Property-Driven Cooperative Logging for Concurrency Bugs Replication

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

Abstract

This paper presents *property-driven partial logging*, a novel information partition scheme that takes into account load balancing and shared variable properties to optimize the replay of Java concurrency bugs through partial log combination. Preliminary evaluation with standard benchmarks and a real-world application provides initial evidence of the feasibility of our approach.

1 Introduction

The advent of the multi-core era brought concurrent programming to the forefront of software development. Unfortunately, the inherent non-deterministic nature of concurrent programs makes them much more challenging to write and debug than their sequential counterparts.

For this reason, a large body of research has been devoted to the development of debugging tools that help developers to identify concurrency bugs, i.e. bugs resulting from specific interleaves of thread accesses to shared memory. Deterministic replay has long been suggested as a key technique to debug concurrent programs [9, 3, 6, 8, 4, 15]. This technique is based on the idea of recording non-deterministic events (including the order of access to shared variables, thread scheduling, etc) during a faulty execution and, then, using the resulting trace to force identical outcomes on subsequent runs of the same program, hence reproducing the error.

Unfortunately, the instrumentation overhead introduced by classic deterministic replay approaches [9, 6, 3] can be prohibitive (up to 10x-100x slowdown), making the approach impractical in most settings.

To mitigate this, some recent approaches made efforts to provide efficient replay on commodity multiprocessors by either eliminating the data-races of the program [10] or parallelizing the record and replay using speculative thread execution [16]. In turn, other previous works [2, 17, 14] have exploited the idea of logging only partial execution information during production runs, and of using different exploration techniques, at the maintenance side, to complete the missing data. Unfortunately, these latter techniques can incur a very long inference time, as they require investigating the space of shared memory access interleavings to find the ones suitable to reproduce the concurrency bug. This is a particular important issue because the memory-access interleaving space grows exponentially with the number of threads in the program [13].

In this context, CoopREP [12] recently proposed a, so-called, *cooperative* approach to achieve deterministic replay in Java applications. The key idea at the basis of CoopREP is to obtain a complete, fault-inducing execution log by *i)* leveraging on cooperative partial logging performed by multiple users running the same software application and *ii)* employing statistical techniques to determine which of the available partial logs should be combined (and attempted).

In this paper we investigate an important, but so far unexplored in the context of deterministic replay, design choice at the basis of cooperative logging strategies, which we argue would deserve further investigation from the research community: how to partition the set of shared variables of a program to determine which subset should be traced by each partial log. CoopREP tackles this issue by adopting a pure random approach, which picks a fixed number of shared variables with uniform probability. We argue that a simple random choice comes with a number of shortcomings. First, it provides no deterministic guarantees on the presence of common variables between pairs of partial logs, which hinders the possibility of identifying compatible partial logs via statistical techniques. Moreover, random selection does not allow to capture the semantic correlations among variables, which are usually accessed "together" (i.e., in close points of the execution flow) and, therefore, should be recorded in the same partial log. Also, it does not aim at ensuring uniform load distribution across the users, which may be an issue, as the frequency of accesses to shared variables is highly heterogeneous.

In the light of these considerations, we advocate the need to design new partition schemes, capable of lever-

aging certain properties of interest to optimize the replay of concurrency bugs through partial logging. This paper makes a first step in this direction, by proposing an innovative and practical scheme, called *Property-Driven Partial Logging* (PPL), to automatically create lightweight, overlapping partitions of semantically correlated shared variables. PPL approaches the problem using an integer linear programming (ILP) model that optimizes the generation of partial logs containing correlated information, considering also load balancing and log overlapping. We integrated PPL with CoopREP and conducted a preliminary experimental evaluation based on three standard benchmarks for Java multi-threaded programs and on a well-known open source application server (Tomcat). The results show that PPL reduces the number of attempts to replay the bug in 55% of the cases, while improving the load balancing among the clients by more than 34%. In terms of performance overhead, PPL imposes a runtime degradation between 2.1% and 11.5%, which is up to 3.8x lower when compared to a classic full logging strategy, e.g. LEAP [8].

2 Cooperative Record and Replay

CoopREP [12] is a system that provides support for the replication of concurrency bugs in Java programs, based on cooperative record and replay.

