Fault Replication in Concurrent Programs

(extended abstract of the MSc dissertation)

Nuno de Ferraz Almeida e Peixoto Machado
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—This thesis addresses the problem of reproducing an execution of a concurrent buggy program, in order to ease its debugging. For this, the thesis introduces CoopLEAP, a system that provides fault replication of concurrent programs, based in cooperative recording and partial log combination. CoopLEAP employs a partial recording scheme to reduce the amount of information that a given program instance is required to store in order to support deterministic replay. The use of partial logs allows to substantially reduce the overhead imposed by the instrumented code execution, but raises the problem of finding the combination of logs capable of replaying the fault. This thesis also proposes an heuristic, denoted Similarity-Guided Merge, to perform this search. Third-party benchmarks and a real-world application, used to evaluate the implemented prototype of CoopLEAP, shown that it can not only successfully replay concurrency bugs, but also impose smaller overheads in comparison with other existing solutions.**

I. Introduction

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

Unfortunately, achieving deterministic replay is not a trivial task, especially in parallel applications. Contrary to sequential bugs that usually depend only on the program input and on execution environments (and therefore can be easily reproduced), concurrency bugs show an inherently nondeterministic nature. This means that even when executing the same code on the same machine with the same input, the exact timing of an instruction or code segment execution may vary from one run to another [4].

The *deterministic replay* technique addresses this problem by recording the execution relevant details [5] (including the order of access to shared memory regions, thread scheduling, program inputs, signals, etc), a task that induces a large space and performance overhead during production runs. However, if one records too little data, it may not be sufficient to reproduce the bug.

In the past decade, a significant amount of research has been performed in order to develop efficient solutions (either based on hardware or software) that provide deterministic replay. Several of these solutions [6], [7], [8] aim at replaying the bug on the first attempt, but this comes with an excessively high cost (10x-100x slowdown) on the original run, which is still too expensive to be practical.

As user-side executions are much more performance critical when compared to developers' in-house debugging, it is important to reduce the production run overhead, even if it results in a slightly longer bug-reproduction time during diagnosis. Our observation is that one can further mitigate the runtime penalties by exploring the fact that there is usually a large number of users running the faulty software. By gathering, analyzing and combining information recorded from different users regarding program's faulty runs, one can make bug reproduction more cost-effective. If each user collects only a fraction of the traces, the performance of its instance of the program will not be so significantly affected.

Thereby, this thesis introduces CoopLEAP, a deterministic replay system (based on LEAP [9]) which leverages on cooperative recording performed by multiple clients and on statistical techniques to combine the collected partial logs. Therefore, the contributions of this thesis can be enumerated as follows:

- A set of novel statistical metrics to detect correlations between partial logs;
- A novel heuristic, named Similarity-Guided Merge, that leverages on the previous metrics to systematically perform a guided search, among the possible combinations of partial logs, to find those which generate complete replay drivers capable of reproducing the bug with high probability.
- An experimental evaluation of the implemented prototype of CoopLEAP, based on third-party benchmarks and on a real-world application.

The rest of this document is structured as follows: Section II presents the background concepts related to this work. Section III overviews some deterministic replay and statistical debugging systems. Section IV introduces CoopLEAP, describing in detail not only its architecture, but also the Similarity-Guided Merge heuristic and the metrics used to

measure similarity of partial logs. Section V shows the results of the experimental evaluation study. Finally, Section VI concludes this document by summarizing its main points and future work.

II. Background

A. Deterministic replay

Deterministic replay (or record/replay) aims to overcome the problems associated with the reproduction of bugs, in particular those raised by non-determinism. The purpose of this technique is to re-execute the program, obtaining the exact same behavior as the original execution. This is possible because almost all instructions and states can be reproduced as long as all possible non-deterministic factors that have an impact on the program's execution are replayed in the same way [4]. Thereby, deterministic replay operates in two phases: *1) Record phase* – consists of capturing data regarding non-deterministic events, putting that information into a trace file; *2) Replay phase* – the application is re-executed consulting the trace file to force the replay of non-deterministic events according to the original execution.

B. Non-determinism

External factors often interfere with the program execution, preventing the timing and the sequence of instructions executed to be always identical. The sources of these factors can be divided into two types: *input non-determinism* and *memory non-determinism* [10]. Input non-determinism encompasses all the inputs that are received by the system layer being recorded and are not produced by that layer (e.g. signals, system calls, hardware interrupts, DMA, etc), and is present in both single-processor and multi-processor machines.

Regarding memory non-determinism in single-processor, it is mainly due to variations in tread scheduling and signal delivering, as a result of differences in the architectural state (e.g. cache line misses, memory latencies, etc). This kind of non-determinism can be tackled by recording the events with "logical time" [11], instead of ordinary physical time. In fact, logical time may be sufficient to support deterministic replay in single-processor systems [12]. However, when one moves to the field of multi-processors (e.g. SMPs and multi-cores), the scenario becomes more complex. In addition to thread scheduling, asynchronous events, and signals, one has to take into account how concurrent threads interleave with each other, since they actually execute simultaneously on different processors. Therefore, one needs to capture the global order of shared memory accesses and synchronization points. Obviously, this is not a problem when threads are independent from each other.

