replay the bug with less than 75% of the total SPEs when logging 512 partial logs.

The results for Piper highlight another interesting observation: An increase in the number of partial logs collected does not always imply a decrease in the number of attempts needed to reproduce the error (see column 75% of Piper for *SGM* in Table 9). This is because, for programs with higher average individual-dispersion, having more partial logs also increases the chances of making "wrong choices." For instance, *SGM* took more attempts to find a bug-inducing replay log with 128 partial logs than with 64, basically because there were more partial logs considered relevant that ended up not being compatible with the partial logs in the k-nearest neighbor set.

Finally, note that these results provide additional evidence to support the claim that *DGM* is more effective than *SGM* when the partial logs trace a larger number of SPEs (see Piper for the 75% scheme). Once again, this is because *DGM* is able to quickly identify the partial logs that have traced all the most disperse SPEs.

8.5 | Runtime overhead

This section analyzes the performance benefits achievable via CoopREP's approach with respect to noncooperative logging schemes, such as LEAP and PRES-BB. To this end, CoopREP$_L$'s and CoopREP$_P$'s recording time were compared respectively against that of LEAP's and PRES-BB's for the benchmark programs. For each test case, the execution time was measured by computing the arithmetic mean of 5 runs. For CoopREP schemes, the average value was computed for 3 different configurations of logging profiles (each profile configuration with 5 runs, as well). **CoopREP$_L$ vs LEAP.** Figure 8 reports the runtime overhead on the programs of Table 7. Native execution times range from 0.01 s (for BubbleSort) to 1.2 s (for Weblech).

From the figure, it is possible to see that CoopREP$_L$ always achieves lower runtime degradation than LEAP. However, the results also highlight that overhead reductions are not necessarily linear. This is because some SPEs are accessed much more frequently than others. Given that the instrumentation of the code is performed statically, the load in terms of thread accesses may not be equally distributed among the users, as previously referred in Section 4.2.

TABLE 9 Number of attempts required by *SGM* and *DGM* to replay the bugs in programs Deadlock, BoundedBuffer, Piper, and Weblech when varying the number of partial logs collected

	#Logs	Deadlock			BoundedBuffer			Piper			Weblech		
		25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%
SGM	16	—	1	1	*X*	*X*	1	*X*	*X*	*X*	—	9	1
	32	—	1	1	*X*	1	1	*X*	*X*	32	—	1	1
	64	—	1	1	*X*	1	1	*X*	*X*	6	—	1	1
	128	—	1	1	3	1	1	*X*	*X*	15	—	1	1
	256	—	1	1	1	1	1	*X*	*X*	1	—	1	1
	512	—	1	1	1	1	1	*X*	1	1	—	1	1
DGM	16	—	1	1	*X*	*X*	1	*X*	*X*	*X*	—	6	1
	32	—	1	1	*X*	1	1	*X*	*X*	1	—	3	1
	64	—	1	1	*X*	1	1	*X*	*X*	1	—	1	1
	128	—	1	1	34	1	1	*X*	*X*	1	—	1	1
	256	—	1	1	6	1	1	*X*	*X*	1	—	1	1
	512	—	1	1	3	1	1	*X*	32	1	—	1	1

Abbreviation: DGM, Dispersion-guided Merge; SGM, Similarity-guided Merge.The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts.

Note. The "—" indicates that the corresponding configuration resulted in a 1SPE case and, therefore, it is not applicable to the heuristics.