To contextualize and motivate the need for more clever partial logging schemes, we first briefly discuss the two main phases according to which CoopREP operates, namely *i)* cooperative partial logging and *ii)* statistically-guided partial log combination and replay.

2.1 Partial Log Recording

To address memory non-determinism, CoopREP traces the *local* order of thread accesses to the *shared program elements* (SPEs), as previously done by LEAP [8]. Here, the SPEs encompass variables that serve as monitors, as well as other class and field variables that can be concurrently accessed by different threads[1]. To track thread accesses, CoopREP assigns an *access vector* to each different SPE. Thereby, during runtime, every time a thread accesses a shared variable, the thread ID is stored in the SPE's correspondent local-order access vector. For instance, if we have the following access vector $< t_1, t_2, t_2 >$ for a given shared variable x, it means that x was accessed first by thread t_1 and, later, two times by thread t_2. Using this technique, one gets lightweight, local order vectors of thread accesses performed on individual shared variables, instead of requiring a global-order vector.

To minimize logging overhead, CoopREP adopts a cooperative approach in which each user instance records accesses to a subset of the total SPEs of the program during the production run. The subset of SPEs to be traced is defined at instrumentation time, at the developer side. As already mentioned, to define the partition of SPEs to be traced in the partial record versions, CoopREP randomly selects a subset of SPEs corresponding to a given percentage of the total number of SPEs in the program.

[1] In the remaining of this paper, we use the terms *SPE* and *shared variable* interchangeably.

2.2 Partial Log Combination

Although achieving less performance overhead, CoopREP, like other previous partial logging techniques [14, 2], may not guarantee the deterministic reproduction of bugs. In fact, as each instance of the user program only traces a subset of the accesses performed to SPEs, it may be impossible, in some cases, to reconstruct the failure-inducing thread interleaving by combining their partial logs. We note that the problem of identifying a failure-inducing thread interleaving is, generally speaking, a complex one, as the state space that needs to be explored may grow exponentially.

To mitigate this issue, CoopREP applies statistical techniques over the set of collected partial logs to identify those that present more similarities. The goal is to increase the probability of combining partial logs that lead to a feasible thread interleaving, capable of reproducing the error during the replay phase.

To accurately measure the similarity between two partial logs, CoopREP relies on the notion of *overall-dispersion*, which is the ratio between the number of different access vectors logged (across all clients) for a given SPE and the total number of different access vectors collected for all SPEs (across all clients). Overall-dispersion is then used in a metric, denoted *Dispersion-based Similarity* (DSIM), to measure the similarity between two partial logs, l_0 and l_1, as follows:

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

where $Equal_{l_0,l_1}$ (respectively $Diff_{l_0,l_1}$) denotes the set of SPEs with identical (respectively different) access vectors, recorded by both partial logs l_0 and l_1, and *disp(x)* gives the overall-dispersion of SPE x. Note that this metric assigns higher similarity to pairs of partial logs that recorded the same access vector for SPEs with great dispersion.

The DSIM metric is later leveraged by a heuristic, called *Similarity-Guided Merge*, to pinpoint the best partial logs to be the basis to start reconstructing the faulty execution. This heuristic systematically combines similar partial logs to fill the missing access vectors in the basis log, thus generating an entire execution trace. As the merged data may not be compatible, the resulting log is replayed to check whether the bug has been triggered or not. If not, the heuristic selects another base partial

log and repeats the procedure until the bug has been reproduced or the maximum number of stipulated attempts has been reached.

Due to space constraints, we omit the details of the Similarity-Guided Merge heuristic, which is fully described in previous work [12]. Instead, in this paper, we focus on how to create effective and efficient partitions of the SPEs to be logged in the first place.

