Towards Effective and Efficient Search-Based Deterministic Replay

Manuel Bravo, Nuno Machado, Paolo Romano, Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{angel.gestoso,nuno.machado,ler}@ist.utl.pt, romano@inesc-id.pt

ABSTRACT

Deterministic replay tools are a useful asset when it comes to pinpoint hard-to-reproduce bugs. However, no sweet spot has yet been found with respect to the trade-off between recording overhead and bug reproducibility, especially in the context of search-based deterministic replay techniques, which rely on inference mechanisms.

In this paper, we argue that tracing the locking order, along with the local control-flow path affected by shared variables, allows to dramatically reduce the inference time to find a fault-inducing trace, while imposing only a slight increase in the overhead during production runs. Preliminary evaluation with a micro-benchmark and third-party benchmarks provides initial evidence that supports our claim.

Categories and Subject Descriptors

D.2.5 [**Software Testing and Debugging**]: Debugging aids; Tracing

General Terms

Algorithms, Reliability, Performance

Keywords

concurrency, deterministic replay, symbolic execution

1. INTRODUCTION

1.1 Motivation

Parallel applications are difficult to design and implement, but debugging their code can be even more challenging. To start with, many concurrency errors (called *heisenbugs*) may only appear when specific thread interleaving occur, and these interleavings may be very hard to experience during test runs. Furthermore, classical debugging techniques, such as cyclic debugging, fall short for heisenbugs. In fact, repeating the faulty execution several times in an attempt to narrow down the root cause of the bug is not effective when the error-inducing interleaving happens only in a few runs. As a result, concurrent programs including widely used applications such as MySQL, Apache, Mozilla, and OpenOffice [9] are commonly deployed with bugs.

Deterministic replay techniques have been designed to overcome these obstacles. The goal is to log enough information during the production run of the application such that, if an error occurs, the development team can later reproduce the faulty run and re-enable cycling debugging. More concretely, deterministic replay works by, first, capturing as many non-deterministic events as possible during the production run to, then, use the log to enforce the reproduction of buggy execution.

Since the number of non-deterministic events can be extremely large, and the application may be required to execute for a long period of time before a bug is found, the main challenge of deterministic replay is to be able to reproduce the bug while performing logging with small spatial and time overhead. The early deterministic replay solutions, that can be named *order-based*, have been designed to reproduce the bug on the first attempt [3, 6, 7, 14], as they track every read and write access performed in shared memory locations. Although this provides guarantees on the successful replay of the bug, these solutions induce a high spatial and time overhead during the recording phase. To mitigate the runtime slowdown, alternative approaches have been recently attempted. In particular, *search-based* solutions avoid recording all memory non-deterministic events and rely on a post-recording inference phase to compute, starting from a partial log, a feasible fault-inducing trace [15, 12, 1, 10, 8, 16]. The drawback is that replay determinism is not guaranteed, mainly because the lack of information on the exact interleaving of threads shared accesses typically causes an exponential growth of the search space during replay [11].

1.2 State of the Art

In this position paper, we argue that, despite the valuable progress that has been made towards more effective and efficient search-based deterministic replay techniques, no sweet spot has yet been found that poses a good balance between recording overhead and replay guarantees. In particular, we are concerned with the following question: *are there any methods that can dramatically reduce the inference time of current solutions, while imposing the same or a slightly greater recording overhead during production runs?* Figure 1 depicts this goal[1] by plotting qualitatively some of

[1]Note that Figure 1 is not based on new measurements, but rather on an approximate qualitative analysis regarding the

HotDep - Workshop on Hot Topics in Dependable Systems '13, Nov. 3, 2013, Farmington, Pennsylvania, USA

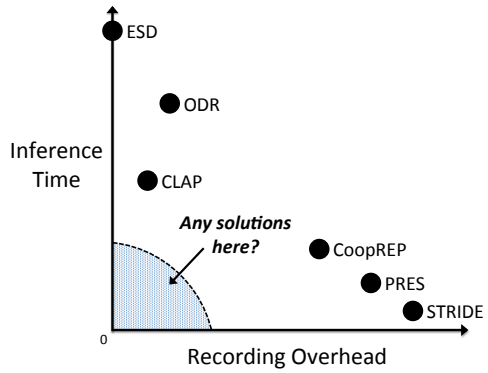

Figure 1: Tradeoff between inference time and recording overhead for current search-based deterministic replay solutions.

the state of the art approaches for search-based deterministic replay.