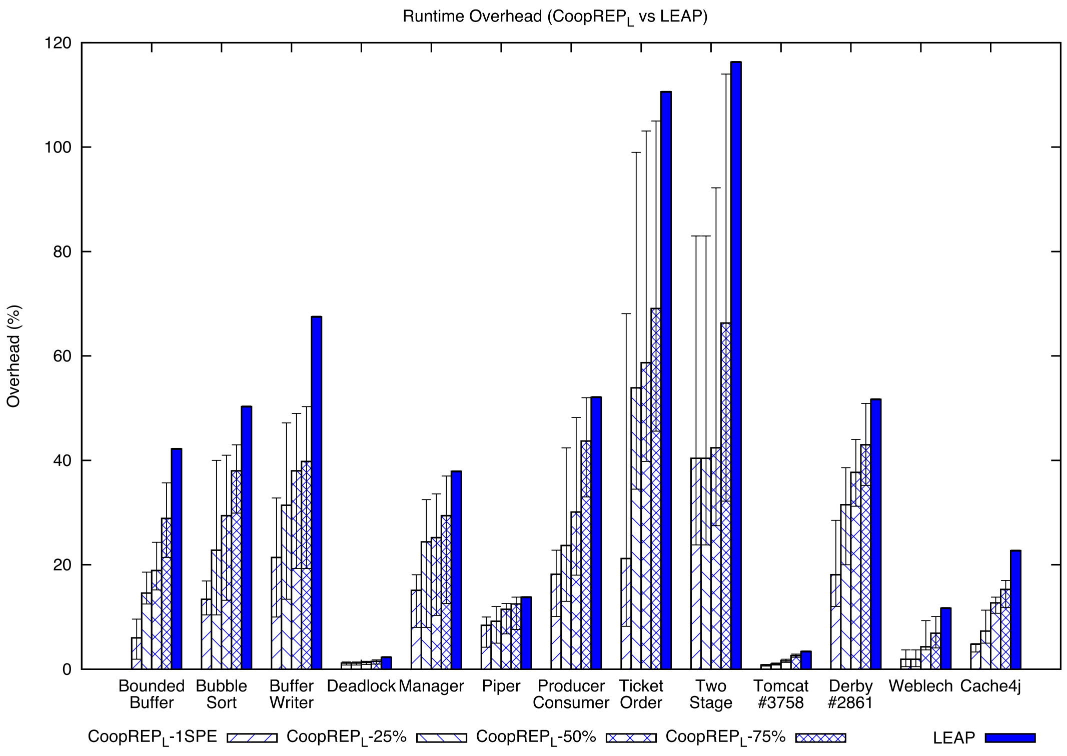


FIGURE 8 Runtime overhead (in %) of CoopREP$_L$ and LEAP, for the benchmark programs. The error bars indicate the minimum and maximum values observed

Figure 8 also shows that it is typically more beneficial to use partial logging in scenarios where the use of full logging has a higher negative impact on performance. The most notorious cases are BufferWriter, TicketOrder, and TwoStage, which are also among the applications with the highest number of SPE accesses (see Table 2). For example, in TwoStage, LEAP imposes a performance overhead of 116% on average, while CoopREP$_L$ incurs only 42% overhead on average when recording half the SPEs (which is sufficient to successfully replay the bug).

CoopREP$_P$ vs PRES-BB. Figure 9 plots the recording overhead for PRES-BB and CoopREP$_P$. Once again, CoopREP$_P$ exhibits always smaller runtime slowdown than PRES-BB, although the reductions are not linear with the decrease in the partial logging percentage. Similarly to LEAP, the benchmarks for which the benefits of CoopREP$_P$ are more prominent are those that incur higher recording overhead, namely, BubbleSort, BufferWriter, TicketOrder, and TwoStage. In particular, for BubbleSort, PRES-BB imposes an overhead of 276%, whereas CoopREP$_P$-1SPE incurs only 21% overhead on average and still reproduces the error. **Summary.** To easily analyze the trade-off between effectiveness and recording overhead for the different approaches, this section matches the most cost-effective scheme for each system (as reported in Tables 7 and 8) to the corresponding average runtime penalty. The outcomes are reported in Table 10. Recall that LEAP is able to replay the bug at the first attempt for all test cases. For the cases where either PRES-BB or CoopREP was not able to reproduce the bug, the table simply reports the highest average recording overhead observed.

As shown by Table 10, CoopREP reduces the average recording overhead of PRES-BB and LEAP by 2.4× and 2× on average, respectively, while still being effective in replaying the failure for all benchmarks except for Manager and Derby (in CoopREP$_P$). In light of these results, one can advocate the benefits of the cooperative record and replay approach with respect to other state-of-the-art approaches.

8.6 | Log size

This section quantifies the benefits achievable by CoopREP with respect to LEAP and PRES in terms of space overhead. To measure CoopREP's space overhead, the average size of 10 random partial logs was computed for each of the partial logging schemes.

CoopREP$_L$ vs LEAP. Figure 10 reports the sizes of the logs generated by LEAP and CoopREP$_L$ for the different benchmarks at each user instance. Unsurprisingly, the space overhead follows a trend that is analogous to that observed for the performance overhead: Logs generated by CoopREP$_L$ are always smaller than those produced by LEAP, although it is possible to observe a significant variance in the log size for some programs. For instance, programs such as BubbleSort and BufferWriter have simultaneously partial logs with much smaller and very similar size to those of LEAP. As in the performance overhead assessment, this variance is due to the heterogeneity in the frequency of accesses to SPEs. Figure 11 depicts this phenomenon, by plotting the average and variance of the number of SPE accesses for each benchmark.

From the analysis of Figure 11, it is possible to see that programs with high variance in the log size correspond to those that also have high variance in the number of accesses per SPE. For instance, BubbleSort has an SPE with only 2 accesses and another with more than 49 700 accesses.