III. Related Work

There are various approaches to prevent bugs in a program and to optimize the debugging process. In this section we focus on approaches that try to reproduce the failure or to statistically isolate it, as these are the most relevant to the work reported in the thesis.

Among the deterministic replay solutions, over the past few years, several solutions have been proposed to cope with the challenges brought by multi-processors. Based on how they are implemented, prior work can be divided in two main categories: *hardware-based* and *software-based*.

The hardware-based solutions rely on hardware extensions to efficiently record the non-deterministic events and, consequently, mitigate the overhead imposed to the production run. As examples, one can highlight FDR [13], BugNet [14], and, more recently, DeLorean [15]. However, despite various optimization endeavors to reduce hardware complexity [15], all the previous approaches still demand significant hardware modifications. These modifications are not yet available nowadays, except on simulations.

Regarding the software-based approach, InstantReplay [8] was the first deterministic replay system to support multi-processors. It leverages on a instrumented version of CREW protocol to control and log the accesses to shared memory locations. JaRec [6] reduces the overheads imposed by InstantReplay, by dropping the idea of global ordering and using a *Lamport's clock* [11] to preserve the partial order of threads and to reduce the size of logs. However, JaRec requires a program to be data race free in order to guarantee a correct replay, otherwise it only ensures deterministic replay up until the first data race. This constraint makes this approach unattractive for most real world concurrent applications, given that is common the existence of benign or harmful data races. All the previous approaches try to reproduce the bug on the first replay run, thus inducing large overheads during production runs. This also has the drawback of penalizing bug-free executions, which are much more frequent than the faulty ones [4]. Motivated by this, some recent solutions, such as PRES [4], ODR [16], and ESD [17], relax the constraint of replaying the bug at the first try, by only logging partial information in order to further minimize the cost of recording the original execution. Later, these solutions apply inference techniques to complete the missing information.

Our solution, denoted CoopLEAP, also relies on the observation that it is not crucial to achieve deterministic replay at the first attempt, but improves these systems as it leverages on information logged by multiple clients to ease the inference task. For this, CoopLEAP draws on statistical debugging techniques, which aim at isolating bugs by analyzing information gathered from a large number of users. Statistical debugging was pioneered by CBI [18]. This system collects feedback reports that contain values recorded for certain predicates of the program (e.g. branches, return values, etc). Then, performs a statistical analysis of the information gathered in order to pinpoint the likely source of the failure. However, CBI does not support concurrency bugs. CCI [19] outstrips this limitation by adjusting CBI's principles to cope with non-deterministic events. For instance, it implements new sampling techniques, with longer sampling periods, because concurrency bugs always involve multiple memory accesses.

Since recording and replaying input non-determinism can be achieved with an overhead less than 10% [4], [5], [12], the work presented in this thesis only focuses on coping with memory non-determinism. In fact, a recent study on the evolution of the types of errors in MySQL database [20] shows a growth trend in the number and proportion of concurrency bugs over the years. Thereby, CoopLEAP addresses the deterministic replay of this kind of bugs (e.g. atomicity violations, data races), disregarding other sources of non-determinism.

IV. CoopLEAP System

This section introduces CoopLEAP, a system that provides fault replication of concurrent programs, based in cooperative recording and partial log combination. Given that CoopLEAP is based on LEAP [9], it shares most of its features and its main components. Hence, we begin by presenting an overview of LEAP.

A. Standard LEAP

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

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

To locate the shared program elements (SPEs), LEAP uses a static escape analysis called *thread-local objects analysis* [21] from the Soot[1] framework. Given that accurately identifying shared variables is generally an undecidable problem, this technique computes a sound over-approximation, i.e. every shared access to a field is indeed identified, but some accesses which are actually not may also be classified as shared [9]. Are considered as SPEs variables that serve as monitors (including Java monitors) and other shared field variables (including class and thread escaped instance variables). For each identified SPE, LEAP assigns offline a numerical index in order to be able to consistently identify objects across different runs. Moreover, as access vectors only contain thread IDs tracked during the production run, it is imperative to correctly recognize each thread in both recording and replay phases. LEAP achieves this by maintaining a mapping between the thread name and the thread ID during recording and using the same mapping for replay.

