Fault Reproduction for Multithreaded Applications

(extended abstract of the MSc dissertation)

Angel Manuel Bravo Gestoso
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—Writing distributed and parallel applications is rather difficult. Because of this difficulty, many bugs appear during development and frequently, on deployed applications. Classical debugging techniques are not enough anymore because of the non-determinism induced by this kind of applications.**

Record and replay techniques have been created in order to help developers. These techniques are composed by two main phases. The record phase captures all non-deterministic events of the execution. Then, during the replay phase, the original execution can be repeated in order to find the causes of the bugs. Unfortunately, tracing all non-deterministic events introduces a large overhead.

This thesis presents Symber, a record and replay tool for multithreaded Java applications that combines partial logging with an inference mechanism based on symbolic execution. Thus, Symber is able to reduce the overhead introduced by only logging the local execution path and the order in which the locks are acquired.

Our results demonstrate that Symber produces a competitive overhead in comparison to other techniques and still maintains the ability to efficiently replay concurrency bugs.

I. Introduction

Parallel and distributed applications are extremely difficult to design and implement. Furthermore, the code is very difficult to debug because some errors only appear when a specific thread interleaving occurs, and these interleavings may be hard to produce during test runs. In fact, there is evidence that a majority of this bugs are related to concurrency problems[1]. As a result, it is not rare that concurrent applications are deployed with bugs, including some large and widely used applications such as MySQL, Apache, Mozilla and OpenOffice.

Classical debugging techniques, such as cyclic debugging, consist of repeating the faulty execution until the cause of the bug is found. Unfortunately, this mechanism cannot be easily applied for parallel and distributed systems, because of the non-deterministic nature of the executions may prevent the interleaving that causes the error to be reproduced in an useful number of re-executions. The bugs that do not appear deterministically in every execution, even if the same input is provided, are called "heisenbugs". All these issues make the development and debugging of applications more complex.

Record and replay techniques have been designed to mitigate these difficulties. The goal is to log enough information during the "normal" execution of the application such that, if an error occurs, the development team can later reproduce the buggy execution. In detail, logging runs with the application and tries to capture as many non-deterministic events as possible. Since the amount 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 this phase is to be able to perform logging with small spatial and time overhead. On the other hand, the replay phase is aimed at re-executing the original execution by using the logs that have been created during the record phase.

The first record/replay techniques have been designed for reproducing the bug on the first attempt. Although this is a desirable goal, this resulting approach is quite expensive in terms of spatial and time overhead of the recoding phase, because all non-deterministic events have to be traced in runtime. In consequence, alternative approaches have been attempted recently. In particular, systems such as [2] and [3], are based on the observation that the overhead imposed on the user-side execution can be more disruptive than a longer developer-side debugging execution. Thus, these techniques reduce the overhead of the recording phase by not tracing all non-deterministic events, at the cost of possibly longer replay phases. Since in order to replay the buggy execution one needs to deterministically reproduce all non-deterministic events, these new approaches introduce a new phase between the recording and the replaying phase. This new phase is in charge of inferring the information that has been omitted from the logs.

Our work extends these results, by introducing techniques that allow to further reduce the logging overhead while, at the same time, being able to provide sufficient information to the developers in order to help them to find the causes that produce the bug. We take CLAP [3] as our starting point, since we believe that among all approaches does the best balance between recording overhead, inference time and the information that is provided to the developers. Therefore, our system, named Symber, is an evolution of the CLAP system. Symber is aimed at substantially reducing and simplifying the inference phase by slightly increasing the recording overhead.

The rest of this document is organized as follows. Section II provides an introduction to the different technical areas related to this work. Section III introduces Symber.

Section IV presents the phases involved in a Symber execution and Section V presents in detail Symber's constraint model. Section VII shows the results of the experimental evaluation study. Finally, Section VIII concludes this document by summarizing its main points and future work.

II. Related Work

Considerable work has been done in the development of deterministic replay debugging techniques. The main problem that these techniques face when applied to parallel and distributed applications is that different executions can produce different outputs. Non-determinism is one of the main challenges in the design, optimization, and debugging of parallel applications. In this section we review some of the solutions related to our work.

Order-based approaches are based on tracing the order in which non-deterministic events occur. In the context of parallel programs, the solutions are mainly concern about the order in which threads access to shared memory positions. Thus, solutions, such as InstantReplay[4], LEAP[5], and Ditto[6], trace during the record phase the order in which threads access to the shared variables. These solutions are able to faithfully replay the buggy execution by following the trace, generated in runtime, during the replay phase. However, tracing all non-deterministic events introduces an unbearable runtime overhead in many cases.

Search-oriented approaches are envisioned to reduce the runtime overhead introduced by order-based approaches. They are based on the observation that the efficiency of the user-side execution is more critical than the efficiency of the developer-side execution. Thus, search-based solutions provide an efficient user-side execution by partially logging the non-deterministic events. Due to the missing information, a new phase that searches for the faulty execution is needed between the record and the replay phases. The efficiency of the "search-phase" directly depends on the amount of information that was not traced during the record phase. Examples of this approach are ODR[2], CLAP[3], and DCR[7].

Symber can be seen as an evolution of CLAP. It is aimed at dramatically reducing the search space of the "search-phase" by slightly increasing the runtime overhead.

III. Symber overview

The main goal of Symber is to deterministically replay a buggy execution in multithreaded applications. Since directly applying record and replay techniques introduces an unbearable overhead, Symber combines the techniques used by tools such as [6] and [5] with an inference phase, that helps to reduce both the spatial and time overhead incurred by the tool. It is worth to mention that, to the best of our knowledge, Symber is the first record/replay tool for Java that uses symbolic execution as inference mechanism.