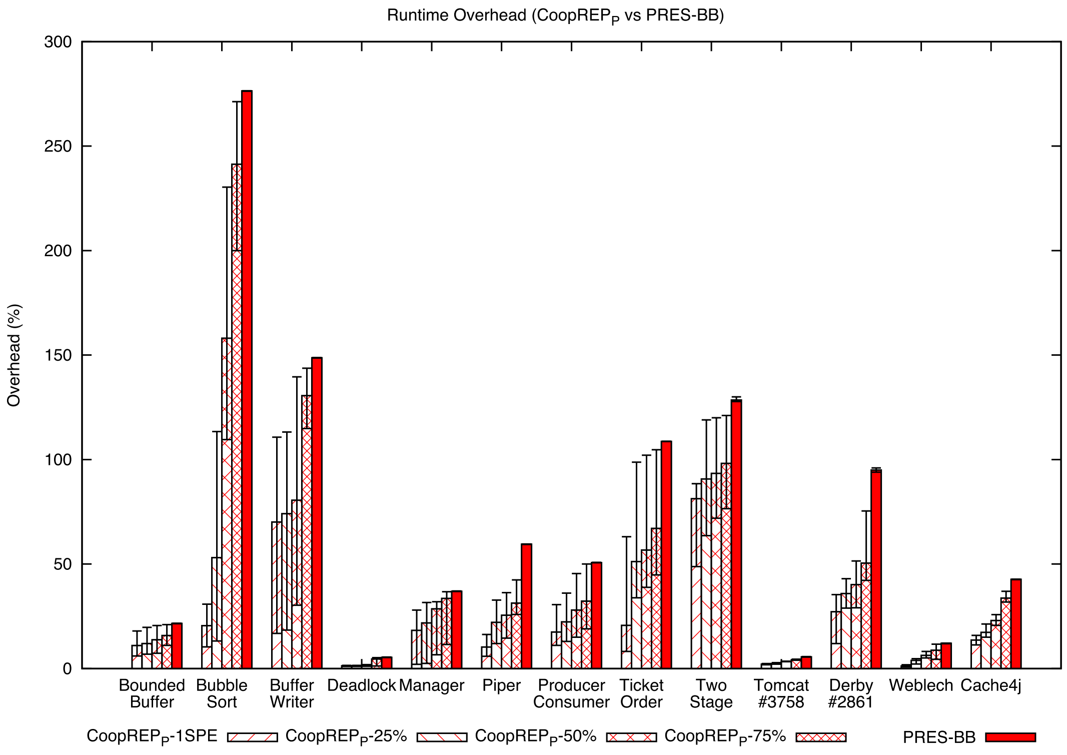


FIGURE 9 Runtime overhead (in %) of CoopREP$_P$ and PRES-BB, for the benchmark programs of Table 8. The error bars indicate the minimum and maximum values observed

TABLE 10 Average runtime overhead incurred by the most effective CoopREP schemes for both PRES-BB and LEAP (according to Tables 7 and 8)

Program	PRES-BB, %	Best of CoopREP$_P$, %	LEAP, %	Best of CoopREP$_L$, %
BoundedBuffer	22	*(1SPE)* 11	42	*(SGM-25%)* 15
BubbleSort	276	*(1SPE)* 21	50	*(DGM-25%)* 23
BufferWriter	149	*(DGM-25%)* 74	68	*(SGM-75%)* 40
Deadlock	5	*(1SPE)* 1	2	*(1SPE)* 1
Manager	37	*(X)* 34	38	*(X)* 29
Piper	60	*(SGM-25%)* 22	14	*(SGM-50%)* 12
ProducerConsumer	51	*(SGM-25%)* 22	52	*(DGM-50%)* 30
TicketOrder	109	*(DGM-50%)* 57	111	*(DGM-50%)* 59
TwoStage	128	*(DGM-75%)* 98	116	*(DGM-50%)* 42
Tomcat#37458	6	*(DGM-75%)* 4	12	*(1SPE)* 2
Derby#2861	95	*(X)* 51	52	*(DGM-75%)* 43
Weblech	12	*(DGM-25%)* 4	12	*(1SPE)* 2
Cache4j	43	*(DGM-25%)* 17	23	*(1SPE)* 5
AVERAGE	*76*	*32*	*45*	*23*

CoopREP$_P$ vs PRES-BB. Figure 12 depicts the log sizes for PRES-BB and CoopREP$_P$. The results are similar to those observed in Figure 10; ie, programs with greater disparity in SPE access frequency exhibit higher variance in partial log sizes. Also, for the programs containing more shared accesses, the benefits of partial logging are more clear. For example, in BubbleSort, a complete PRES-BB log has 1.2 MB, whereas the average log size for CoopREP$_P$-1SPE has 63 KB on average (and 535 KB at most), which suffices to replay the bug.

Summary. Table 11 presents, for each benchmark, the log size reduction achievable by CoopREP's most cost-effective scheme with respect to PRES-BB and LEAP. The results in the table also support the claim about CoopREP being more cost-effective than full-logging solutions. In particular, CoopREP produced logs that are, on average, 4.2× and 3.2× smaller than those generated by PRES-BB and LEAP, respectively.