3 Limits of Random Selection Schemes

The random scheme adopted by CoopREP to determine the variables traced by each partial log is attractive due to its simplicity, but suffers also from a number of relevant shortcomings:

i) Log Overlapping: Random partial logging, although often effective, does not guarantee that two partial logs (acquired at different users) necessarily overlap. The drawback of this is that, if two partial logs have no SPEs in common, it is impossible to deduce whether these partial logs were traced from equal executions and, consequently, if they are suitable to be combined.

ii) Load Balancing: The ultimate goal of cooperative record and replay is to minimize the time and space overhead imposed during production runs. Although partial logging *per se* helps to achieve this goal, randomly selecting the SPEs to be traced may result in some users being assigned with the shared variables most frequently accessed and other users with the least accessed ones, thus providing a poor overall load balance.

iii) SPE Correlation: To optimize the combination of compatible information, one should trace SPEs that depend on one another in the same partial log. The rationale is that, during the partial log merging phase, we can avoid having to combine dependent information collected from potentially different production runs. Random partial logging, however, generates SPE partitions disregarding this property.

The above considerations raise the following question: *is it possible to design solutions capable of addressing the limitations of random strategies, and to identify effective and efficient partitions of the SPEs in a practical and scalable fashion?*

We argue that this problem presents two main challenges: the first one is related to the need to ensure the practical tractability of the partitioning model, as its combinatorial nature may easily lead to the formulation of solutions with prohibitive computational costs. A second challenge is related to how to encode the above-mentioned objectives into a global optimization problem in a coherent and sound fashion.

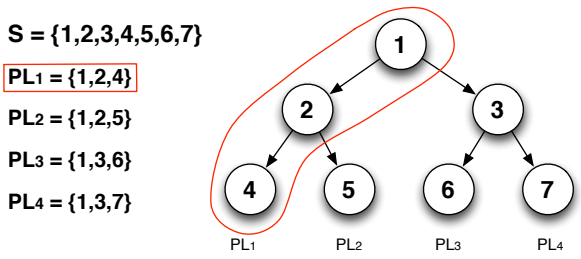

Figure 1: Partial logs generated using the tree quorum technique for a program with 7 SPEs.

4 Property-Driven Partial Logging

In this section we make a first step to answer the above question, by presenting *Property-driven Partial Logging* (PPL), a novel algorithm that casts the problem of selecting the SPEs to be traced by different partial logs as a linear optimization that aims at jointly pursuing the following goals: *i)* maximizing correlation among the SPEs logged by the same partial log, *ii)* ensuring the presence of at least one SPE in common between any two partial logs, and *iii)* ensuring a bounded unbalance level for the logging overhead imposed by recording any two partial logs.

Next, we describe how each of the three objectives mentioned above are encoded in PPL. Finally, in Section 4.4, we discuss how they are combined into a unique optimization problem solvable using linear programming.

4.1 Log Overlapping

To meet the overlapping requirement, we build partial logs in such a way that they form a *coterie* (i.e. a set of quorums $\mathscr{Q}$, such that for any $Q, Q' \in \mathscr{Q} : Q \not\subset Q'$). For this, we rely on an adaptation of the tree quorum technique [1]. This technique takes a universe of elements $\mathscr{U} = \{1,...,n\}$ and builds a binary tree. A quorum is then given by any path from the root to a leaf, and the coterie is the set of all these paths. Therefore, we can conclude that, in order to have a coterie of partial logs, it suffices to have $2^{\lfloor log_2 S \rfloor}$ partial logs (where S is the total number of SPEs of the program), each one containing $\lfloor log_2 S \rfloor + 1$ variables.

Figure 1 illustrates an example of this technique applied to a program with 7 SPEs ($S = \{1,2,3,4,5,6,7\}$). In this case, to obtain a coterie of partial logs, we require four partial logs, each one with three SPEs. A possible configuration is to have SPE 1 as root, as depicted in the figure. Given that this SPE is going to be traced by all partial logs, one guarantees that any two partial logs will overlap. However, it should be noticed that, for partial log overlapping, we do not require the tree to be binary, but only that there exists a common root. For instance, for the example in Figure 1, a partial log configuration such as $PL_1 = \{1,2,3\}, PL_2 = \{1,2,4\}, PL_3 = \{1,2,5\}$,

and $PL_4 = \{1, 6, 7\}$ would also be feasible.

4.2 Load Balancing

The load associated to a partial log is related to the size of the access vectors traced for its SPEs. Hence, the higher the number of times a SPE is accessed, the greater its access vector will be and, consequently, the stored log.