The idea of using inference has been previously applied with success in systems like ODR [2], DCR [7], and CLAP [3]. However, we consider that previous works have not thoroughly addressed all the factors that must be considered when inference is used. Namely, we are concerned in achieving the right tradeoff among the following factors:

- The time overhead of the record phase.
- The time that the inference phase takes.
- The amount of debugging information that the tool provides to the developer. We consider that a useful tool should not only be able to replay the bug, but it should also provide some useful information to the developer in order to find the origin of the bug.

Our hypothesis is that *by adding a small overhead in the record phase (when compared to what CLAP and ODR do), it should be possible to dramatically reduce the inference phase.* The design and development of Symber was performed with the goal of validating this hypothesis.

A regular execution of Symber starts with a static analysis. The goal of this analysis is to prepare the target application for Symber. This preparation is basically an instrumentation of the application code. As an output of this phase, a record and a replay version of the application are generated. Subsequently, the record phase can start. Traces are filled during this phase with not only the execution path but also with the locking order. Then, using the traces, the inference phase starts. This is the most complex and slow phase because it has to find an execution that triggers the same bug than the original execution among all the possible executions. Once the buggy execution has been found, a trace of the accesses to shared variables is created and fed to the replay phase. Then, the buggy execution can be replayed as many times as needed using the trace generated during the inference phase. Figure 1 illustrates the different phases involved in a Symber execution.

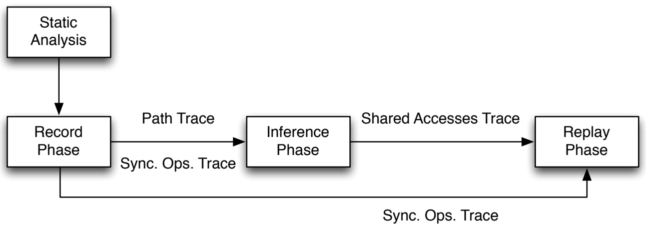

Figure 1. Symber phases.

The inference phase is based on symbolic execution [8] [9] [10]. It uses both the path and the locking order traces to guide the execution. It creates a new fresh symbolic symbol for each shared variable read. Therefore, Symber does not assume anything with regard to data races. During the symbolic execution, Symber creates constraints that represent the execution and feeds them to a constraint solver. The constraint solver sequentially outputs solutions for those constraints that represent thread schedule candidates. The candidates are sequentially checked until the bug is reproduced. Finally, once the buggy execution has been found, a trace with the shared accesses order and the locking order is generated and sent to the replayer.

CLAP	Symber
It only records the execution path	It records the execution path and the locking order
Its symbolic execution is only guided by the path trace	Its symbolic execution is guided by the path trace and the locking order
Its constraint model contains synchronization order constraints that increments the complexity of the constraints	It is capable of discard the synchronization order constraints; therefore, the final formula is easier to solve
Most of the solutions proposed by the constraint might represent an infeasible execution	Reduced number of infeasible thread schedule candidates. It simplifies the solution generation and the solution checking phases.

Table I
MAIN DIFFERENCES BETWEEN SYMBER AND CLAP

Symber uses some ideas that were introduced by CLAP; nevertheless, it differs in many aspects. Similarly to CLAP, it locally logs the execution path on runtime. This path profiling does not need any synchronization among threads; therefore, it is substantially cheaper than other techniques based on tracing the order or the value of shared memory accesses. However, we believe that logging the local execution path does not provide sufficient useful information. In consequence, the inference phase could need to generate and check an unbearable number of execution candidates. For this reason, Symber not only logs the execution path but it also traces the locking order (by locking order we refer to a global order of synchronization operations such as *signal* and *wait* operations or *lock* and *unlock* operations). In consequence to this decision, the inference phase of Symber differs in many parts to the inference phase of CLAP as we will see in the following sections.

Table I lists the main differences between Symber and CLAP.

IV. PHASES IN DETAIL

This section is aimed at explaining in detail the phases that are involved in a Symber execution. It also motivates the need for those phases and how they have been implemented. The multiple phases are depicted in Figure 1.

A. Static analysis

This is the first phase of Symber's execution. It statically analysis the target application. It identifies all shared variable accesses and logs them into a file. This file will be used during the inference phase. Furthermore, during this phase, the record and the replay version are created.

On one hand, for the record phase, the static analysis needs to prepare the application for tracing the execution path and the locking order. In order to trace the execution path, we opted for a simple approach. For each *if* statement, two method invocations are inserted. One right before the statement (*beforeIfStmt* hereafter) and one right after (*afterIfStmt* hereafter). Thus, *beforeIfStmt* is called in case the if statement is about to be executed and *afterIfStmt* is executed only if the condition is satisfied.

Switch statements are instrumented in a different way. The tool injects one method invocation after any of the targets of the switch statement. Thus, we are able to trace which *case* clause has been executed on runtime. In addition, a method is inserted right before the *switch* as it is done with *if* statements.

We are aware that some more efficient approaches could have been used for tracing the execution path [11]. Nevertheless, that is not the goal of Symber. Therefore, due to the temporal constraints we had to develop the prototype, we opted for this more straightforward approach.

On the other hand, in order to trace the locking order, we not only need to trace lock acquisitions but we also have to log any signal and wait operation. The instrumentation is made by inserting *beforeMonitorEnter* method before any signal operations and *afterMonitorEnter* method right after any wait operation or lock acquisition.

Regarding the replay version instrumentation, read/write operations are wrapped by *beforeRead*, *afterRead*, *beforeWrite* and *afterWrite* methods. Furthermore, since it also has to follow the locking order trace, synchronization operations are also wrapped by *beforeMonitorEnter* and *afterMonitorEnter* methods.