Finally, it is worth highlighting that if one used LEAP and PRES only on a single (or a few) user instance(s), and not on the whole users' population, the global space overhead incurred by these solutions would be lower than CoopREP's. However, this would come at the cost of reducing the

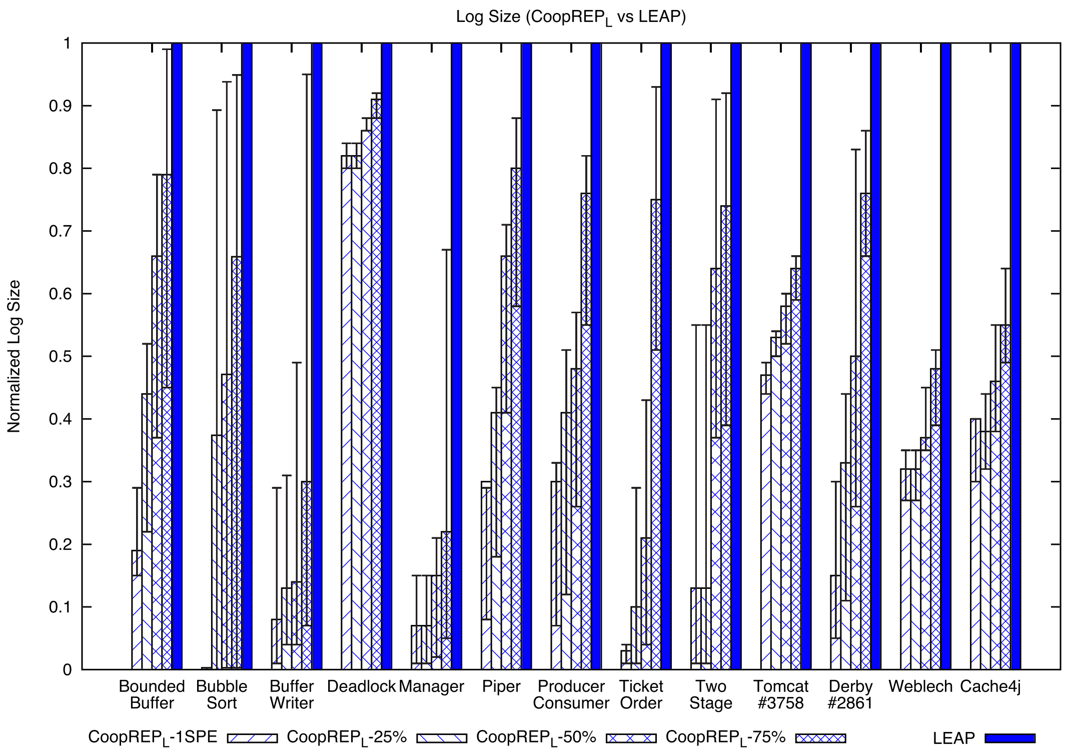


FIGURE 10 Log size reduction achieved by the various CoopREP$_L$ partial logging schemes with respect to LEAP. The error bars indicate the minimum and maximum values observed

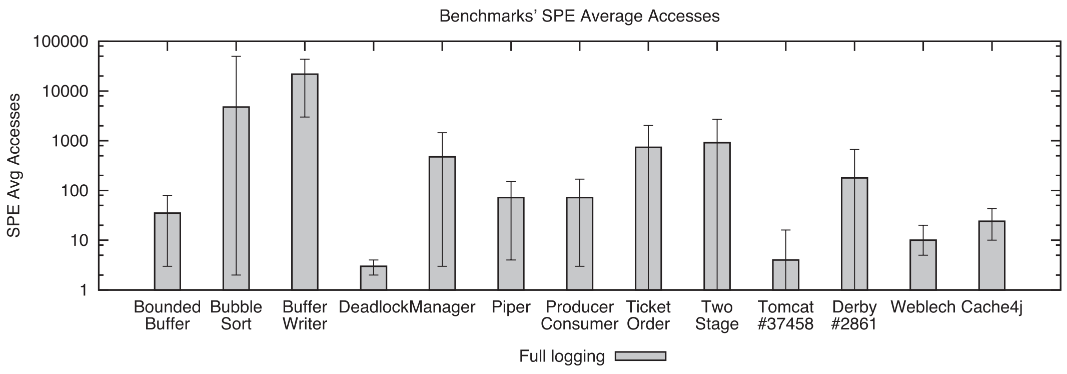


FIGURE 11 Variance in shared program element (SPE) accesses. The error bars indicate the minimum and maximum values observed

likelihood of capturing buggy executions and, indeed, it is common practice to enable software monitoring tools on large scale populations of users precisely for this reason[30].

8.7 | Discussion

This section summarizes the findings of the experiments by answering the research questions that motivated this evaluation study.

1. **Which similarity metric—*PSIM*, *DSIM*, or *DHS*—provides the best trade-off in terms of accuracy, execution time, and scalability?** The experiments presented in Section 8.2 reveal that *DSIM* is, at least for the considered set of benchmarks, the most cost-effective metric to measure similarity between partial logs. Comparing to *PSIM*, *DSIM* allowed CoopREP to reproduce the bug in the same or in a smaller number of attempts for all 9 test cases, thus providing more accuracy for a similar execution time.