On one hand, ESD [15], ODR [1], and CLAP [8] achieve low recording overhead by not tracing any information with respect to the order in which threads access to shared variables at runtime. Instead, during the search phase, these solutions rely on symbolic execution and constraint solving to infer the buggy interleaving.

In particular, ESD does not perform any recording at all and strives to infer an execution that reproduces the original failure by analyzing the program symbolically with hints extracted from the bug report. In turn, ODR traces both the program outputs and inputs, the execution path, and the locking order to generate constraints that ease the search of a thread interleaving capable of providing the same output as the original run. CLAP improves ODR both by parallelizing the constraint solving phase and by focusing on matching shared reads to writes, rather than inferring concrete values for the operations over shared variables.

On the other hand, PRES [12], CoopREP [10], and Stride [16] incur smaller inference times, at a cost of recording, respectively, the global execution order of every basic block containing shared variables (PRES-BB), the local order of access to a subset of shared program elements, and a relaxed order between shared memory reads and writes.

1.3 Our Claim

In this paper, we move a step forward towards more effective and efficient search-based deterministic replay techniques. Concretely, we argue that, in search-based solutions, to trace the synchronization points can dramatically decrease the time required for the inference phase, while incurring only a slight increase in the logging overhead. To illustrate this point, we have extended CLAP [8], a search-based solution that, from our perspective, offers the best balance among the recording overhead, inference time, and the information that is provided to the developers (however, other approaches, such as ESD, can also benefit from our idea). Preliminary experiments with a prototype implemented in Java support our claim: the inference time to find a feasible buggy interleaving is reduced around 93x on average, at a cost of increasing recording overhead by up to 27%.

tradeoff between logging non-deterministic events and post-recording search time, according to published results.

2. CLAP OVERVIEW

CLAP [8] is a search-based deterministic replay system for C/C++ multithreaded programs, which aims at reducing the recording overhead via inference mechanisms (namely symbolic execution and constraint solving). Like other search-based record and replay solutions, CLAP consists of three main phases: the recording phase, the inference phase, and the replaying phase. The three phases are further described in the following.

Record Phase. Instead of capturing information regarding thread interleaving during the production run, CLAP traces the path that each thread executes locally. The purpose of tracing the control-flow choices is to guide the symbolic execution directly through the fault-inducing path, thus avoiding the need to explore all the possible executions paths. This approach accelerates the inference phase since there is no need to backtrack to previous states, as ESD [15] does.

Inference Phase. This phase aims at computing a global execution that follows the thread local path previously recorded and is capable of triggering the error deterministically. The inference phase can be divided in the following sub-phases:

a) Symbolic Execution. CLAP uses a concolic (concrete + symbolic) execution, in the sense that only operations over shared variables create fresh symbolic symbols, whereas operations on other variables have concrete values as in a normal execution. Hence, during the symbolic execution, there is no need to care about the data races since no concrete values are assumed when reading from a shared variable. In fact, the main goal of this phase is to gather enough information for encoding a set of execution constraints representing the buggy run.

b) Formula Generation. Once the symbolic execution has finished, CLAP generates a formula composed by constraints that represent the execution. The purpose of this formula is to model the problem of finding the buggy schedule as a constraint solving problem. The solution of this problem will then indicate the order in which threads should access shared memory positions such that the fault is reproduced. In particular, the formula generated by CLAP is as follows.