B. Record phase

This phase uses the version generated by the static analysis. It basically runs the application and generates both the path trace and the locking order trace.

Figure 2 shows an example of how the path profiler algorithm works. In the figures, methods in **bold** are the instrumented methods. The method *beforeIfStmt*, which goes right before an if statement, provisionally stores (represented by [] in the example) a "false" on the thread's path trace. If the next method for that thread, among those two methods, is an *afterIfStmt* method, that value is changed to true and stored in the permanent path execution log. In case next invocation is another *beforeIfStmt*, the provisional "false" is stored into the permanent path execution log. A similar idea is used for loops and switch statements.

```
                   c1 == True   c1 == True   c1 == False   c1 == False
                   c2 == False  c2 == True   c2 == True    c2 == False

beforeIfStmt()     [F]          [F]          [F]           [F]
if (c1){
  afterIfStmt()    T            T
  stmt 1
}else{
  stmt2
}
stmt3
beforeIfStmt()     T , [F]      T , [F]      F , [F]       F , [F]
if (c2){
  afterIfStmt()                 T , T        F , T
  stmt4
}
...
```

Figure 2. Path profiler.

On the other hand, for tracing the locking order, we have been inspired by the algorithms implemented by Ditto[6]. We consider synchronization operations any monitor acquisition (synchronized method, synchronized block or lock method) and any signal/wait operation (*wait*, *notify*, *notifyAll*, *await*, *signal*, and *signalAll*). These algorithms are aimed at tracing the global order of synchronization operations per locking object. For this purpose, Symber maintains a clock for each locking object and for each thread.

Apart from filling the logs, this phase also generates the replay driver. It basically links the generated trace and the target application with the replayer. The goal of this driver is to automatize the process. Thus, once the inference phase finishes (and, in consequence, the trace that represents the buggy execution has been generated), the user simply needs to run the replay driver to faithfully replay the buggy execution.

C. Inference phase

This is the most complex phase. We can divide it into smaller phases in order to better explain in detail how it works. Figure 3 shows how the phase is divided and how those sub-phases interact.

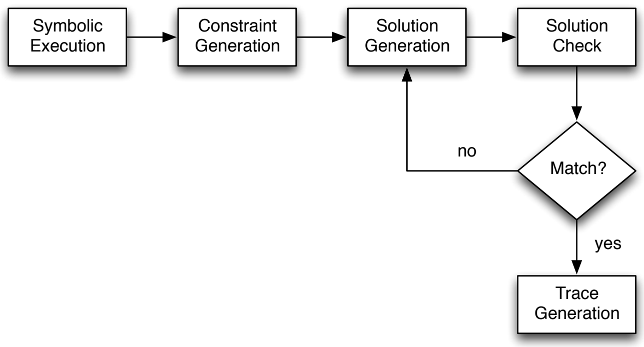

Figure 3. Inference phase in detail.

Symbolic execution: First, the path, locking order, and shared variable operation traces are read. Then, the symbolic execution starts. It is a concolic symbolic execution; therefore, some variables are concrete and some are symbolic. In our tool, any shared read operation creates a new fresh symbolic variable. The identification of shared variable operations is made using the shared variable operation trace generated by the static analysis phase. This trace simply lists all shared memory operations.

Path and locking order traces are used for guiding the symbolic execution. Thus, for each branch in which at least one of the operands is symbolic, the decision is taken based on the path trace. On the other hand, before executing a synchronization operation, the locking order trace has to be checked. In the case the thread is invoking the lock in the logged order, the thread continues the execution. Otherwise, the thread gets blocked and a context switch is executed. The thread keeps blocked until it is his turn according to the locking order trace.

The goal of the symbolic execution is to gather as much information about the execution as possible. For this purpose, it records the following useful information:

- Path condition (PC): It gathers constraints related to the symbolic branch decisions. Thus, when a decision is taken, it is translated into a constraint and added to the PC. For instance, considering the following if statement (where $sym1$ is a symbolic variable): $if(sym1 > 5)$. If, according to the path trace, the condition was satisfied in the original execution, the constraint that is added to the path condition would be written as $sym1 > 5$, otherwise $sym1 \leq 5$.
- Shared memory accesses sorted by thread. Thus, it represents a local order of shared memory accesses.
- Write set and read set per shared variable.
- Initial values of the shared variables.
- Synchronized shared memory accesses sorted by locking object. Since the symbolic execution is guided by the locking order, we can be sure that this order is part of the buggy execution.

Once the symbolic execution has finished and all the information has been gathered, the symbolic execution subphase finishes and the constraint generation phases starts.

Constraint generation: This sub-phase takes all the information produced by the symbolic execution and generates a global formula. This formula represents the set of possible executions to which the buggy execution belongs. Section V presents the constraint model in detail.

Solution generation: Once all the constraints have been created, the formula is solved by a constraint solver. We are not interested on the values of the symbolic variables; we are just interested on the global order of read/write operations because it represents how the threads interleave during the execution.

Having the symbolic execution guided by the locking order let us to create the synchronization order constraints. This substantially reduces the search space and the number of infeasible solutions that the solver might suggest.

Solution check and trace generation: The checker takes the solution proposed by the solver and executes the target application in order to check whether in produces the bug or not. In case the bug is reproduced, the final trace is generated and the inference phase finishes. Otherwise, it requests another solution to the constraint solver and the solution check phase starts again. Since the buggy execution belongs to the group of the possible solutions, this phase will always finish.

An optimization has been applied in order to reduce the size of the generated traces. The idea is to store deltas values instead of absolute values. Thus, large numbers are never traced and a cheaper representation can be used (such as $short$ instead of int).