To achieve load balance, we force an upper bound in the difference allowed between the amount of SPE accesses traced by different partial logs. This upper bound, denoted *MaxLoadDiff*, is computed as follows:

$$MaxLoadDiff = \phi WorstCase$$

where *WorstCase* corresponds (for a given partial log capacity *cap*) to the load difference between two extreme partial log configurations: one recording the *cap* "heavier" SPEs and the other tracing the *cap* "lighter" SPEs. In turn, the parameter $\phi \in [0, 1]$ allows to tune the *tightness* of that load variation. Hence, a value of ϕ close to 0 will indicate that the difference between the number of SPE accesses traced by the partial logs should be minimal, thus ensuring good load balance.

Unfortunately, the information regarding the size of each SPE access vector is only accurately known at runtime. To address this, we perform an off-line "training phase", during which we trace the frequency of accesses to SPEs for a given number of successful executions, thus obtaining an approximate value for each SPE's load.

4.3 SPE Correlation

The term *correlation* is used in this context (analogously to the classic definition used in statistics) to measure to what degree two random variables satisfy probabilistic independence. In the particular case of software, one can observe that dependence relationships among program variables are ubiquitous. In fact, it is common for developers to express program semantic aspects using correlated variables, even if unconsciously. For instance, developers may rely on variable correlation to represent correlated real-world entities; use a variable to define another variable's state, properties or constraints; or use multiple variables to describe different aspects of a complex object [11]. For this reason, correlated variables are usually updated and accessed together during the program execution. Thereby, if we are able to capture these access dependences among variables, we can get a better insight on which variables should be tracked together and on how to effectively partition the SPE set into subsets to be logged by different users. The goal is to later facilitate the deterministic replay through partial log combination.

We achieve this in two steps. First, we infer SPE correlations and compute the *correlation coefficient* for each SPE (i.e. its inter-dependences with all the other SPEs of the program). Second, we maximize the sum of the correlation coefficients per partial log through an ILP model.

In the following we describe how SPE correlation inference is performed, whereas the ILP model is further explained in Section 4.4.

4.3.1 Correlation Inference

We infer correlations among SPEs by employing a technique similar to that of MUVI [11]. MUVI focuses on extracting multi-variable access correlations through the analysis of variable access patterns and the examination of what variables are usually read or written together. However, MUVI aims at localizing semantic and multi-variable concurrency bugs, while we are interested in replaying the faulty execution rather that detecting the error. Despite that, we rely on the same notion of *togetherness* to claim that two shared variables are usually accessed together and, thus, should be considered correlated. More formally, we say that two SPEs are *together* if they are accessed in the same method with less than *MaxDistance* statements apart, where *MaxDistance* is a parameter defined by the developer.

We also define *variable correlation* for two SPEs x and y as follows: x is correlated with y (written as $x \Rightarrow y$), iff x and y are accessed *together* at least *MinSupport* times and whenever x appears, y appears together with at least *MinConfidence* probability, where *MinSupport* and *MinConfidence* are adjustable thresholds.

The inference of the SPE correlations is then carried in four steps:

i) Collect SPE access information. We perform a static analysis of the source code to collect information regarding accesses to SPEs within the same method. For each access, we collect a tuple containing: the id of the SPE, the name of the method, the line number of the SPE access instruction, and a flag indicating whether the access is from a method itself (direct access) or its callee methods (indirect access). All these tuples are then stored in a database for each method, denoted *AccSet*. The need for collecting indirect accesses arises due to encapsulation, where variables are often read or written inside utility methods, such as `get()` and `set()`.

ii) Identify frequent together SPEs. After obtaining the *AccSets* for each method, we need to identify pairs of SPEs that appear together a *MinSupport* number of times, in order to be considered as possibly correlated. Given that computing these patterns can be expensive, we employ a widely-used frequent itemset mining algorithm, called *FPGrowth*[7]. FPGrowth is tailored to, given a database where each entry is a itemset (i.e. a set of items), efficiently pinpoint which sub-itemsets appear in more than a minimum number (denoted *MinSupport*) of entries.