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

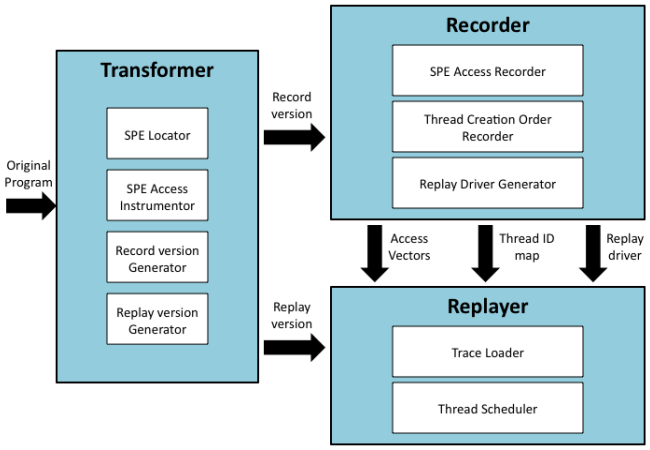

Figure 1. Overview of the LEAP architecture (adapted from the LEAP paper).

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

The transformer receives the Java program bytecode and employs two types of instrumentation schemes to produce the *record version* and the *replay version*, respectively.

The record version is then executed and the recorder component stores the accesses to each SPE in its correspondent access vector. When the production run ends, LEAP generates three different files: the access vectors, the thread ID map information, and the replay driver.

Finally, the replayer uses the logged information and the generated replay driver to start the execution of the replay version of the program. To guarantee the correct execution order of threads, LEAP takes control of the thread scheduling and consults the thread ID map information file.

The evaluation study presented in [9], shows that LEAP incurs a runtime overhead ranging from 7.3% to 626% (for applications with several shared variables accessed in hot loops). In terms of space overhead, the log size in LEAP is still considerable, ranging from 51 to 37760 KB/sec. To mitigate the log sizes and further minimize the recording overhead, CoopLEAP extends LEAP to leverage on information recorded by multiple users of the program and support partial log combination.

B. CoopLEAP Architecture

1) Overview: Figure 2 illustrates the overall architecture of CoopLEAP. During the instrumentation phase (Figure 2 - 1), the *transformer* instruments the Java program bytecode to generate both the record version and the replay version, as done in LEAP. The nuance resides on the record version, where only a subset of all SPEs is actually instrumented (as

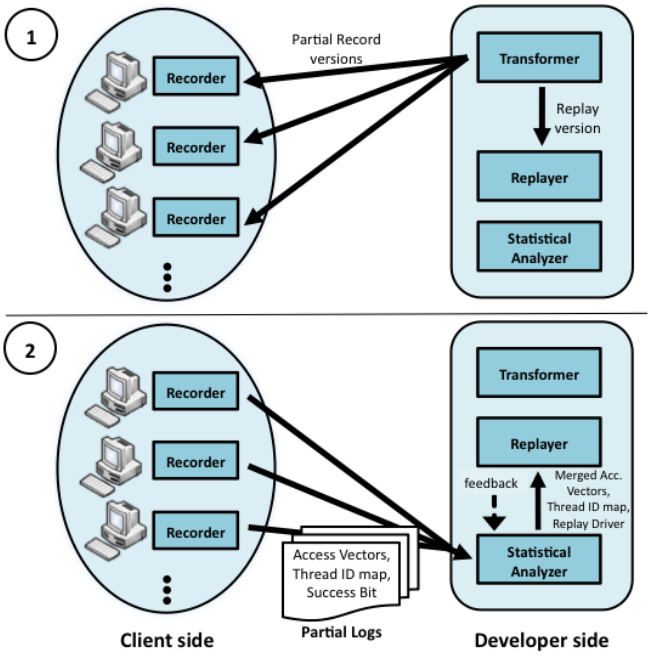

Figure 2. Overview of the CoopLEAP architecture: (1) Instrumentation phase; (2) Record and Replay phases.

this version will be further called as *partial record version*). It should also be noted that each client is assigned a different subset of SPEs to record, according to some defined criterion (see Section IV-C). The partial record versions are then sent to the clients, whereas the replay version is sent to the replayer.

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

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

Here, the *statistical analyzer* will employ an heuristic (see Section IV-E) to explore and merge the received access vectors in order to generate a complete log of the faulty execution[2]. By pinpointing the most correlated partial logs, CoopLEAP plans to ease the task of inferring the original unrecorded access vectors. Once the merge of partial logs is complete, the combination of access vectors is sent to the *replayer*, along with the thread ID map and the generated replay driver.

Finally, just like in LEAP, the replay driver will serve as an entry point for the replayer to control the replaying of the program execution. However, in CoopLEAP the replay phase is slightly different. Given that access vectors come from independent executions, the combined information can be incompatible. As a result, the execution replay will fail and the bug will not be reproduced. In this case, the replayer will send *feedback* to the statical analyzer communicating the replay failure, so the latter can investigate another access vector combination and produce a new complete log for replay. This process ends when the bug is successfully replayed or when the maximum number of attempts to do it is reached.