$$\phi = \phi_{path} \wedge \phi_{bug} \wedge \phi_{so} \wedge \phi_{rw} \wedge \phi_{mo}$$

where:

- ϕ_{path} denotes the path constraints. These are gathered by the symbolic execution. Each constraint represents one symbolic branch decision, that consists of an *if* statement in which, at least, one operand is symbolic.
- ϕ_{bug} represents the bug constraint. It is usually an assertion of the condition that must be verified in order to manifest the bug. For instance, *division by zero* exception in the instruction $\mathsf{val} = 1/\mathsf{x}$ can be defined as the following read constraint $R_x == 0$.
- ϕ_{so} denotes the synchronization order constraints, which are divided in two types, namely *locking constraints*

and *partial order constraints*. The former are concerned with *lock* and *unlock* operations, and stipulate that two blocks of read/write operations protected by the same lock should not be interleaved. In turn, partial order constraints are related to *start, exit, fork, join, wait*, and *signal* operations. These constraints further help to determine the happens-before relation between two shared access points (i.e. a read, write, or synchronization).

- ϕ_{rw} are called read-write constraints. They represent the linkage between read and write operations (potentially from different threads) and restrict the execution order of those instructions. In particular, read-write constraints can be written as follows:

$$(V_r = init \bigwedge_{\forall w_j \in W} O_r < O_{w_j})$$
$$\bigvee_{\forall w_i \in W} (V_r = w_i \wedge O_{w_i} < O_r \bigwedge_{\forall w_j \neq w_i} O_{w_j} < O_{w_i} \vee O_{w_j} > O_r)$$

  where V_r is the value returned by a read r on a given shared variable s, *init* the initial value of s, and W the write set for s. In turn, O_r and O_w determine, respectively, the order of r and the order of the write w_i in W.

  Thereby, if the value read by r is *init*, then r must precede all write operations $w_i \in W$ (i.e. $O_r < O_{w_i}, \forall w_i \in W$). Otherwise, r must follow a particular write w_i (i.e. $\exists w_i \in W : O_{w_i} < O_r$) and the read value is the result of that write ($V_r = w_i$). Finally, the last condition states that no other write is between them.

- ϕ_{mo} denotes the memory-order constraints, i.e. the order in which instructions are executed in a specific thread, according to the memory consistency model. For sequential consistency, for instance, if a thread executes r_i prior to w_b and after w_a, the corresponding memory order constraint will be: $O_{w_a} < O_{r_i} < O_{w_b}$. However, CLAP not only supports the sequential consistent memory model, but also both partial store ordering and total store ordering (in case they are supported by the hardware).

c) Constraint Solving. CLAP does not try to feed a constraint solver with the whole formula. Instead, it first leverages a technique, denoted preemption-bounding, to generate a set of candidate schedules with an increasing number of thread context switches, that also satisfy memory order constraints (ϕ_{mo}). For each candidate schedule, the context switches are then translated into ordering constraints and added to the global formula. This scheme allows not only to bound the search space (and generate schedules with a minimal number of context switches), but also to speed up the formula constraint solving, as CLAP can fork a new process to validate each single candidate schedule. Also, minimizing the number of thread preemptions might further help developers to find the cause of the bug since a thread schedule with few context switches should be easier to understand than a thread schedule with more context switches.

Replay Phase. Once the solver has found a solution, the produced thread interleaving is used to enforce the bug reproduction. In practice, CLAP runs a previously instrumented version of the program that, before each shared memory access, checks the schedule to see whether the running thread is the one that should execute the operation at that moment. If it is, the execution proceeds normally, whereas if it not, the thread is blocked and waits until its turn.