By applying FPGrowth to our *AccSet* we then obtain the different subsets of SPEs that are accessed together

in more than *MinSupport* methods. However, for PPL, we are only interested in the subsets containing two elements. We call these subsets *candidate pairs*.

iii) Generate and prune SPE correlations. In order to determine whether a candidate pair is a valid correlation or a false positive, we rely on the following two metrics:

Support: The support of a correlation candidate pair $P : x \Rightarrow y$, denoted as *support(P)*, is given by the number of methods in which x and y are accessed together (according to the definition of togetherness). If this number is smaller than *MinSupport*, the pair is pruned out.

Confidence: The confidence of a correlation candidate pair $P : x \Rightarrow y$ gives the conditional probability that y is accessed in a method, given that x is also accessed in the same method. It is computed as $\frac{support(P)}{support(x)}$, where *support(x)* is the number of methods that access x.

Confidence is important to assess the accuracy of the correlation, because a low confidence value makes a candidate pair untrustworthy, despite having many support methods. Hence, if the confidence value is lower than *MinConfidence*, the candidate pair is pruned out. On the other hand, the candidate pairs that remain after the sieving are considered as valid correlations.

iv) Compute SPE correlation coefficient. For a given SPE s, let V_s be the set of all valid correlations between s and any other SPE y of the program (i.e. $\forall v \in V_s, v : s \Rightarrow y$). The *correlation coefficient* of SPE s is then computed as $\sum_{v \in V_s} Confidence(v)$ and allows to capture the overall dependence of each shared variable with respect to the other variables. In other words, a SPE with high correlation coefficient means that its thread accesses are directly related with the accesses to several other SPEs, which makes it a good candidate to serve as overlapping point between different partial logs.

4.4 Property-Driven SPE Selection

As already mentioned, PPL combines the three aspects discussed above into a global optimization problem, which is described as follows.

Let $\mathscr{S} = \{1,...,s\}$ be the universe of SPEs in the program, $l = 2^{\lfloor log_2 s \rfloor}$ the number of partial log configurations to be created, and $cap = \lfloor log_2 s \rfloor + 1$ the capacity of each partial log. Let us also consider a weight vector $W^{1\times s} = (w_j), j \in \{1,...,s\}$, that gives the load associated with each SPE$_j$ in terms of the number of times it is accessed, and *MaxLoadDiff* as the maximum load difference allowed between each two partial logs (see Section 4.2). Finally, let $C^{1\times s} = (c_j), j \in \{1,...,s\}$, be a vector containing the *correlation coefficient* for each SPE j (see Section 4.3.1.iv). The ILP model is then formulated as follows.

$$max \ \sum_{i=1}^{l} \sum_{j=1}^{s} x_{ij}.c_j$$
subject to:

$$\sum_{j=1}^{s} x_{ij} = cap, \forall i \in \{1,...,l\} \quad (i)$$

$$\sum_{i=1}^{l} x_{ij} \geq 1, \forall j \in \{1,...,s\} \quad (ii)$$

$$|\sum_{j=1}^{s} x_{ij}.w_j - \sum_{j=1}^{s} x_{i^*j}.w_j| \leq MaxLoadDiff, \forall i, i^* \in \{1,...,l\}, i \neq i^* \quad (iii)$$

$$x_{ij} \in \{0,1\}$$

where $X^{l\times s} = (x_{ij})$ is the unknown binary variable matrix (x_{ij} indicates whether partial log i contains SPE j or not). Notice that constraints $(i), (ii)$, and (iii) ensure fixed capacity per partial log, SPE covering, and load balancing, respectively.

This is a maximization problem, so we know that an optimal solution will always try to "pack" the SPEs having higher correlation coefficient. However, as the SPE covering constraint limits this selection by imposing that each SPE must be packed at least once, this will cause the partial logs to be populated in a tree quorum fashion. Assuming that *MaxLoadDiff* is chosen wisely to allow the problem to be solvable, the SPE with the greatest correlation coefficient will then be the root of the tree, thus being present in every partial log.