C. Partial Log Recording

CoopLEAP introduces the novelty of recording accesses to only a fraction of the entire set of the SPEs of the program. The subset of SPEs to be traced is defined at instrumentation time by the transformer. For this purpose, different criteria can be used, e.g. random selection, load balancing distribution, subset of fixed SPEs, etc. However, in this work, we only consider the random selection of a certain percentage of the total number of SPEs of the program for partial recording. Basically, whenever a new SPE is identified, CoopLEAP generates a random value and compares it with a pre-defined threshold (used to bound the percentage of the total number of SPEs to be instrumented) to decide whether the SPE is to be instrumented or not.

It should be noted that this scheme is statistically fair, assuming that there is a significant number of users running the program. However, it is not granted to be optimal, as it does not always allow partial log overlapping, i.e. it may not exist SPEs in common for each two potential similar partial logs. For instance, let us consider a program with eight SPEs (`speIndex = {0..7}`) and a coverage of 50%. It could happen that two partial logs (l_1 and l_2) collected from two identical executions may not be considered as similar, since l_1 may only record the subset `{0..3}` and l_2 may only record the subset `{4..7}`, for example. This could be addressed by increasing the percentage of coverage (at the cost of greater overheads), or by defining a smaller fixed subset of SPEs to be logged by all users.

Moreover, one can also note that the overhead reductions may not be linear with the decrease of the coverage percentage. The reason is because some SPEs could be accessed more times than others, therefore, when instrumenting the code, the load balance may not be equally

[2]In this work, we are assuming that all the partial logs refer to the same bug. Despite that, for different bugs on the same program, some additional data could be used for distinction purposes, namely the line of code where the bug appeared.

distributed among the users. A solution for this could be instrumenting the whole program and execute it one time to measure the number of accesses performed on each SPE, at the developer side. Later, when instrumenting the user versions, CoopLEAP could already take into account the SPEs workload.

The investigation of new partial recording schemes is scheduled as future work.

D. Merge of Partial Logs

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

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

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

1) Statistical Metrics for Partial Log Correlation: These metrics are related to the partial logs as a whole and measure the amount of information that they have in common, so one can increase the probability of merging compatible access vectors. In particular, the following statistical metrics are used to calculate the partial log correlation: *Similarity* and *Relevance*. Both metrics are described in detail below.

a) Similarity – The rationale behind the classification of the similarity between two partial logs is related to their number of SPEs with identical access vectors (i.e. that had recorded exactly the same thread interleaving). Hence, the more SPEs with equal access vectors the partial logs have, the better. The computation of this metric can come in two flavors: *Plain Similarity* and *Dispersion-based Similarity*, according to the weight given to the SPEs of the program.

To better define these metrics, let us first present some formal notation:

$\mathcal{S}$ – Set of all the SPE identifiers of the program.

$\mathcal{S}_l$ – Set of the SPE identifiers recorded only by the partial log l.

$\mathcal{AV}$ – Set of the hashes of the access vectors recorded by all the partial logs.

$\mathcal{AV}_l$ – Set of the hashes of the access vectors recorded only by the partial log l.

$avecs(s) : \mathcal{S} \rightarrow \{\mathcal{AV}_1, \mathcal{AV}_2, ..., \mathcal{AV}_n\}$ – Map that, for a given SPE identifier s, returns the set of the hashes of its different access vectors, recorded by all the partial logs.

$avec_l(s) : \mathcal{S}_l \mapsto \mathcal{AV}_l$ – Function that maps a SPE identifier s to the hash of its access vector, recorded by the partial log l.

$Equal_{l_1,l_2} = \{s \mid s \in \mathcal{S}_{l_1} \cap \mathcal{S}_{l_2} \wedge avec_{l_1}(s) = avec_{l_2}(s)\}$ – Set of the SPE identifiers, recorded by both partial logs l_1 and l_2, with identical access vectors.

$Diff_{l_1,l_2} = \{s \mid s \in \mathcal{S}_{l_1} \cap \mathcal{S}_{l_2} \wedge avec_{l_1}(s) \neq avec_{l_2}(s)\}$ – Set of the SPE identifiers, recorded by both partial logs l_1 and l_2, with different access vectors.

$Sim_{l_0} = \{l_1, l_2, ..., l_k\}$ – Set of the k partial logs more similar to l_0 (denoted as *group of similars of* l_0).

$Fill_{l_0,Sim_{l_0}} = \{s \mid s \in \mathcal{S}_{l_0} \cup \mathcal{S}_{l_1} \cup \mathcal{S}_{l_2} \cup ... \cup \mathcal{S}_{l_k} \wedge l_1, l_2, ..., l_k \in Sim_{l_0}\}$ – Union of the sets of the SPE identifiers recorded by the partial log l_0 and by the partial logs of its group of similars Sim_{l_0}.

With this, we can now define the metrics as follows.