3. BENEFITS OF TRACING SYNC ORDER

Our work is motivated by the insight that the inference phase of search-based replayers such as CLAP can be dramatically improved by tracing the program locking order, i.e. the global order of synchronization operations, such as *wait/signal* or *lock/unlock*. The rationale for this approach is the following. First, previous work has shown that recording the thread access order to synchronization points comes with a relatively low overhead during the production run [1, 13]. Second, it allows to remove the synchronization order constraints (ϕ_{so}) from CLAP's global formula, which corresponds to discarding a cubic formula in the number of *lock/unlock* pairs per lock object. This also results in a substantial decrease of the number of solutions to be checked, as the correct locking order is known *a priori* and, therefore, does not have to be inferred. In fact, according to our experiments, the first solution suggested by the solver always corresponds to a buggy schedule.

In the following, we present a simple example to better understand the impact of tracing the synchronization order in CLAP's constraint model.

3.1 Example

Let us consider CLAP* as a version of CLAP that records the locking order at runtime, in addition to the execution path. Figure 2.a) shows a simple example that helps to visualize the benefits of CLAP* in comparison to CLAP. It represents a multithreaded program that contains both synchronized and non-synchronized accesses to a shared variable x. Assuming that the production run follows the execution depicted by the arrows, CLAP* will track that thread t_1 acquired the lock before thread t_2 (this can be easily done by inserting a *probe* right after the lock operations, thus avoiding the need for extra synchronization). Conversely, CLAP will not record anything, since the program does not contain any conditional instructions.

Afterwards, during the symbolic execution phase, shared read/write accesses are identified and logged. These opera-

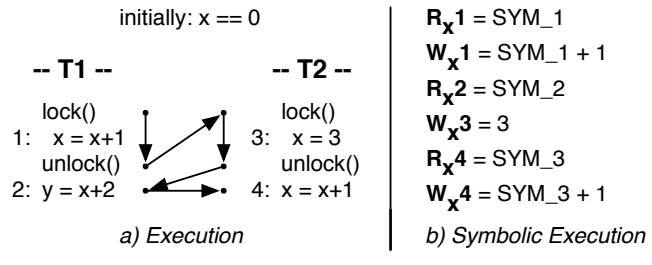

Figure 2: Simple multithreaded program. R_xn (W_xn) **denotes a symbolic read (write) operation on shared variable x in line** n**.**

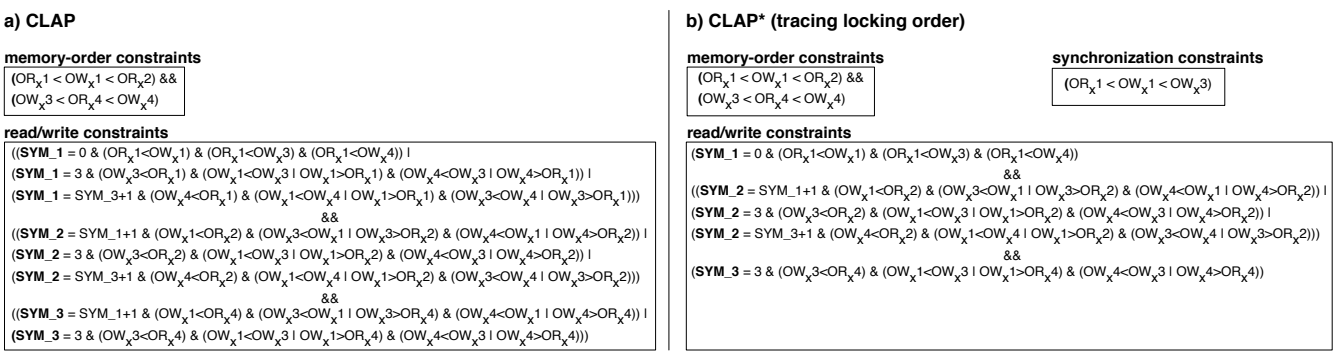

Figure 3: Constraints generated by CLAP and CLAP* for the example program shown in the Figure 2.

tions are depicted in Figure 2.b), where R_xn (W_xn) denotes a read (write) operation over the shared variable x in line n. In particular, each shared read creates a new fresh symbolic symbol (e.g. the operation R_x1 creates the symbolic symbol SYM_1).