5 Evaluation

To assess the benefits and limitations of the proposed scheme, we implemented PPL over our CoopREP prototype, using JavaILP[2] with lp_solve to solve the ILP problem. We then compared our approach against an unbalanced PPL approach, as well as against SPE random partitioning (assuming the same capacity per partial log), according three main criteria:

- **Effectiveness:** which scheme allows CoopREP to replay the concurrency bug in less attempts, when varying the number of partial logs collected?
- **Load Balancing:** which scheme provides better load distribution among the users?
- **Performance Overhead:** does PPL incur greater performance overhead with respect to a random scheme?

Program	SLOC	#SPE	#Total Accesses	#Partitions Generated	Partition Capacity
TwoStage	170	4	27103	4	3
TicketOrder	216	8	22470	8	4
Piper	260	6	347	4	3
Tomcat#37458	535K	21	72	16	5

Table 1: Benchmarks description.

[2] http://javailp.sourceforge.net

Program	Random						PPL ($\phi = 0.7$)						Unbalanced PPL					
	16	32	64	128	256	512	16	32	64	128	256	512	16	32	64	128	256	512
TwoStage	7	7	1	1	1	1	**1**	**1**	1	1	1	1	**1**	**1**	1	1	1	1
TicketOrder	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	**3**	**3**	**1**	**1**	*X*	*X*	**3**	**3**	**1**	**1**
Piper	*X*	6	1	1	1	1	**3**	**1**	1	1	1	1	**3**	**1**	1	1	1	1
Tomcat#37458	*X*	9	8	8	1	1	*X*	**2**	**2**	**2**	1	1	*X*	**2**	**1**	**1**	1	1

Table 3: Attempts required by CoopREP to replay the benchmark bugs using the three partial logging schemes when varying the number of logs collected. The *X* indicates that the heuristic failed to replay the bug in the maximum number of attempts. The shaded cells highlight the cases where PPL schemes outperformed random partial logging.

Program	%Load Difference
TwoStage	0.8%
TicketOrder	0.01%
Piper	1.1%
Tomcat#37458	2.7%

Table 2: Difference between the load of the logs recorded from both 100 successful and 100 failed executions.

5.1 Methodology

For the experiments, we used three programs from the ConTest benchmark suite [5] and the real-world application *Tomcat* (bug #37458). These programs are described in Table 1 in terms of their source lines of code (SLOC), number of SPEs, and the total amount of SPE accesses, respectively. The two final columns indicate how many partitions should the PPL schemes create and how many SPEs should each partition contain.

As a remark, it should be noted that failure rate of the benchmarks that we observed empirically was 80%, 7%, 5%, and 4%, respectively for *TwoStage*, *TicketOrder*, *Piper*, and *Tomcat*. This means that, in general, these bugs are not easily triggered without the help of deterministic replay techniques.

Regarding the training phase, it was conducted with 100 logs from successful executions. Table 2 reports the percentage of load difference between the access vectors of those logs and the access vectors of another 100 logs captured from failed runs. As it can be verified, the frequency of accesses to the SPEs is almost identical in both correct and faulty executions. This allows to conclude that using an off-line training phase to obtain an approximate value for each SPE's load does not significantly affect the accuracy of the ILP model.

Finally, concerning parameter setting, for correlation pruning we used *minSupport* = 2 (some benchmark programs do not have many methods) and *minConfidence* = 0.5. For CoopREP, the Similarity-Guided Merge heuristic [12] was executed using *DSIM* metric (see Section 2.2) with a similarity threshold of 0.01 and a maximum number of attempts to reproduce the bug of 512. To get a fairer comparison of the recording schemes, the partial logs were generated from complete logs, picking the SPEs to be stored according to each scheme's policy.

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.

5.2 Effectiveness

The effectiveness of the partial logging schemes are directly related to CoopREP's bug replay capacity. In other words, an effective partitioning scheme should allow CoopREP to successfully reconstruct the faulty execution within a few tries, even when collecting a small number of partial logs from the user instances.

To assess of the effectiveness of PPL, we compared the number of attempts required to reproduce the bug when using partial logs generated by this scheme (considering $\phi = 0.7$ when computing *LoadMaxVar*[3], see Section 4.2), as well as by both an unbalanced PPL approach and random partial logging. For each case, CoopREP was executed with N generated partial logs, where $N \in \{16, 32, 64, 128, 256, 512\}$. Table 3 reports the outcomes of the experiments.