   Comparing to *DHS*, albeit *DSIM* has shown the same bug replay ability, it required much smaller execution time to compute the similarity of a pair of partial logs. In fact, *DSIM* was up to 1252× faster than *DHS* to measure similarity for 1 GB logs.
2. **Which CoopREP heuristic replays the bug in a smaller number of attempts? How do the heuristics compare to previous state-of-the-art deterministic replay systems?** Section 8.3 evaluates CoopREP's effectiveness by providing a 3-fold comparison. The first experiment compared the effectiveness of the 2 CoopREP heuristics: *SGM* and *DGM*. Next, CoopREP was compared against LEAP and PRES-BB in terms of bug

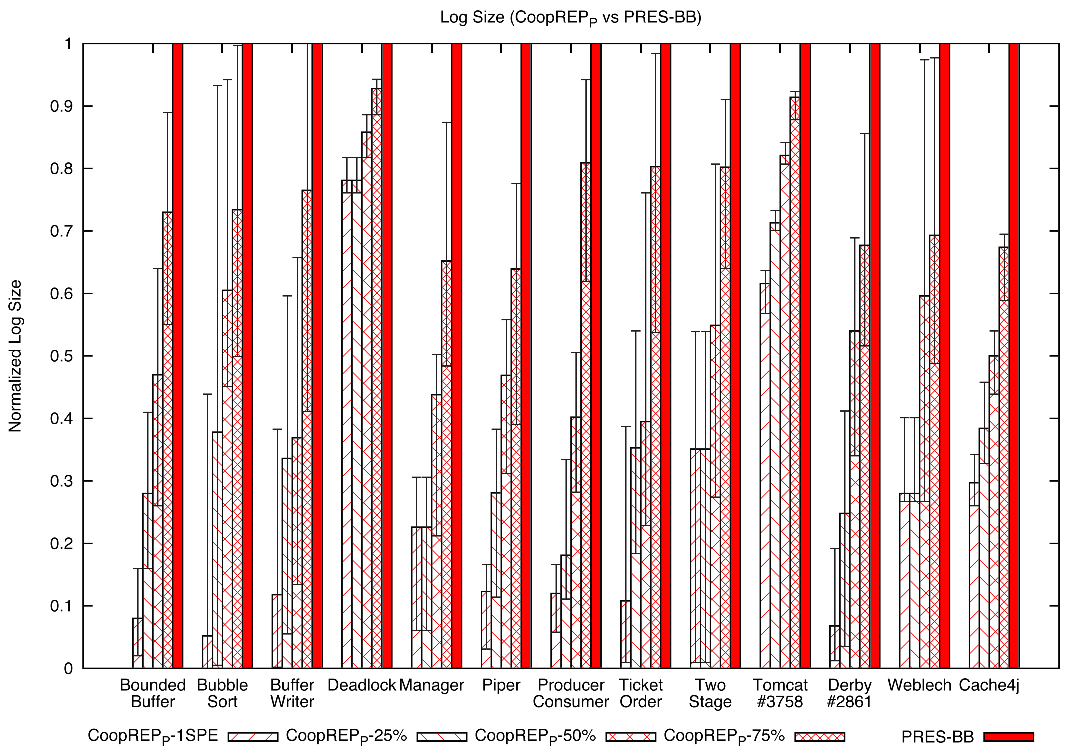


FIGURE 12 Log size reduction achieved by the various CoopREP$_P$ partial logging schemes with respect to PRES-BB. The error bars indicate the minimum and maximum values observed

TABLE 11 Average size of the partial logs generated by the most cost-effective CoopREP schemes for both PRES-BB and LEAP (according to Tables 7 and 8)

Program	PRES-BB	Best of CoopREP$_P$	LEAP	Best of CoopREP$_L$
BoundedBuffer	45 KB	*(1SPE)* 4 KB	6 KB	*(SGM-25%)* 3 KB
BubbleSort	1.2 MB	*(1SPE)* 63 KB	560 KB	*(DGM-25%)* 209 KB
BufferWriter	214 KB	*(DGM-25%)* 72 KB	3.3 MB	*(SGM-75%)* 988 KB
Deadlock	500 B	*(1SPE)* 400 B	500 B	*(1SPE)* 400 B
Manager	29 KB	*(X)* 19 KB	107 KB	*(X)* 23 KB
Piper	17 KB	*(SGM-25%)* 5 KB	7 KB	*(SGM-50%)* 5 KB
ProducerConsumer	15 KB	*(SGM-25%)* 27 KB	8 KB	*(DGM-50%)* 4 KB
TicketOrder	72 KB	*(DGM-50%)* 28 KB	71 KB	*(DGM-50%)* 15 KB
TwoStage	138 KB	*(DGM-75%)* 111 KB	52 KB	*(DGM-50%)* 33 KB
Tomcat#37458	3.4 KB	*(DGM-75%)* 3.1 KB	3 KB	*(1SPE)* 2 KB
Derby#2861	171 KB	*(X)* 116 KB	27 KB	*(DGM-75%)* 20 KB
Weblech	2 KB	*(DGM-25%)* 400 B	2 KB	*(1SPE)* 600 B
Cache4j	3 KB	*(DGM-25%)* 1 KB	3 KB	*(1SPE)* 900 B
AVERAGE	*147 KB*	*35 KB*	*319 KB*	*100 KB*