Once the symbolic execution has finished, both CLAP and CLAP* generate a set of constraints that helps to infer the original execution. Figure 3 presents a simplified version of the formula that CLAP and CLAP* would generate for this example. Here, OR_xn denotes the order of the corresponding operation (R_xn) in the to-be-computed thread schedule. In turn, memory-order constraints represent the order in which the operations have been locally executed by each thread.

Figure 3 shows that the read-write constraints produced by CLAP* are simpler than the constraints generated by CLAP. For instance, according to CLAP's constraint model, SYM_1 can be equal to 0, 3, or SYM_3+1, while for CLAP* SYM_1 can only be equal to 0.

This is due to the fact that CLAP must consider all possible execution interleavings between instructions in lines 1,3, and 4, which means that R_x1 can be matched either with the initial value of x, write W_x3, or W_x4. Conversely, since CLAP*, apart from tracing the local execution path, also logs the order in which locks are acquired, it can assume that the instruction $x{+}{+}$ was executed before the instruction $x = 3$. As a result, CLAP* is able to trivially correspond R_x1 to the initial value.

In summary, by guiding the symbolic execution according the synchronization order, CLAP* can simplify both the read-write constraints and the number of possible solutions. In fact, for the example in Figure 2, CLAP*'s read-write constraints only have 3 solutions versus 12 of CLAP. Furthermore, CLAP also needs to infer the order in which the locks were acquired during the original execution (for the sake of simplicity, these constraints were omitted in Figure 3.a), which in some cases may not be a trivial task.

4. PRELIMINARY EXPERIMENTS

To provide initial evidence of our hypothesis, we performed some preliminary experiments. In particular, our assessment focused on the following two criteria:

i) Recording Overhead. We are interested in demonstrating that tracing the locking order incurs a slightly larger recording overhead with respect to CLAP, but is still competitive in comparison to the baseline.

ii) Inference Time. We want to show that the information regarding the order of thread accesses to synchronization points allows to substantially reduce the inference time to find a bug-reproducing execution schedule.

To perform the experiments, we implemented a prototype in Java supporting both CLAP's constraint model and our extended version containing the locking order (CLAP*). We used Soot[2] to instrument the target applications in order to record information at runtime, Java PathFinder[3] to perform symbolic execution, and Z3[4] to solve the constraint model[4].

As test subjects, we used a micro-benchmark and four programs from the IBM ConTest benchmark suite [5]. All experiments were conducted in a machine with an Intel Core 2 Duo at 2.26 Ghz, with 4 GB of RAM and running Mac OS X.

4.1 Recording Overhead

We developed a micro-benchmark to intensively assess how the recording and space overhead of both CLAP and CLAP* varies when increasing: *i)* the number of branches, *ii)* the number synchronization operations, and *iii)* the ratio between synchronized and non-synchronized accesses. In particular, the micro-benchmark consists of a simple a multithreaded program with four threads that perform 10^7 concurrent accesses to four shared variables, where some of the accesses are synchronized some are not. The purpose of the micro-benchmark is allowing us to easily tune either the number of branches or the number of synchronization accesses, while keeping the other parameter constant. Thereby, when we vary the number of branches, the number of synchronization operations is constant, whereas, when we increase the latter, both the amount of branches and accesses to shared variables are kept at 10^7. Finally, for the ratio between synchronized and non-synchronized accesses, we simply vary the percentage of shared operations that are synchronized among the total 10^7 concurrent accesses. The results are plotted in Figure 4 (*Baseline* indicates the non-instrumented version of the program).

As we can see in Figure 4.a), by increasing solely the number of conditional instructions, both CLAP and CLAP* exhibit practically the same recording overhead, which is only 38% for 25 millions branches[5]. The slightly larger logs

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

[3]http://babelfish.arc.nasa.gov/trac/jpf

[4]We did not employ parallelization in the solving process.