Let l_1 and l_2 be two partial logs, their *Plain Similarity* is given by the following equation:

$$PlainSimilarity(l_1, l_2) = \frac{\#Equal_{l_1,l_2}}{\#\mathcal{S}} \times \left(1 - \frac{\#Diff_{l_1,l_2}}{\#\mathcal{S}}\right) \quad (1)$$

where $\#Equal_{l_1,l_2}$, $\#\mathcal{S}$, and $\#Diff_{l_1,l_2}$ denote the cardinality of the sets $Equal_{l_1,l_2}$, $\mathcal{S}$, and $Diff_{l_1,l_2}$, respectively.

It should be noted that this metric will only be 1 when both logs are complete and identical, i.e. they have recorded access vectors for all the SPEs of the program ($\mathcal{S}_{l_1} = \mathcal{S}_{l_2} = \mathcal{S}$) and those access vectors are equal for both logs ($avec_{l_1}(s) = avec_{l_2}(s), \forall s \in \mathcal{S}$). This implies that, for every two partial logs, their plain similarity will always be less than 1. However the greater this value is, the more probable is that the both partial logs come from the same production run.

Let l_1 and l_2 be two partial logs, their *Dispersion-based Similarity* is given by the following equation:

$$DispersionSimilarity(l_1, l_2) = \sum_{x \in Equal_{l_1,l_2}} weight(x) \times \left(1 - \sum_{y \in Diff_{l_1,l_2}} weight(y)\right) \quad (2)$$

where $weight(s)$ is a function of type $\mathcal{S} \rightarrow Double$ that maps each SPE identifier to a double value referent to its relative weight, in terms of *overall-dispersion*. Here, the overall-dispersion of a given SPE corresponds to the proportion of its different access vectors when compared to the total number of different access vectors collected for all the SPEs. Thereby, the weight function of a SPE identifier s can be calculated as follows:

$$weight(s) = \frac{\#avecs(s)}{\#\mathcal{AV}} \quad (3)$$

Notice that some other metrics could be defined if one consider other types of weights (e.g. the average number of accesses recorded for each SPE), but in this work we only use overall-dispersion.

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

cases when there are many SPEs whose access vectors are identical in every execution.

b) Relevance – this metric allows to classify each partial log according to its likelihood of being completed with compatible information:

$$Relevance(l_0) = \alpha \times \frac{\#Fill_{l_0, Sim_{l_0}}}{\#\mathcal{S}} + (1-\alpha) \times \frac{\sum_{n=1}^{k} Similarity(l_0, l_n)}{k}, l_n \in Sim_{l_0} \quad (4)$$

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

As one can see, the Relevance metric is the sum of two parcels with different importance (given by α). The first parcel is related to the number of SPEs that is possible to fill joining the access vectors from the partial log l_0 and its group of similars Sim_{l_0}. This follows the rationale that the more missing SPEs of l_0 that can be filled with access vectors from similar partial logs, the better.

In turn, the second parcel gives the similarity ratio of all the partial logs in the group. This allows to pick, as the base partial log, the one whose group of similars is composed by partial logs with high similarity, thus increasing the probability of merging compatible information. It should be noted that the maximum size k of the group of similars can be defined by the developer. Moreover, a partial log l_1 can only be part of Sim_{l_0} if $Similarity(l_0, l_1) \geq threshold$. This avoids the group of similars to be composed by partial logs with a very low value of similarity.

In our experiments, we found $\alpha = 0.7$, $k = 5$, $threshold = 0.3$ for Plain Similarity, and $threshold = 0.01$ for Dispersion-based Similarity, to be good values.

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

To compute these metrics, we adapt the scoring method proposed by Liblit et al [18]. Thereby, access vectors are classified based on their *Sensitivity* and *Specificity*, i.e. whether they account for many failed runs and few successful runs. With this information, it is possible to define a third metric, denoted *Importance*, which identifies the access vectors that are simultaneously high sensitive and specific.

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

$$Sensitivity(v) = \frac{F(v)}{F_{total}} \quad (5)$$

$$Specificity(v) = \frac{F(v)}{S(v) + F(v)} \quad (6)$$

$$Importance(v) = \frac{2}{\frac{1}{Sensitivity(v)} + \frac{1}{Specificity(v)}} \quad (7)$$

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

E. Similarity-Guided Merge

To merge the partial logs and generate a complete log capable of replaying the faulty execution, we developed a heuristic denoted *Similarity-Guided Merge*. This heuristic operates in the following five steps:

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

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

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

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

5. Complete the base partial log with statistical indicators – if it is still not possible to complete the log for replay (the union of the different groups of similars may not cover all the SPEs of the program), CoopLEAP applys the metrics described in Section IV-D2 to the universe of access vectors collected, and picks the ones with greater Importance (see Equation 7) to fill the missing SPEs.