D. Replay Phase

The replay phase starts by reading the trace generated by the inference phase and the record phase. This trace contains the order in which read/write operations and the synchronization operations have to be executed for reproducing the buggy execution. Then, the execution begins. Since it runs the instrumented version of the target application, each time a read/write operation or a synchronization operation is about to be executed, our injected methods (*beforeRead*, *beforeWrite*, or *beforeMonitorEnter*) are invoked. The purpose is to block the thread in case it does not have to execute the operation. Once, the thread has been allowed to execute the operation (because the trace matches the current execution), the trace advances and the clocks are updated by our second part of injected methods (*afterRead*, *afterWrite*, and *afterMonitorEnter*).

V. Constraint Model

Symber's final formula is a composition of constraints. It can be written as:

$\alpha = \alpha_{pc} \wedge \alpha_{rw} \wedge \alpha_{mo} \wedge \alpha_{so} \wedge \alpha_b$

where α_{pc} denotes the path condition constraints, α_{rw} represents the read-write relationship constrains, α_{mo} represents the memory order constraints, α_{so} denotes the synchronization order constraints and α_b represents the bug.

Path condition (α_{pc})*:* It represents the execution path. Thus, each constraint bounds the value of the symbolic variables involved in the if statement that has generated the constraint. These group of constraints restrict the number of solutions for the read/write constraints.

Bug constraint (α_b)*:* It is the translation of the bug into a constraint. It helps to bound the possible solutions of the formula to those in which the bug is triggered.

Read-Write constraint (α_{rw})*:* It expresses the relationship between read and write operations for a specific shared variable. Thus, it tries to infer which write operation precedes each read. The constraints for each read operation (r) are written as follows:

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

where V_r is the read value, $init$ is the initial value of the shared variable, W is the write set for that shared variable, O_r denotes the order of r and O_{w_i} the order of the write operation w_i.

Memory order constraint (α_{mo})*:* It represent the local order for all read/write operations executed by a thread.

Synchronization order constraint (α_{so})*:* It represents the global order of shared variable accesses under a specific locking object. For those read/write operations that are performed in a synchronized block, we can assume a global order among all threads.

Although Symber constraint model is inspired by CLAP, it is substantially simpler. Symber does not need to infer the locking order because it has been traced. Therefore, it can discard CLAP's "Synchronization Order Constraints" whose, according to CLAP, size can be written as:

$N_{lo}(2|S|^3+2|S|)+N_{jo}+N_{fo}+N_{sv}(2|SG||WT|+|SG|)$

where N_{lo} denotes the number of locking objects, S denotes the set of lock/unlock pairs of a specific locking object, N_{jo} denotes de number of join operations, N_{fo} represents the number of fork operations (start method in Java), N_{sv} denotes the number of signal variables, SG is the set of signal operation that operates in a specific signal variable and WT represents the set of wait operations that can be mapped by a specific signal operation.

We note that removing these constraints from our formula simplifies the constraint solving task, considering that we discard a cubic formula from our model. Furthermore, CLAP cannot straighly solve the formula since the majority of the solutions might represent a infeasible execution (in constrast to Symber solutions) and going trough all the solutions might require an unreasonable amount of time. The lack of the locking order trace is the cause of this drawback.

A. Example

Figure 4 shows a simple example that helps to visualize the benefits of Symber in comparison to CLAP. It represents a multithreaded application that contains both synchronized and non-synchronized accesses to shared variables. R_xn (W_xn) denotes a read (write) operation over the shared variable x (n simply denotes the line in the source code).

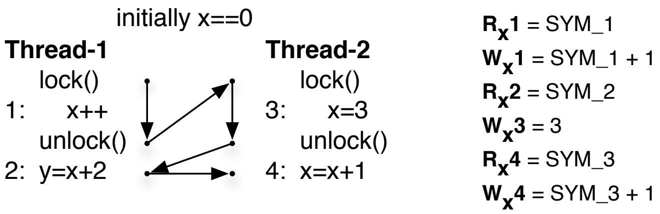

Figure 4. Simple multithreaded program.

During the symbolic execution phase of CLAP and Symber, shared read/write operations are identified and logged. A read operation creates a new fresh symbolic symbol (E.g. the operation R_x1 creates the symbolic symbol SYM_1). In addition, the tools store the value that it is written by every shared write operation.

Once the symbolic execution has finished, both CLAP and Symber generate a set of constraints that helps to infer the original execution. Figure 5 presents a simplified version of the constraints that CLAP and Symber would generate for the example. OR_xn denotes the order of the corresponding operation (R_xn) in the to-be-computed thread schedule. Memory-order constraints represent the order in which the operations have been locally executed by each thread.

Figure 5 shows that the read-write constraints generated by Symber 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 Symber, SYM_1 can only be equal to 0.

This simplification can be achieved because Symber, apart from logging the local execution path, logs the order in which locks are acquired. Thus, Symber has the symbolic execution guided by the path trace and the locking order. In consequence, information regarding the global execution path is generated by the symbolic executor (in case the