replay ability. Finally, the partial log combination approach was compared against that of tracing only one SPE per partial log and producing a complete replay log by computing the access vectors most correlated to the error.

The results of the first comparison show that *DGM* tends to be more effective than *SGM* (the former replayed the bug in less attempts than the latter in 10 out of 39 test cases, whereas the opposite scenario was only verified in 3 of the 39 cases). In particular, *DGM* was able to reproduce the failure in very few attempts when the information included in the partial logs encompassed all the most disperse SPEs. When this did not happen, *SGM* was a better choice to find a feasible replay log, because it was quicker in pinpointing groups of similar partial logs.

The results for the second comparison show that CoopREP achieves similar replay ability to LEAP and PRES-BB. In fact, CoopREP$_L$ (ie, CoopREP using LEAP's approach) was able to reproduce the error at the first attempt for 11 out of the 13 benchmarks. In turn, CoopREP$_P$ (ie, CoopREP using PRES-BB's approach) was always able to match the same number of replay attempts as the latter solution.

The results of the third comparison show that using heuristics to combine partial logs is clearly better than using the *1SPE* approach for the large majority of the cases. However, if the program exhibits identical access vectors for almost all SPEs (ie, has a very low SPE dispersion), then tracing only one SPE per partial log and using statistical indicators is sufficient to reproduce the failure in a small number of attempts.

3. **How does CoopREP's effectiveness vary with the number of logs collected?** Section 8.4 evaluates the impact of varying the number of logs collected on the number of attempts required by CoopREP heuristics to replay the bug. The results show that, for programs with low SPE dispersion, the number of logs does not affect the effectiveness of the heuristics. However, for programs with higher SPE dispersion, the lower the number of collected logs, the more difficult it becomes to replay the bug. Concretely, for the program with highest SPE dispersion in these experiments, the bug was only replayed with less than 512 partial logs when each partial log contained 75% of the total SPEs of the program.
4. **How much runtime overhead reduction can CoopREP achieve with respect to other classic, noncooperative logging schemes?** The results described in Section 8.5 show that CoopREP is, indeed, able to achieve lower recording overhead than previous solutions, namely, PRES-BB and LEAP. In particular, CoopREP imposed, on average, 2.4× less overhead than PRES-BB (maximal reduction of 13×) and 2× less than LEAP (maximal reduction of 6×).
5. **How much can CoopREP decrease the size of the logs generated?** The results for the space overhead presented in Section 8.6 follow an analogous trend to those of the runtime overhead: CoopREP generated smaller logs when compared to PRES-BB and LEAP. Concretely, for a similar bug replay ability, CoopREP's partial logs were, respectively, 4.2× and 3.2× smaller than the logs produced by PRES-BB and LEAP, on average.

8.8 | Limitations and open research directions

This section discusses a number of interesting research directions that could be pursued to further enhance CoopREP's applicability and scalability.

Number of user instances. Like any other system that relies on collaborative logging techniques, CoopREP may require the existence of a significant number of user instances running the program, to make the statistical analysis effective. This is particularly true for bugs that do not occur frequently. However, one could argue that today's applications are commonly used by populations of thousands/millions of users, which has been shown to increase the likelihood of rare bugs to manifest [20,30]. Moreover, as demonstrated by the experiments in Section 8.4, the number of instance may not be a problem if the program's SPEs have low dispersion. On the other hand, when the dispersion is high, the number of partial logs collected has indeed a great impact on the effectiveness of CoopREP's heuristics. An interesting future research direction to tackle this issue would be selecting the SPEs to be logged by different user instances using a strategy different from the CoopREP's random selection. For instance, prior work has shown that by taking into account factors such as correlation of variables and SPE overlapping when devising the partial log strategy, one can achieve analogous replay capabilities with significantly less user instances [24].

Random partial logging. Although simple and often effective, random partial logging does not guarantee that 2 partial logs (acquired at different instances) necessarily overlap. The drawback of this is that, if 2 partial logs have no SPEs in common, it is impossible to deduce whether these partial logs were traced from equal executions and, consequently, are suitable to be combined. Furthermore, random partial logging disregards the fact that SPEs that depend on one another should be traced in the same logging profile, to avoid having to combine dependent information collected from potentially different production runs.