At the end of this process, CoopLEAP replays the merged log and verifies if the bug is reproduced. If it is, the goal has been achieved and the process ends. If it is not, CoopLEAP chooses the next partial log in the list of the most relevant to

be the new base partial log and re-executes the Similarity-Guided Merge from the step 3. It should be referred that, in the worst case scenario, where all the most important indicators failed to replay the bug, the heuristic switches to a *brute force* mode. Here, all the possible access vectors are tested for each missing SPE.

V. Evaluation

A. Experimental Setting

All the experiments were conducted in a machine Intel Core 2 Duo at 2.26 Ghz, with 4 GB of RAM and running Mac OS X. CoopLEAP prototype was implemented over a LEAP public version[3]. In order to get comparative figures, this standard version of LEAP was also used in the experiments.

Regarding partial logging, three different configurations were employed: CLEAP-25%, CLEAP-50%, and CLEAP-75%, where each partial log traces 25%, 50%, and 75% of the SPEs of the program, respectively. For each configuration, 500 partial logs from failed executions were used, plus more 50 of successful runs. To get a fairer comparison of the three recording schemes, the partial logs were generated from 500 complete logs, picking randomly the SPEs to be stored according to the scheme's percentage.

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

B. Evaluation Criteria

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

To assess CoopLEAP's bug replay capacity, we used some bugs from the IBM ConTest benchmark suite [22], and a real bug from the widely-used application Tomcat. To measure the overheads imposed, we compared CoopLEAP against LEAP on the Java Grande workload benchmark.

C. Bug Replay Capacity

1) ConTest Benchmark: The IBM ConTest benchmark suite [22] contains programs with many types of concurrency bugs. The ones used in our experiments are described in Table I, in terms of its number of SPEs, the total number of SPE accesses, and the bug-pattern according to [22].

Table II shows the number of attempts of the heuristic (using both Plain Similarity and Dispersion-based Similarity) to replay the ConTest benchmark bugs, when recording 25%, 50%, and 75% of the SPEs.

[3]Available at http://sites.google.com/site/leaphkust/

Program	SPEs	Total Accesses	Bug Description
BubbleSort	10	49964	Not-atomic
Manager	4	30240	Not-atomic
TwoStage	4	27103	Two-stage
ProducerConsumer	8	997	Orphaned thread
Piper	6	347	Missing condition for Wait

Table I
DESCRIPTION OF THE CONTEST BENCHMARK BUGS USED IN THE EXPERIMENTS.

Program	Plain Similiarity			Dispersion-based Similarity		
	25%	50%	75%	25%	50%	75%
BubbleSort	1	1	1	1	1	1
Manager	*X*	*X*	*X*	*X*	*X*	*X*
TwoStage	34	13	*X*	7	1	1
ProducerConsumer	5	2	1	1	1	1
Piper	2	1	1	1	1	1

Table II
NUMBER OF THE ATTEMPTS REQUIRED BY THE HEURISTIC TO REPLAY BUG IN THE CONTEST BENCHMARK (THE *X* INDICATES THAT THE HEURISTIC FAILED TO REPLAY THE BUG IN THE MAXIMUM NUMBER OF ATTEMPTS STIPULATED).

Analyzing the results, one can verify that the Similarity-Guided Merge heuristic only failed to replay the bug in programs `Manager` and `TwoStage` (only in the particular case of Plain-Similarity for CLEAP-75%). These results confirm that executions containing a higher number of total SPE accesses are more unlikely to be successfully reproduced using partial logging strategies. However, an overall high number of accesses per se is not an indicator that the heuristic will fail, as shown by the results obtained for program `BubbleSort`. The main reason for the failure of the heuristic is related to how the accesses are distributed between the SPEs and how that influences the SPE's dispersion ratio. Here, the dispersion ratio indicates *how disperse is the SPE*, i.e. whether many different access vectors were recorded for it or not. The dispersion ratio is computed by dividing the number of *different* access vectors recorded for the SPE by the total number of access vectors recorded for that SPE. Figure 3 depicts the SPE dispersion ratios for the ConTest benchmark programs (for the sake of readability and to ease the comparison, Figure 3 only presents the values for the full logging configuration).

As one can note, the `BubbleSort` program has only one SPE with a very high dispersion ratio (this SPE also accounts for about 99% of the total accesses), while the remaining SPEs always present the same access vector across all the executions. For this reason, the partial log combination ended to be trivial, since it was easy for the Similarity-Guided Merge heuristic to combine compatible information.