CLAP

memory-order constraints

$(OR_x1 < OW_x1 < OR_x2)$ &&
$(OW_x3 < OR_x4 < OW_x4)$

read/write constraints

$((\mathbf{SYM\_1} = 0 \,\&\, (OR_x1<OW_x1) \,\&\, (OR_x1<OW_x3) \,\&\, (OR_x1<OW_x4)) \mid$
$(\mathbf{SYM\_1} = 3 \,\&\, (OW_x3<OR_x1) \,\&\, (OW_x1<OW_x3 \mid OW_x1<OR_x1) \,\&\, (OW_x4<OW_x3 \mid OW_x4<OR_x1)) \mid$
$(\mathbf{SYM\_1} = SYM\_3+1 \,\&\, (OW_x4<OR_x1) \,\&\, (OW_x1<OW_x4 \mid OW_x1<OR_x1) \,\&\, (OW_x3<OW_x4 \mid OW_x3<OR_x1)))$
&&
$((\mathbf{SYM\_2} = SYM\_1+1 \,\&\, (OW_x1<OR_x2) \,\&\, (OW_x3<OW_x1 \mid OW_x3<OR_x2) \,\&\, (OW_x4<OW_x1 \mid OW_x4<OR_x2)) \mid$
$(\mathbf{SYM\_2} = 3 \,\&\, (OW_x3<OR_x2) \,\&\, (OW_x1<OW_x3 \mid OW_x1<OR_x2) \,\&\, (OW_x4<OW_x3 \mid OW_x4<OR_x2)) \mid$
$(\mathbf{SYM\_2} = SYM\_3+1 \,\&\, (OW_x4<OR_x2) \,\&\, (OW_x1<OW_x4 \mid OW_x1<OR_x2) \,\&\, (OW_x3<OW_x4 \mid OW_x3<OR_x2)))$
&&
$((\mathbf{SYM\_3} = SYM\_1+1 \,\&\, (OW_x1<OR_x4) \,\&\, (OW_x3<OW_x1 \mid OW_x3<OR_x4) \,\&\, (OW_x4<OW_x1 \mid OW_x4<OR_x4)) \mid$
$(\mathbf{SYM\_3} = 3 \,\&\, (OW_x3<OR_x4) \,\&\, (OW_x1<OW_x3 \mid OW_x1<OR_x4) \,\&\, (OW_x4<OW_x3 \mid OW_x4<OR_x4)))$

(a)

SYMBER

synchronization constraints

$(OR_x1 < OW_x1 < OW_x3)$

memory-order constraints

$(OR_x1 < OW_x1 < OR_x2)$ &&
$(OW_x3 < OR_x4 < OW_x4)$

read/write constraints

$(\mathbf{SYM\_1} = 0 \,\&\, (OR_x1<OW_x1) \,\&\, (OR_x1<OW_x3) \,\&\, (OR_x1<OW_x4))$
&&
$((\mathbf{SYM\_2} = SYM\_1+1 \,\&\, (OW_x1<OR_x2) \,\&\, (OW_x3<OW_x1 \mid OW_x3<OR_x2) \,\&\, (OW_x4<OW_x1 \mid OW_x4<OR_x2)) \mid$
$(\mathbf{SYM\_2} = 3 \,\&\, (OW_x3<OR_x2) \,\&\, (OW_x1<OW_x3 \mid OW_x1<OR_x2) \,\&\, (OW_x4<OW_x3 \mid OW_x4<OR_x2)) \mid$
$(\mathbf{SYM\_2} = SYM\_3+1 \,\&\, (OW_x4<OR_x2) \,\&\, (OW_x1<OW_x4 \mid OW_x1<OR_x2) \,\&\, (OW_x3<OW_x4 \mid OW_x3<OR_x2)))$
&&
$(\mathbf{SYM\_3} = 3 \,\&\, (OW_x3<OR_x4) \,\&\, (OW_x1<OW_x3 \mid OW_x1<OR_x4) \,\&\, (OW_x4<OW_x3 \mid OW_x4<OR_x4))$

(b)

Figure 5. Constraints generated by CLAP for the example shown in the Figure 4.

application has synchronization operations). E.g. assuming that the original run follows the execution depicted by the arrows in Figure 4, Symber has the ability to generate the synchronization constraints that represent the locking order. Thus, Symber can assume that the instruction $x++$ was executed before the instruction $x = 3$. Knowing this extra information, let Symber simplify the read-write constraints, and, in consequence, the number of possible solutions. For the example depicted by the Figure 4, Symber read-write constraints only have three solutions, while, for CLAP, the number of solution is multiplied by four (12).

CLAP also needs to infer the order in which the locks were acquired during the original execution which might be a tedious task.

VI. Implementation

Symber is completely implemented in Java. Soot [12] tool has been used for the static analysis and the instrumentation of the target application. On the other hand, we have used the open source project Java PathFinder [13] for implementing Symber's symbolic executor. This executor not only uses jpf-core, which is the basic package, but it also uses multiple extensions such as jpf-symbc [14], which is envisioned for symbolic execution, and jpf-concurrent.

Regarding the constraint solvers, we have implemented Symber thinking of the constraint solver component as a pluggable element. Therefore, it is quite straight forward to substitute it by a different one. We mainly experiment with Choco2 [15].

A. Challenges

We had to overcome many challenges during the implementation phase. In the following paragraphs, we detail how we solved some of the most significant challenges we have faced.

Thread consistent identification: Different execution of the same multithreaded application can identify threads in a different way. Since Symber executes the same application several times along all the phases, we need to remove that non-determinism from our tool. Therefore, all our components implement the same mechanism for identifying the threads.

The main thread is assigned to the identifier "1". When a new thread is created, its identifier is created by the concatenation of his parent-id, ":", and a children counter that gets incremented every time a new thread is created. The children counter is local for each thread.

Thread termination: Symber only logs the local execution path and the locking order. Therefore, Symber does not know which is the last instruction executed by each thread. In order to circumvent this problem in the inference phase, if during the symbolic execution, the thread is about to execute an *if* statement or a synchronization operation, and the trace for any of those statements has been already consumed, the thread is immediately killed. A similar approach is used by the replayer.

We are aware that this is not the perfect solution since some extra instructions might be executed; nevertheless, we consider that those extra instruction do not modify the accuracy of our tool.

