CLAP: Recording Local Executions to Reproduce Concurrency Failures

Jeff Huang
Hong Kong University of Science and Technology
smhuang@cse.ust.hk

Charles Zhang
Hong Kong University of Science and Technology
charlesz@cse.ust.hk

Julian Dolby
IBM Thomas J. Watson Research Center
dolby@us.ibm.com

Abstract

We present CLAP, a new technique to reproduce concurrency bugs. CLAP has two key steps. First, it logs thread local execution paths at runtime. Second, offline, it computes memory dependencies that accord with the logged execution and are able to reproduce the observed bug. The second step works by combining constraints from the thread paths and constraints based on a memory model, and computing an execution with a constraint solver.

CLAP has four major advantages. First, logging purely local execution of each thread is substantially cheaper than logging memory interactions, which enables CLAP to be efficient compared to previous approaches. Second, our logging does not require any synchronization and hence with no added memory barriers or fences; this minimizes perturbation and missed bugs due to extra synchronizations foreclosing certain racy behaviors. Third, since it uses no synchronization, we extend CLAP to work on a range of relaxed memory models, such as TSO and PSO, in addition to sequential consistency. Fourth, CLAP can compute a much simpler execution than the original one, that reveals the bug with minimal thread context switches. To mitigate the scalability issues, we also present an approach to parallelize constraint solving, which theoretically scales our technique to programs with arbitrary execution length.

Experimental results on a variety of multithreaded benchmarks and real world concurrent applications validate these advantages by showing that our technique is effective in reproducing concurrency bugs even under relaxed memory models; furthermore, it is significantly more efficient than a state-of-the-art technique that records shared memory dependencies, reducing execution time overhead by 45% and log size by 88% on average.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics; Tracing; Symbolic execution; Debugging aids

General Terms Algorithms, Design, Performance, Theory

Keywords Concurrency, Bug Reproduction, Local Execution, Constraint Solving

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

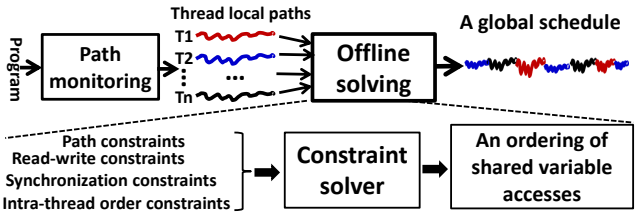

Figure 1. CLAP technical overview

1. Introduction

When diagnosing the root cause of a concurrency bug, to be able to reproduce the bug is crucial but notoriously difficult due to the non-deterministic memory races. Researchers have proposed a wide spectrum of techniques to address the bug reproduction problem. At one end, the deterministic record-replay techniques [12, 14, 21, 22, 26, 32] faithfully capture shared memory dependencies online. At the other end, execution synthesis techniques [36, 40] completely rely on offline analysis to search for shared memory dependencies without any runtime monitoring. Standing in between are several hybrid techniques [1, 23, 24, 29, 41] that explore the right balance between online recording and offline search.

Considering the production-like environments where we should minimize the diagnostic perturbation to the program execution, there are several critical drawbacks of the aforementioned bug reproduction techniques. First, the techniques that introduce locks [14, 21, 22, 41] to track race orders often make the program run much slower (LEAP[14]>6x and Chimera[21]>2.4x for programs with heavy shared memory dependencies). What is worse is that they can also exert the so-called Heisenberg effect by eliminating the concurrency bugs while trying to capture them. The locks they insert act as memory barriers and can prevent the instruction reordering common to most modern commodity multiprocessors. Second, recording values traces [23, 24, 41] to match the shared loads and stores usually incurs a considerable program slowdown and a large disk space to store the logs (Lee *et al.* [23, 24] and Zhou *et al.* [41] reported average trace sizes from 2MB/s to 200MB/s). Third, the complete offline analysis has limited bug reproduction capabilities due to the explosion of both the number of program paths and choices of thread schedules.

In this work, we propose a new technique, CLAP, that reproduces concurrency bugs without recording any shared memory dependency nor any value or order information, and without introducing any extra synchronization. Our key insight is to reduce this problem into two well-known problems: monitoring thread local execution paths and solving symbolic constraints. Since these two problems have been studied for decades, many highly optimized

solutions can be directly leveraged. For example, for path collection, efficient solutions are widely available on both software and hardware levels, such as the classical Ball-Larus path profiling algorithm [4, 20] (around 30% overhead) and the hardware monitoring techniques based on branch predictors [17] and path descriptors [31] (as low as 0.6% overhead); for constraint solving, the SMT solvers such as Yices [8] and Z3 [7] are becoming increasingly powerful with the advances of theorem provers and decision procedures.

As illustrated in Figure 1, CLAP has two key phases:

1. Monitoring an instrumented execution of the program. Unlike most dynamic techniques that collect a global trace, this phase records only the local control-flow choices of each thread. In threads that exhibit bugs, these local traces lead to the occurrence of the bug.