On the other hand, the `Manager` program has all its SPEs with a dispersion ratio of 1 or closer, which means that almost all the recorded executions had a different thread interleaving. These clearly represent unfavorable conditions for the partial logging approach, which in fact failed to

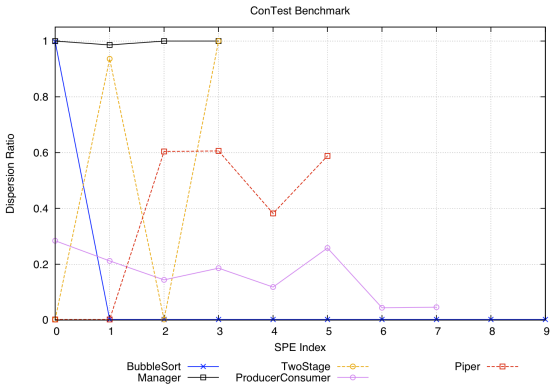

Figure 3. SPE dispersion ratios for the ConTest benchmark programs, when logging all the SPEs of the program.

replay the bug, as indicated in Table II.

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

CLEAP-75% – with this configuration, each partial log was composed by three SPEs. Hence, the list of base partial logs ended being composed by the partial logs whose group of similars contained only other partial logs matching in the SPEs 0 and 2. As a consequence, the access vectors combined for filling either SPE 1 or 3 were incompatible.

CLEAP-50% – with this configuration, each partial log was composed by two SPEs. Here, the list of base partial logs was filled with the partial logs that have other ones matching access vectors for the SPE 1. This because all the partial logs containing only SPEs 0 and 2, albeit having many other similar partial logs, could not generate a complete replay log just by combining information from their group of similars. Therefore, their relevance was lower (see Equation 4). The same did not happened for the partial logs containing SPE 1 and the bug was replayed by trying different access vectors for filling SPE 4.

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

On the other hand, when using Dispersion-based Similarity, the heuristic could easily reproduce the bug, because the SPEs had different importances. Thence, the partial logs with the same access vector for SPE 1 were immediately identified as the best base partial logs and used to generate a complete replay log.

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

2) Tomcat: Tomcat[4] is a widely-used complex server application. The bug replay capacity of the Similarity-Guided Merge heuristic was tested with bug `#37458` [5] of Tomcat v5.5. This error consists of a `NullPointerException`, resulting from a data race, and was already used in [9] to test LEAP.

This application bug requires the recording of 15 SPEs, which only account for a total of 61 accesses. This is due to the fact that we use a test unit (JUnit) to trigger the bug, therefore the transformer only instruments the SPEs accessed during the execution of the JUnit class. The use of a driver can be considered as an useful asset, since it allows to circumscribe the really needed SPEs to replay the bug, which is better in terms of scalability (one avoids to instrument all the unnecessary SPEs of the program).

Table III shows the number of attempts of the Similarity-Guided Merge heuristic (using both Plain Similarity and Dispersion-based Similarity) to replay the Tomcat#37458 bug, when recording 25%, 50%, and 75% of the SPEs.

Program	Plain Similiarity			Dispersion-based Similarity		
	25%	50%	75%	25%	50%	75%
Tomcat#37458	2	1	1	1	1	1

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

From the analysis of Table III, it can be verified that our heuristic could easily replay the bug. In fact, one can say that this is a relatively simple error in terms of complexity, as can be proved by the SPE dispersion ratios illustrated in Figure 4. This means that practically all the 500 execution logs collected resulted from production runs originating very similar thread interleavings.

D. Overheads

The Java Grande Forum[6] benchmark contains typically computationally intensive science and engineering applications that require high-performance computers. Given that Java Grande Forum Benchmark does not have known bugs, it was only used in our experiments to assess the benefits and limitations of CoopLEAP when compared to LEAP, on demanding computing environments. Table IV describes the benchmark programs used in terms of number of SPEs and the overall number of times that they are accessed. For the

[4]http://tomcat.apache.org/
[5]https://issues.apache.org/bugzilla/show_bug.cgi?id=37458
[6]http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/

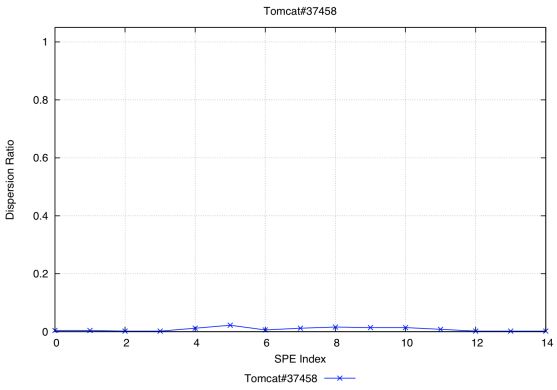

Figure 4. SPE dispersion ratios for the Tomcat#37458 bug, when logging all the SPEs of the program.

sake of readability, the results of the tests performed are presented in tables, since the values obtained vary within a large scale.

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

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

1) Performance Overhead: Table V contains the experiments with respect to the performance overhead measured when tracing the SPEs with the previous logging configurations.