Shared variable identification: Other record and replay tools, such as LEAP [5], opt for a field identification of shared variables. Thus, different instances of the same class are treated as one. This increments the recording overhead (for those tools that also record read/write operations order) and it is not compatible with our inference tool since some of the constraints in the path condition might not be compatible with the read/write constraints. In consequence, we decided to identify shared variables by the instance. This makes our constraint solver to generate less constraints and to correctly work.

VII. Evaluation

In this section, we present an experimental evaluation of the Symber system.

A. Evaluation Methodology

Our evaluation addresses the following three complementary aspects of the tool:

- The overhead imposed by Symber, namely the recording overhead and the space overhead (trace files size).
- The Symber capacity for reproducing bugs. This criterion mainly tells us whether Symber is able to reproduce the bug in the tested applications.

	#Threads	**#Branches**	**#Accesses**	**#Shared Vars.**	**%Syn. Accesses**
MB 1	*variable*	10^7	10^7	4	50%
MB 2	4	10^7	*variable*	4	50%
MB 3	4	10^7	10^7	4	0%
MB 4	4	10^7	10^7	*variable*	0%
MB 5	4	*variable*	10^7	4	0%
MB 6	4	10^7	10^7	4	*variable*

Table II
MICROBENCHMARKS

- The number of solutions that are checked by the inference tool until the buggy execution is found. This criterion captures how efficient Symber's inference phase is.

In order to have a comparative assessment of how well Symber performs with regard to other tools, we have also implemented a non-optimized version of Ditto and a simplified version of CLAP.

We have evaluated Symber using a combination of microbenchmarks and third-party benchmarks. All experiments were run in a machine with an Intel Core 2 Duo at 2.26 Ghz, with 4 GB of RAM, and running Mac OS X.

B. Time and Spatial Overhead

The experiments presented in the section have two main goals:

- Demonstrate that Symber's recording overhead is slightly larger than CLAP's recording overhead but still competitive in comparison to the baseline.
- Show that Symber still introduces substantially smaller overhead in comparison to full-recording tools.

1) Microbenchmarks: We have used a number of microbenchmarks for intensively analyze and compare Symber. In our experiments, we have varied one parameter and kept constant the rest of them. Thus, we can appreciate how the isolated variation of one of the parameters affects both the time and the spatial overhead of the recording phase. Using this methodology, we have defined six different microbenchmarks, depicted in Table II.

Using these benchmarks we have measured the performance of: 1) the baseline time (without tracing any information); 2) a Ditto-like implementation that traces the order of synchronization operations and shared read/write operations; 3) CLAP that only traces the execution path, and; 4) Symber that traces the execution path and the order of synchronization operations (the locking order).

Our experiments have demonstrated that the optimized version of Symber creates insignificant trace sizes. We now describe in detail, the results of each microbenchmark with regard to the runtime overhead.

Number of threads*:* In this experiment we assess how Symber scales as the number of threads increases. The results confirm that both CLAP and Symber execution times are close to the baseline. We can conclude that for both tools the number of threads does not affect the execution time. On the other hand, for the Ditto-like application, apart from incurring a considerable larger overhead in comparison to the other approaches, the more threads the application runs, the greater the time overhead is (from 34 seconds to 40 seconds). This is due to the synchronization of shared read/write operations produced by the recording algorithm.

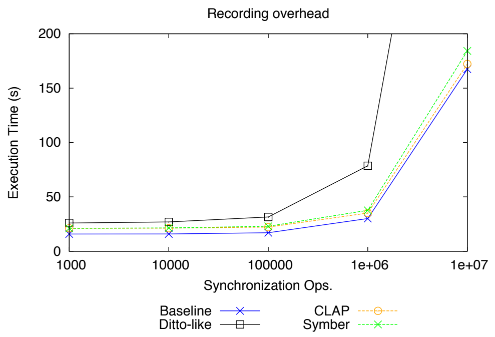

Figure 6. Results for Microbenchmark 3.

Number of shared accesses*:* This experiment, using Microbenchmark 2, checks how Symber behaves, in comparison to the other approaches, when the number of shared read/write operations increases. Since we keep the percentage of synchronized accesses, as the number accesses is increased, the number of synchronization operations is also increased.

As expected, the increment of accesses does no affect CLAP that maintains the same distance to the baseline along the experiment (7s). Still, since the number of synchronization operations also increases, Symber slightly augments the distant to both CLAP and the baseline. Finally, the Ditto-like tool is heavily affected.

Number of synchronization operations*:* This experiment assesses how Symber scales as the number of synchronization operations increases. Figure 6 shows the time overhead imposed with Microbenchmark 3. Both CLAP and Symber start almost from the same point. This is due to the small amount of synchronization operations (only 1000). As the number of synchronization operations increases, the gap between Symber and CLAP executions increases. However, the distance between them holds quite small.

This results confirm that even for large amounts of synchronization operations, Symber is still competitive in comparison to CLAP.

Number of shared variables*:* This experiment checks how the number of shared variables affects the performance of Symber. As it is expected, neither Symber nor CLAP are affected by the increment of shared variables.

Number of branches*:* This experiment measures how the number of branches affects Symber. This is also useful to visualize when a full-recording tool and a tool based on tracing the execution path (Symber or CLAP) might converge.

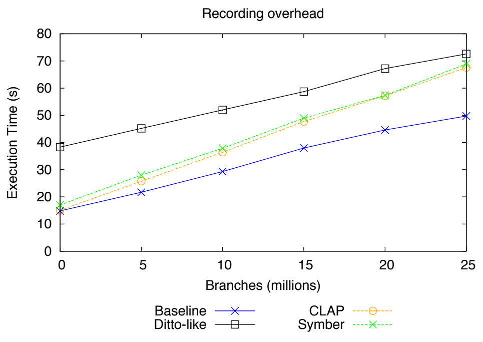

Figure 7. Results for Microbenchmark 5.