The results show that property-driven schemes clearly outperform random partial logging (the number of attempts to replay the bug is smaller in 55% of the cases), thus supporting our claim that leveraging log overlapping and SPE correlation when recording production runs optimizes the statistical-guided partial log combination.

In particular, *TicketOrder* was the benchmark for which the benefits of PPL are most evident. In this program, the bug was triggered due to unsynchronized accesses to two correlated shared variables. As the random scheme does not capture correlation and these two particular SPEs exhibited a different thread interleaving in all the traced production runs (thus being irrelevant to measure similarity between partial logs), CoopREP ends to choose, as a basis to start reconstructing the complete execution, partial logs where at least one of this SPEs is missing. Conversely, by using PPL schemes, the relevant SPEs are recorded together, thus facilitating the task of merging compatible information.

[3] We found $\phi = 0.7$ to be the most suitable value from a sensitivity analysis that evaluated the effect of varying ϕ on CoopREP's bug replay capacity, for 512 partial logs. We omit this analysis due to space constraints.

5.3 Load Balancing

In order to quantify the load distribution benefits achievable via load balanced PPL with respect to unbalanced PPL and random partial logging, we measured the load difference between the maximum and the minimum load for sixteen partial logs generated with the three schemes. Figure 2 depicts the obtained results.

As expected, the load upper bound imposed in the formulation of the integer linear programming for property-driven partial logging always resulted in smaller disparities between the partial logs with the most and the least load, respectively. However, surprisingly, the unbalanced PPL approach never exhibited worse load balance than random partial logging. This can be explained by the fact that, for unbalanced PPL, despite removing the constraint for load distribution in the linear programming problem, the SPEs partitions are still generated in a quorum fashion, thus promoting the tracing of similar SPEs among different partial logs. In fact, the partial logs configurations produced by both PPL approaches are equal for programs *TicketOrder*, *Piper*, and *TwoStage*, as highlighted by the results in both Table 3 and Figure 2.

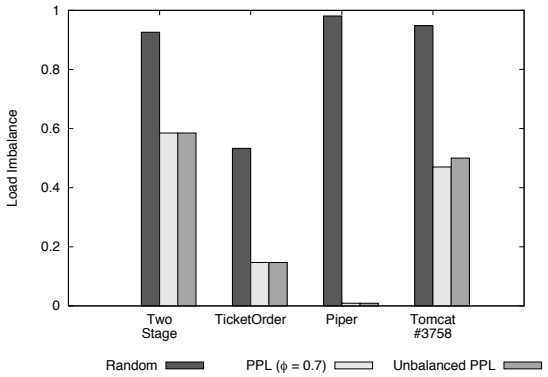

Figure 2: Load imbalance for the three schemes, normalized for the *WorstCase* (see Section 4.2).

5.4 Performance Overhead

Figure 3 depicts the performance overhead due to instrumentation. As one can see, the three partial logging schemes incur a runtime degradation between 2.1% and 11.5% on average, being up to 3.8x lower when compared to LEAP [8], a classic full logging strategy.

Comparing now the partial logging strategies among each other, Figure 3 shows that the random scheme slightly outperforms PPL schemes for programs *TwoStage* and *TicketOrder*, whereas the opposite scenario is verified for benchmarks *Piper* and *Tomcat*. This is explained by the fact that, since performance overhead is only determined by the frequency of accesses to the SPEs traced, PPL may identify either heavy or light SPEs as root of the quorum tree (depending on whether they are very correlated to other SPEs or not). Despite that, for all cases, one can see that the random strategy always exhibits greater difference between the maximum and the minimum values of performance degradation when compared to PPL approaches.