[5]Note that more efficient path tracing algorithms (e.g. Ball-

Program	LOC	#Th	#SV	#BR	#Unknown Variables			#Constraints			Inference Time	
					CLAP	CLAP*	Red.	CLAP	CLAP*	Red.	CLAP	CLAP*
TwoStage	136	16	3	164	873	453	↓**48.1%**	2592851	1119124	↓**71.4%**	>**2h**	61s
Piper	165	21	4	160	2132	539	↓**74.7%**	737478	225244	↓**69.5%**	>**2h**	30s
TicketOrder	161	4	6	232	620	548	↓**11.6%**	199799	172033	↓**13.9%**	95s	40s
Manager	180	4	4	613	1286	1115	↓**13.3%**	60824	59370	↓**2.4%**	>**2h**	743s

Table 1: Results for the IBM ConTest benchmarks. Shaded cells indicate that no solution was found within 2 hours.

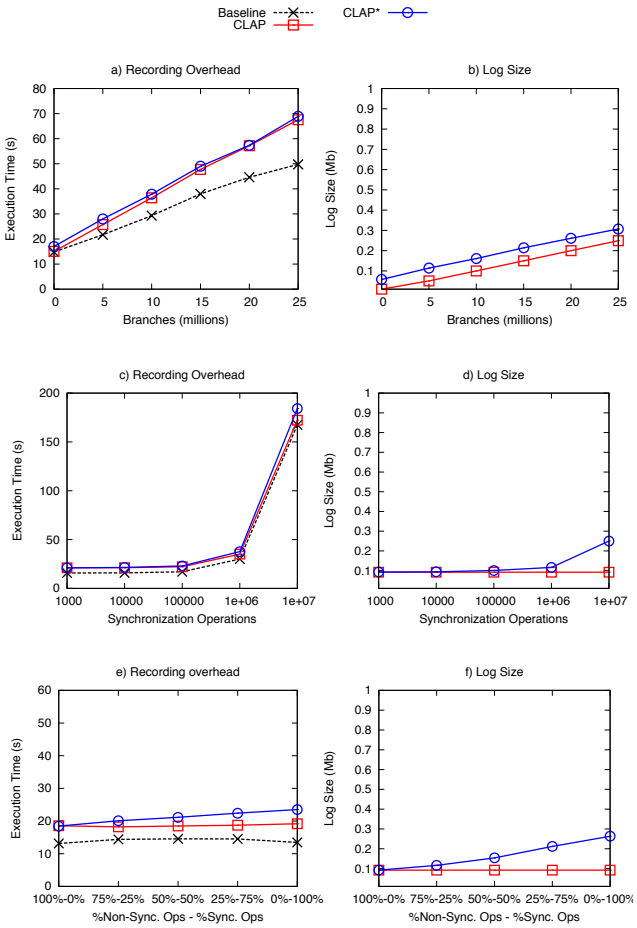

Figure 4: Recording overhead and log sizes for micro-benchmark.

produced by CLAP* in this test are due to the fixed small amount of synchronization operations executed.

Regarding the test aimed at assessing the impact of synchronization operations (Figure 4.c), we can verify that, as their number grows, CLAP*'s performance degradation slightly increases with respect to that of CLAP, although their difference is almost negligible (up to 6%). The same applies for the log sizes, with exception for the case with 10 million synchronization operations, where CLAP*'s log has 250KB and and CLAP's only 92KB. Despite that, these results confirm that, even for large amounts of synchronization operations, our hypothesis is still competitive in comparison to CLAP.

Larus [2]) can be later added to our prototype.

Finally, Figures 4.e) and f) show, respectively, the runtime overhead and the space cost for the micro-benchmark test when increasing the percentage of synchronized shared accesses in the program. As expected, as the percentage grows, CLAP* and CLAP start diverging, with the former achieving a performance slowdown up to 22.5% higher. In terms of space cost, the size of the trace files produced by CLAP* ranges from 92KB to 263KB, whereas CLAP constantly generates logs of 92KB, given that it traces only the thread local path. Once again, these results highlight the potential of our approach, although they also show that the amount of synchronized operations can have a negative impact in both time and space overhead, especially for programs where this kind of operations accounts for the majority of shared accesses.