More effective partial logging profiles can be achieved by resorting to program analysis techniques to identify SPE dependencies and optimization models. Alternatively, one could leverage machine learning techniques to infer the minimum SPE coverage needed for a target software system, based on similarities with different systems that were successfully traced by CoopREP. Example of features that may be used to detect such similarities could include, for instance, the average SPE dispersion, the total number of SPEs, and the number of threads.

Replay guarantees. Like most search-based systems, CoopREP offers best-effort determinism, in the sense that it can only reproduce the bug if there are compatible partial logs. This means that, despite having collected a great amount of information with respect to failing executions, CoopREP is only able to successfully replay the error if it finds a combination of access vectors that allows mimicking a thread interleaving that leads to the failure. When there is no such combination, CoopREP is not able to leverage this information in any other way. Although the results of the experiments show that cooperative record and replay is effective in reproducing concurrency failures, a possible solution to improve CoopREP's bug replay ability would be to use the information in the partial logs to guide a symbolic execution of the program and find the buggy thread schedule via constraint solving techniques, such as in Huang et al. [9]

9 | CONCLUSIONS AND FUTURE WORK

This paper introduces CoopREP, a system that provides fault replication of concurrent programs via cooperative logging and partial log combination. CoopREP achieves significant reductions of the overhead incurred by other state-of-the art systems by letting each instance of a program trace only a subset of its shared programming elements (eg, 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 real-world applications, highlighted the effectiveness of the proposed technique, in terms of its capability to successfully replay nontrivial concurrency bugs, as well as its performance advantages with respect to noncooperative logging schemes.

This paper also discusses a number of challenging research directions, including not only the design of additional partial logging schemes but also new heuristics and metrics to correlate partial logs. Finally, CoopREP can also be easily integrated with bug localization tools, [31,32] to automatically identify nonvisible bugs and isolate the error's root cause.

ACKNOWLEDGEMENTS

The authors wish to thank the anonymous reviewers for their valuable and constructive feedback on both the conference and the journal versions of this paper. Also, a special word for Nuno Diegues for the helpful discussions. This work was partially supported by (1) national funds through Fundação para a Ciência e a Tecnologia (FCT), under project UID/CEC/50021/2013 and (2) project "TEC4Growth—Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

ORCID

Nuno Machado http://orcid.org/0000-0003-1531-1875

REFERENCES

1. S. Park et al., *Probabilistic replay with execution sketching on multiprocessors*, Proceedings of the International Symposium on Operating Systems PrinciplesPres. SOSP '09. ACM, Big Sky, Montana, USA, 2009, pp. 177-192.
2. G. W. Dunlap et al., *Execution replay of multiprocessor virtual machines*, Proceedings of the International Conference on Virtual Execution Environments, VEE '08. ACM, Seattle, WA, USA, 2008, pp. 121-130.
3. A. Georges et al., *Jarec: A portable record/replay environment for multi-threaded java applications*. Software Pract. Exper. 2004;**34**(6), 523-547.
4. J.-D. Choi, and H. Srinivasan. *Deterministic replay of Java multithreaded applications*, Proceedings of the International Symposium on Parallel and Distributed Tools, SPDT '98. ACM, Welches, Oregon, USA , 1998, 48-59.
5. T. J. LeBlanc and J. M. Mellor-Crummey. *Debugging parallel programs with instant replay*. IEEE Trans. Comput. 1987;**36**, 471-482.
6. J. Huang, P. Liu, and C. Zhang. *Leap: lightweight deterministic multi-processor replay of concurrent Java programs*, Proceedings of the International Symposium on Foundations of Software Engineering, FSE '10. ACM, Santa Fe, New Mexico, USA, 2010, pp. 207-216.
7. G. Altekar, andI. Stoica. *ODR: output-deterministic replay for multicore debugging*, Proceedings of the Symposium on Operating Systems Principles, SOSP '09. ACM, Big Sky, Montana, USA, 2009, pp. 193-206.
8. C. Zamfir, and G. Candea. *Execution synthesis: a technique for automated software debugging*, Proceedings of the European Conference on Computer Systems, EuroSys '10. ACM, Paris, France, 2010, pp. 321-334.
9. J. Huang, and C. Zhang, J. Dolby. *CLAP: recording local executions to reproduce concurrency failures*, Proceedings of the International Conference on Programming Language Design and Implementation, PLDI '13. ACM, Seattle, Washington, USA, 2013, pp. 141-152.
10. T. C. Bressoud, and F. B. Schneider. *Hypervisor-based fault tolerance*, Proceedings of the International Symposium on Operating Systems Principles, SOSP '95. ACM, Copper Mountain, Colorado, USA, 1995, pp. 1-11.
11. M. Musuvathi et al., Finding and reproducing heisenbugs in concurrent programs, Proceedings of the International Conference on Operating Systems Design and Implementation, OSDI '08. USENIX Association, San Diego, California, USA, 2008, pp. 267-280.
12. G. W. Dunlap et al., *Revirt: enabling intrusion analysis through virtual-machine logging and replay*. SIGOPS Oper. Syst. Rev. 2002;**36**(SI), 211-224.
13. A. Avizienis et al., *Basic concepts and taxonomy of dependable and secure computing*. IEEE Trans. Dependable Secur. Comput. January 2004;**1**(1), 11-33.
14. M. Xu , R. Bodik, and M. D. Hill. *A "flight data recorder" for enabling full-system multiprocessor deterministic replay*, Proceedings of the International Symposium on Computer Architecture, ISCA '03. ACM, San Diego, California, USA, 2003, pp. 122-135.
15. S. Narayanasamy , G. Pokam , B. Calder. *Bugnet: continuously recording program execution for deterministic replay debugging*, Proceedings of the International Symposium on Computer Architecture, ISCA '05. IEEE Computer Society, Madison, Wisconsin, USA, 2005, pp. 284-295.
16. P. Montesinos , L. Ceze , and J. Torrellas. *Delorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently*, Proceedings of the International Symposium on Computer Architecture, ISCA '08. IEEE Computer Society, Beijing, China, 2008, pp. 289-300.
17. P. Montesinos et al., *Capo: a software-hardware interface for practical deterministic multiprocessor replay*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '09. ACM, Washington, DC, USA, 2009, pp. 73-84.
18. Z. Yang et al., *Order: object centric deterministic replay for Java*, Proceedings of the USENIX Annual Technical Conference, ATC '11. USENIX Association, Portland, OR, 2011, pp. 30-43.
19. Y. Jiang et al., *Care: cache guided deterministic replay for concurrent Java programs*, Proceedings of the International Conference on Software Engineering, ICSE '14. ACM, Hyderabad, India, 2014, pp. 457-467.
20. B. Liblit et al., *Bug isolation via remote program sampling*, Proceedings of the International Conference on Programming Language Design and Implementation, PLDI '03. ACM, San Diego, California, USA, 2003, pp. 141-154.
21. G. Jin et al., *Instrumentation and sampling strategies for cooperative concurrency bug isolation*, Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10. ACM, Reno/Tahoe, Nevada, USA, 2010, pp. 241-255.
22. R. L. Halpert , C. J. F. Pickett , and C. Verbrugge. *Component-based lock allocation*, Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, PACT '07. IEEE Computer Society, Brasov, Romania, 2007, pp. 353-364.