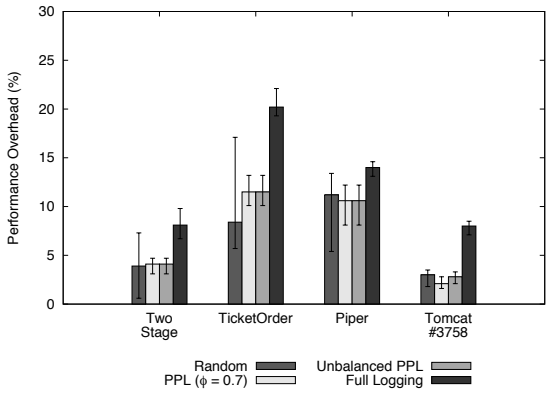

Figure 3: Performance overhead for the three partial logging schemes, as well as for a full logging strategy [8] (the error bars indicate the maximum and the minimum values observed).

6 Conclusions

In this paper, we shed light on the importance of exploring an important design choice for cooperative record and replay strategies, that is how to select the shared variables to be assigned to different partial logs. We also make a first step in this direction by proposing *property-driven partial logging* (PPL), a novel information partition scheme that automatically creates load balanced, overlapping partitions of semantically correlated shared variables. Early results highlight the relevance of this design choice both in terms of increased ability to replay concurrency bugs (the number of attempts to reproduce the bug is smaller in 55% of the cases) and load distribution among the clients (the load balance is improved by more than 34%). In terms of performance overhead, PPL imposes a runtime degradation between 2.1% and 11.5%, which is up to 3.8x lower when compared to a classic full logging strategy, e.g. LEAP [8].

We believe that PPL paved the way for future works aimed at building more elaborate solutions (e.g. by combining shared variable tainting with correlation).

7 Acknowledgments

The authors wish to thank the anonymous reviewers for the valuable feedback and suggestions. This work has been partially supported by FCT (INESC-ID multi-annual funding) through the PEst-OE/EEI/LA0021/2011 Program Funds.

References

[1] AGRAWAL, D., AND EL ABBADI, A. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 1–20.

[2] ALTEKAR, G., AND STOICA, I. Odr: output-deterministic replay for multicore debugging. In *ACM SOSP* (2009), pp. 193–206.

[3] CHOI, J.-D., AND SRINIVASAN, H. Deterministic replay of java multithreaded applications. In *ACM SPDT* (1998), pp. 48–59.

[4] DUNLAP, G., LUCCHETTI, D., FETTERMAN, M., AND CHEN, P. Execution replay of multiprocessor virtual machines. In *ACM VEE* (2008), pp. 121–130.

[5] FARCHI, E., NIR, Y., AND UR, S. Concurrent bug patterns and how to test them. In *IEEE IPDPS* (2003), pp. 286–293.

[6] GEORGES, A., CHRISTIAENS, M., RONSSE, M., AND DE BOSSCHERE, K. Jarec: a portable record/replay environment for multi-threaded java applications. *Software Practice and Experience 40* (May 2004), 523–547.

[7] HAN, J., PEI, J., AND YIN, Y. Mining frequent patterns without candidate generation. In *ACM SIGMOD* (2000), pp. 1–12.

[8] HUANG, J., LIU, P., AND ZHANG, C. Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *ACM FSE* (2010), pp. 385–386.

[9] LEBLANC, T., AND MELLOR-CRUMMEY, J. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36* (April 1987), 471–482.

[10] LEE, D., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Chimera: hybrid program analysis for determinism. In *ACM PLDI* (2012), pp. 463–474.

[11] LU, S., PARK, S., HU, C., MA, X., JIANG, W., LI, Z., POPA, R., AND ZHOU, Y. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM SOSP* (2007), pp. 103–116.

[12] MACHADO, N., ROMANO, P., AND RODRIGUES, L. Lightweight cooperative logging for fault replication in concurrent programs. In *IEEE DSN* (2012), pp. 1–12.

[13] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *ACM PLDI* (2007), pp. 446–455.

[14] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K., AND LU, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *ACM SOSP* (2009), pp. 177–192.

[15] SRINIVASAN, S., KANDULA, S., ANDREWS, C., AND ZHOU, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference* (2004), pp. 29–44.

[16] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P., FLINN, J., AND NARAYANASAMY, S. Doubleplay: parallelizing sequential logging and replay. In *ACM ASPLOS* (2011), pp. 15–26.

[17] ZAMFIR, C., AND CANDEA, G. Execution synthesis: a technique for automated software debugging. In *ACM EuroSys* (2010), pp. 321–334.