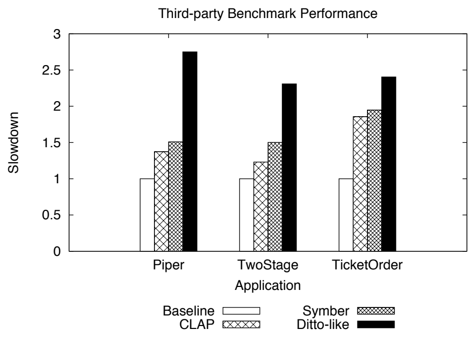

Figure 8. Third-party benchmarks. Performance slowdown.

Program	LOC	#Threads	#sv	#Br	Bug
TwoStage	136	16	3	164	Two-stage
Piper	165	21	4	160	Missing condition for Wait

Table III
DESCRIPTION OF THE IBM CONTEST BENCHMARK APPLICATIONS USED IN THE EXPERIMENTS

Figure 7 shows the time overhead using Microbenchmark 5. As expected, when number of branches is zero, Symber and CLAP are close to the baseline (actually CLAP does not incur any overhead). As the number of branches increases, both CLAP and Symber become separated from the baseline. Since the number of synchronization operation is constant, the distance between CLAP and Symber is also steady. Furthermore, we can notice that when the number of branches is 25 millions, the overhead of tracing the branches almost compensate the overhead introduced by the Ditto-like tool.

Synchronized accesses vs non-synchronized accesses*:* This experiment tests how the approaches behaves when the ratio between synchronized and non-synchronized shared read/write operations varies. Regarding runtime overhead, we expect Symber to converge to CLAP when there are no synchronized accesses. As we were expecting, CLAP and Symber converge when there are no synchronized accesses. The maximum difference between them is produced when all the accesses are synchronized since Symber has more synchronization operations to trace.

2) Third-party Benchmarks: In order to test Symber with more realistic applications, third-party benchmarks have been used. The IBM ConTest benchmark suite [16] is composed by multiple applications that contain concurrency bugs. Table III shows a brief description of the used applications in terms of lines of code (LOC), number of threads (#Threads), number of shared variables (#sv), number of branches (#Br) and the bug-pattern.

Figure 8 shows the slowdown (times slower) introduced by the approaches. In the figure, the baseline, CLAP, Symber, and a full-recording tool are compared. Thus, as expected, the full-recording tool introduces a large slowdown (almost 3x for Piper application). On the other hand, CLAP and Symber maintain an effective overhead that never goes beyond the 51%.

Table IV reports the results. All data were average over ten runs. The reduction of the overhead produced by Symber in comparison to the full-recording tool is significant. Furthermore, Symber still maintains a competitive overhead in comparison to CLAP.

Figure 9 presents the slowdown introduced by Symber in detail. Thus, for each benchmark, the slowdown has been divided into three boxes. The black box represents the naive application, the white box represents the slowdown introduced by tracing the locking order and the grey box represents the slowdown introduced by the path profiler. As the figure shows, Symber incurs a similar overhead for all applications. This totally depends on the application as the microbenchmarks suggest. Nevertheless, as the Figure 9 shows, the slowdown introduced by both tracing mechanisms tends to be similar. None of the tracing mechanisms need extra synchronization; therefore, their use is always more efficient than tracing the order of shared read/write operations.

C. Capacity of Reproducing Bugs

Apart from measuring both the time and the spatial overhead that Symber introduces during the record phase, we also have to evaluate Symber's capacity for reproducing bugs. Thus, we decided to use the third-party benchmarks used in the previous section to test Symber.

Symber was able to find the buggy execution in all the tested applications (Table III). Furthermore, the experiments have proven that Symber is very efficient finding the buggy execution. E.g. for the TwoStage application, Symber was able to find the faulty execution by only checking the first solution suggested by the solver. On the other hand, these experiments have shown that even for medium-size applications, the number of constraints generated by the

Program	Baseline	Ditto-like	CLAP	Symber
Piper	17.8668ms	49.1391ms (175%)	24.5357ms (37%)	26.9579ms (51%)
TwoStage	16.5868ms	38.293ms (131%)	20.4114ms (23%)	24.9112ms (50%)
TicketOrder	9.176ms	22.06ms (140%)	17.0288ms (85%)	17.8709ms (95%)

Table IV
RUNTIME OVERHEAD COMPARISON BETWEEN DITTO-LIKE TOOL, CLAP AND SYMBER

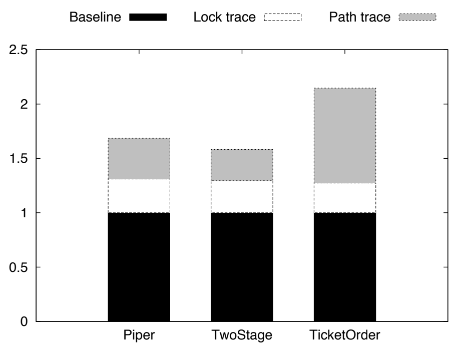

Figure 9. Third-party benchmarks. Symber detailed slowdown.

constraint generator might be pretty large. For instance, the TwoStage program, with only 136 LOC and 274 shared read/write operations, generates 1,113,117 constraints.

D. Efficiency of the Inference Mechanism

Finally, we have also compared Symber with CLAP in terms of number of constraints and number of variables generated. Table V presents the results. As it is expected, Symber dramatically reduces the number of both constraints and variables seen by the constraint solver.