23. J. Steven et al., *jRapture: a capture/replay tool for observation-based testing*, Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '00. ACM, Portland, Oregon, USA, 2000, pp. 158-167.
24. N. Machado , P. Romano , L. Rodrigues. *Property-driven cooperative logging for concurrency bugs replication*, Proceedings of the International Workshop on Hot Topics in Parallelism, HotPar '13. USENIX Association, San Jose, California, USA, 2013, pp. 1-12.
25. R. Hamming. *Error detecting and error correcting codes*. Bell Syst. Tech. J. 1950;**26**(2), 147-160.
26. E. Farchi , Y. Nir , and S. Ur. *Concurrent bug patterns and how to test them*, Proceedings of the International Symposium on Parallel and Distributed Processing, IPDPS '03. IEEE Computer Society, Nice, France, 2003, pp. 286-293.
27. J. Huang , and C. Zhang. *Lean: simplifying concurrency bug reproduction via replay-supported execution reduction*, Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12. ACM, Tucson, Arizona, USA, 2012, pp. 451-466.
28. K. Sen. *Race directed random testing of concurrent programs*, Proceedings of the International Conference on Programming Language Design and Implementation, PLDI '08. ACM, Tucson, Arizona, USA, 2008, pp. 11-21.
29. B. Lucia , B. P. Wood , and L. Ceze. *Isolating and understanding concurrency errors using reconstructed execution fragments*, Proceedings of the International Conference on Programming Language Design and Implementation, PLDI '11. ACM, San Jose, California, USA, 2011, pp. 378-388.
30. G. Candea. *Exterminating bugs via collective information recycling*, Proceedings of the Workshop on Hot Topics in Dependable Systems, HotDep '11. IEEE Computer Society, Hong Kong, China, 2011, pp. 1-15.
31. S. Park , R. W. Vuduc , and M. J. Harrold. *Falcon: fault localization in concurrent programs*, Proceedings of the International Conference on Software Engineering, ICSE '10. ACM, Cape Town, South Africa, 2010, pp. 245-254.
32. N. Machado , B. Lucia , and Luís Rodrigues. *Concurrency debugging with differential schedule projections*, Proceedings of the International Conference on Programming Language Design and Implementation, PLDI '15. ACM, Portland, Oregon, USA, 2015, pp. 586-595.

How to cite this article: Machado N, Romano P, Rodrigues L. CoopREP: Cooperative record and replay of concurrency bugs. *Softw Test Verif Reliab*. 2017;e1645. https://doi.org/10.1002/stvr.1645