4.2 Inference Time

To compare CLAP* against CLAP in terms of inference time, we used four programs from the IBM ConTest benchmark suite [5]. This benchmark suite is composed by multiple applications that contain concurrency bugs. Columns 2-5 of Table 1 characterize the used programs in terms of lines of code (LOC), number of threads ($\#Th$), number of shared variables ($\#SV$), and number of branches ($\#BR$).

In turn, columns 6-13 present the results regarding the comparison between CLAP and CLAP* with respect to the number of unknown variables generated, number of constraints, and inference time (which encompasses both the time to build the constraint formula and the time to solve it). For the former two criteria, Table 1 also indicates the respective achieved reduction (*Red.*).

Analyzing the results for the amount of both unknown variables and constraints produced, we see that tracing the locking order provides a maximum reduction of 74.7% and 71.4%, respectively. As expected, this fact results in a substantial decrease of the inference time (both in terms of constraint generation and formula solving). In particular, CLAP* was able to find the buggy execution in less than 61s for TicketOrder, TwoStage, and Piper, whereas, for the latter two programs, the solver could not find a solution for CLAP's set of constraints within 2 hours. Note that, even if CLAP had found solution in this amount of time, CLAP* would have been 118x and 240x faster solving the formula, respectively for TwoStage and Piper. It should also be noted that these reductions in the inference time come at a cost of an increment of 27% and 13% in the runtime overhead, respectively. This further supports our claim regarding the benefits of recording the locking order at runtime.

On the other hand, for program Manager, CLAP* decreased the number of constraints and unknown variables by only 2.4%. Despite that, we can see that the reduction of the solving time is still significant.

Finally, it should be noted that, considering the same in-

put, CLAP* was always able to reproduce the bug at the first attempt (i.e. with the first obtained solution).

5. CONCLUSIONS AND FUTURE WORK

In this paper we explore an interesting tradeoff between recording overhead and efficiency of the inference phase, in the context of search-based deterministic replay. In particular, we argue that extending search-based approaches such as CLAP [8] to also trace the synchronization order results in a significant reduction of the inference time, while imposing a tolerable overhead during the production run. Preliminary experiments with a micro-benchmark and third-party benchmarks support our claim, but need to be reinforced with more complex real-world applications.

Although in its early stages, we believe that this work paves the way for the exploration of novel trade-offs between recording overhead and inference time.

6. ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their valuable feedback. This work has been partially supported by FCT (INESC-ID multi-annual funding) through the PEst-OE/EEI/LA0021 /2013 Program Funds.

7. REFERENCES

[1] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *ACM SOSP*, pages 193–206, 2009.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *ACM/IEEE MICRO*, pages 46–57, 1996.

[3] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *ACM SPDT*, pages 48–59, 1998.

[4] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, pages 337–340, 2008.

[5] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IEEE IPDPS*, pages 286–293, 2003.

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

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

[8] J. Huang, C. Zhang, and J. Dolby. Clap: recording local executions to reproduce concurrency failures. In *PLDI*, pages 141–152, 2013.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM ASPLOS*, pages 329–339, 2008.

[10] N. Machado, P. Romano, and L. Rodrigues. Lightweight cooperative logging for fault replication in concurrent programs. In *IEEE DSN*, pages 1–12, 2012.

[11] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM PLDI*, pages 446–455, 2007.

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

[13] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *ACM ASPLOS*, pages 15–26, 2011.

[14] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. Order: object centric deterministic replay for java. In *USENIX ATC*, 2011.

[15] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *ACM EuroSys*, pages 321–334, 2010.

[16] J. Zhou, X. Xiao, and C. Zhang. Stride: search-based deterministic replay in polynomial time via bounded linkage. In *ICSE*, pages 892–902, 2012.