Program	Performance Overhead			
	25%	50%	75%	LEAP
Raytracer	9566.7%	17452.1%	44610.6%	92908.4%
SparseMatmult	598.2%	1725.5%	2505.1%	2606.7%
SOR	1.1%	2.0%	2.4%	2.7%
Montecarlo	1.5%	2.3%	3.7%	7.3%
Series	0.1%	0.4%	2.3%	6.5%

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

One can verify that there is always a decrease of the performance overhead of CoopLEAP when compared to standard LEAP. The most preponderant case is `Series`, where CoopLEAP achieved a runtime degradation 65x and 16x smaller than LEAP, for CLEAP-25% and CLEAP-50%, respectively. These reductions are explained by the fact the majority of the accesses are confined to only two of the eight SPEs of the program. Hence, when those specific SPEs are not traced, the imposed overhead is automatically lower. However, for this program, even the worst case overhead was not very significant.

On the other hand, for `Raytracer`, one can note that both CoopLEAP and LEAP still incur in a heavy performance overhead, as a result of the high number of accesses performed to the SPEs. Nonetheless, once more CoopLEAP brought visible improvements, reducing LEAP penalties by 9.7x, 5.3x, and 2.1x when logging 25%, 50%, and 75% of the SPEs, respectively. This scenario is similar to that of `SparseMatmult`, where CoopLEAP achieved decreases of 4.4x and 1.5x (for CLEAP-25% and CLEAP-50%, respectively) when compared to the runtime degradation of LEAP. This trend also holds for the remaining programs, however with less significant overheads.

2) Log Sizes: Concerning the log size ratios with respect to LEAP, the results are shown in Figure 5.

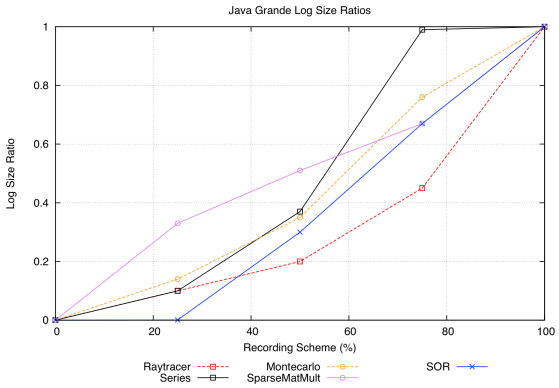

Figure 5. Log size ratios for the Java Grande benchmark programs with reference to log size generated by LEAP (which corresponds to the recording scheme of 100%).

From the figure analysis, the benefits of partial logging are clear. The most evident case is `SOR`, where the log sizes when using CLEAP-25% account for only 0.1% of LEAP's log size. For `SOR` with both CLEAP-50% and CLEAP-75%, the ratios were 0.33 and 0.67, respectively, which is even smaller than the expected. In fact, for all the benchmark programs, there was a high heterogeneity in the size of the access vectors of the program SPEs, which significantly influenced the actual reduction in the log sizes. In other words, the decreases are just not completely linear because some SPEs are accessed more times than others. Given that the instrumentation of the code is performed statically, the load balance in terms of thread accesses may not be equally distributed among the users, as previously referred in Section IV-C. This implies that the impact of logging $x\%$ of the SPEs will not necessarily mean a reduction of $x\%$ in both performance overhead and log size. In fact, sometimes the reduction may be greater than expected (as in `Raytracer` and `SparseMatMult`), but other times may be lower (as in `Series` when using CLEAP-75%). This motivates future research in how one can equally distribute the information to be recorded among the different clients.

VI. Conclusions

This thesis introduced CoopLEAP, a system based on LEAP [9] that provides fault replication of concurrent programs, through cooperative recording and partial log combination. As each user collects only a fraction of the traces, one can further minimize the overhead imposed by logging the original execution. To avoid a brute force approach to find a compatible combination of partial logs, capable of successfully replaying the bug, we also developed a heuristic, denoted Similarity-Guided Merge.

The evaluation study, performed with third-party benchmarks and a real-world application, shown that the benefits from partial recording are clear, as CoopLEAP could always reduce both performance degradations and log sizes produced by LEAP. Furthermore, it was shown that the Similarity-Guided Merge heuristic can successfully replay concurrency bugs by combining information traced by different partial logs. Unfortunately, in the presence of more complex programs (where almost every faulty execution presents a different thread interleaving), the heuristic exhibited limitations in replaying the bug within the maximum number of attempts set.

As future work, one points out the need to further evaluate the bug replay capacity and the performance of CoopLEAP in more complex and realistic scenarios, with also a larger number of partial logs collected. Moreover, new partial logging schemes (e.g. that take into account load balancing) and new similarity metrics (e.g. that use euclidean or edit distances between access vectors) should be studied.

Acknowledgments

This work was partially supported by FCT (INESC-ID multi-annual funding) through the PIDDAC program funds, and by the european project "FastFix" (FP7-ICT-2009-5). Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Paolo Romano, João Garcia, Pedro Louro, and João Matos.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge, "Component-based lock allocation," in *PACT*. IEEE Computer Society, 2007.

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