E. Discussion

Evaluating our system in comparison to CLAP, the baseline and a full-recording tool (Ditto-like) serves us to confirm that a full-recording tool introduces an unbearable runtime overhead in many cases. The results also confirm that locally tracing the execution path introduces a tolerable runtime overhead. Furthermore, the size of the traces generated by the path profiler are almost insignificant.

More importantly, the results confirm that tracing the locking order is a cheap operation. Every experiments has denoted that the overhead introduced by tracing the synchronization operations is rather low. E.g. in the worst case situation when the number of synchronization operations is 10^7, the overhead introduced by Symber in comparison to CLAP is lower than 7%. This confirmation is important since Symber is mainly based on this observation. Furthermore, real applications tend to have a greater number of branches than synchronization operations; therefore, we consider that tracing the path is a more expensive mechanism in terms of time overhead than tracing the locking order.

When using the third-party benchmarks, Symber has proved its ability to replay concurrency bugs very efficiently. Nevertheless, we consider that additional evaluation effort needs to be done, in order to fully analyze Symber's capability.

Regarding the comparison between Symber and CLAP, results are promising, since a very significant reduction in terms of number of constraints and number of variables is achieved. This reduction directly diminishes the search space of the constraint solver. This implies that the worst case scenario (when all possible solutions have to be checked in order to find the buggy execution) is also reduced. Still, we would like to experiment with Symber and CLAP in order to confirm these observations.

Finally, it is important to emphasize that Symber is still a research prototype that, currently, has the following limitations:

- Exceptions affect the execution path. Thus, simply tracing the output of if and $switch$ statements is not enough.
- The constraint solving phase, with the current system implementation, may take a very large amount of time for medium-large applications.
- We still believe that tracing every $if/switch$ statement is considerable expensive.

VIII. CONCLUSIONS

This thesis has presented Symber, a record/technique that proposes a cheaper record phase in order to reduce the runtime overhead. Symber reduces the recording overhead by not tracing all non-deterministic events. Thus, a new phase is introduced in order to infer the missing information. A regular Symber execution is composed by three phases: the record phase, the inference phase and the replay phase. The record phase traces the local execution path and the order in which locks are acquired. The inference phase infers the missing information needed for faithfully replaying the buggy execution. This phase is based on a symbolic execution guided by both the local execution path and the locking order where every shared read operation creates a fresh symbolic symbol. Finally, when the inference phase has found the buggy execution, the replay phase uses the trace generated by the inference phase for replaying the faulty execution.

Although there are other approaches that use symbolic execution in its inference phase ([2], [7], and [3]), we believe that Symber provides a new and interesting tradeoff among the following factors: recording overhead, efficiency of the

	#Variables			#Constraints		
Program	**CLAP**	**Symber**	**Reduction**	**CLAP**	**Symber**	**Reduction**
TwoStage	5407122	1113282	↓71.4%	5406748	1113117	↓71.4%
Piper	580609	37793	↓93.5%	579686	37617	↓93.5%

Table V
#CONSTRAINTS AND #VARIABLES COMPARISON BETWEEN CLAP AND SYMBER

inference phase, and the information that the tool gives to the developers.

The evaluation of the tool has shown that tracing the locking order is not an expensive mechanism. Furthermore, the results have shown that Symber is capable to efficiently reproduce concurrency bugs.

Although this document has presented an intensive evaluation of Symber, we plan to improve it further with real applications. Furthermore, as Section VII-E suggests, we need to experimentally compare Symber to CLAP. For this purpose, a full CLAP tool for Java applications has to be implemented.

On the other hand, we have realized that Choco2 (the constraint solver used by Symber) performs poorly for the kind of constraints that Symber generates. We are considering to add MiniZinc and Z3 constraint solvers to Symber. Furthermore, some parts of the inference phase, such as the "Execution Checker", can be parallelized.

Finally, we still believe that tracing the local execution path is an expensive mechanism. We are convinced that probabilistic record/replay can be combined to Symber in order to reduce the recording overhead.

ACKNOWLEDGMENTS

Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Nuno Machado, Nuno Diegues and João Matos.

REFERENCES

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. Seattle, Washington, USA: ACM, 2008, pp. 329–339.

[2] G. Altekar and I. Stoica, "ODR: output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP'09. Big Sky, Montana, USA: ACM, 2009, pp. 193–206.

[3] J. Huang, C. Zhang, and J. Dolby, "CLAP: recording local executions to reproduce concurrency failures," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 141–152.

[4] T. LeBlanc and J. Mellor-Crummey, "Debugging parallel programs with Instant Replay," *Computers, IEEE Transactions on*, vol. C-36, no. 4, pp. 471–482, 1987.

[5] J. Huang, P. Liu, and C. Zhang, "LEAP: lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 207–216.

[6] J. Silva, "Ditto - deterministic execution replay for java virtual machine on multi-processor," Master's thesis, Instituto Superior Técnico.

[7] G. Altekar and I. Stoica, "DCR: Replay-Debugging for the Datacenter," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-33, March 2010.

[8] R. Floyd, "Assigning meanings to programs," in *Program Verification*, ser. Studies in Cognitive Systems, T. Colburn, J. Fetzer, and T. Rankin, Eds. Springer Netherlands, 1993, vol. 14, pp. 65–81.

[9] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[11] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29. Paris, France: IEEE Computer Society, 1996, pp. 46–57.

[12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–.

[13] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, Grenoble, France, 2000, pp. 3–11.

[14] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. Antwerp, Belgium: ACM, 2010, pp. 179–180.

[15] N. Jussien, G. Rochart, X. Lorca *et al.*, "Choco: an open source java constraint programming library," in *CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08)*, Paris, France, 2008, pp. 1–10.

[16] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Nice, France: IEEE Computer Society, 2003.