2. Assembling a global execution that exhibits the bug. This phase in turns has several key steps:

- Find all the possible shared data access points (called SAP - a read, write, or synchronization) on the thread local paths that may cause non-determinism, via a static escape analysis.
- Compute the path conditions for each thread with symbolic execution. Given the program input, the path conditions are all symbolic formulae with the unknown values read by the SAPs.
- Encode all the other necessary execution constraints – *i.e.*, the bug manifestation, the synchronization order, the memory order, and the read-write constraints – into a set of formulae in terms of the symbolic value variables and the order variables.
- Use a SMT solver to solve the constraints, which computes a schedule represented by an ordering of all the SAPs, and this schedule is then used by an application-level thread scheduler to deterministically reproduce the bug.

With thread local path monitoring and constraint solving, CLAP achieves several important advances over previous approaches:

1. CLAP obviates logging of shared memory dependencies and program states, and completely avoids adding extra synchronizations. This not only substantially reduces the logging overhead compared to the shared memory recorders [14, 21, 41], but also minimizes the perturbation that extra synchronizations foreclose certain racy behaviors.

2. CLAP not only works for sequential consistent executions, but also for a range of relaxed memory models such as TSO and PSO [2]. We show that the memory order constraints between SAPs can be correctly modeled to respect the memory model relaxation. This is of tremendous importance, because it makes CLAP applicable for real production setting on commodity multiprocessors that allows the reordering of instructions.

3. CLAP can produce simpler bug-reproducing schedules than the original one. We are able to encode preemption-bounding constraints over the order of shared data accesses to always produce a schedule with the minimal number of thread context switches. With this property, it becomes much easier to understand how the bug occurs due to the prolonged sequential reasoning [15, 16]. Moreover, through preemption-bounding, the complexity of constraint solving is dramatically reduced from exponential to polynomial with respect to the execution length.

4. The constraint solving in CLAP is much easier to scale. The solver does not need to directly solve the complex path constraints (such as non-linear arithmetic or string constraints), but only to find a solution for the order variables that satisfies the path constraints. Thus, the solving task can be divided into two parts – generating candidate schedules (that respect the memory order) and validating them (using the other constraints). Since the first part which is searching possible executions does not have complex constraints, and the second which does have complex constraints is focused on a single execution, our approach is easier than traditional model checking. Moreover, observing that generating and validating multiple candidate schedules can be done in parallel, we have also developed a parallel constraint solving algorithm, which theoretically scales CLAP to programs with arbitrary execution length when there are sufficient computation cores.

We have implemented CLAP for C/C++ programs and evaluated it on a range of multithreaded benchmarks as well as several real world applications with intensive shared memory dependencies. Our experimental results show that CLAP is highly effective in reproducing concurrency bugs. CLAP was able to compute correct schedules that deterministically reproduce all of the evaluated bugs, and incurred only 9.3%-269% runtime overhead based on an extension of the Ball-Larus algorithm [4, 20] for collecting the thread paths. The constraint solving took 0.5s to 2280s for the sequential solver, while on an eight-core machine with our parallel solving algorithm, it typically took much less time (0.2s-63s). The computed schedules by CLAP typically contain less than three preemptive thread context switches, which is much easier to reason about for diagnosing the bug. Moreover, compared to a state of art record-replay technique LEAP [14], CLAP achieved significantly smaller overhead in both runtime (with 10%-93.9% reduction) and space (with 72%-97.7% reduction).

We highlight our contributions as follows:

- We present the design and implementation of CLAP, a new concurrency bug reproduction technique that computes shared memory dependencies through thread local path collection and constraint solving.
- We present a sound modeling of the execution constraints with respect to both the sequential consistent and TSO/PSO models. Any schedule that satisfies the constraints is guaranteed to reproduce the bug.
- We formulate the thread context switches into the constraints, which enables CLAP to produce the bug-reproducing schedule with minimal thread context switches and also bound the search space of the solver to be polynomial to the execution length.
- We present a parallel constraint solving algorithm that scales CLAP to programs with arbitrary execution length in theory.
- We evaluate CLAP on a set of real world concurrent applications. The result demonstrates the efficiency and the effectiveness of our technique.

2. Overview

We first provide an example to illustrate the key challenges in reproducing concurrency bugs. We then show how CLAP works for the example and outline the core constraint modeling.

2.1 Example

Figure 2 shows an example with two threads accessing two shared variables (`x` and `y`). Consider the two assertions `assert①` and `assert②` at line 9 and line 18, respectively. On the sequential consistent (SC) model, `assert①` will be violated if the two threads execute following the annotated interleaving `1-10-2-11-3-12-13-4-5-14-9`. However, this assertion violation is difficult to reproduce because the program contains more than `10000` different interleavings [25, 27] and a slightly different one may make the bug disappear. Worse, `assert②` will never be violated under the SC model, but can be violated under the PSO model that allows the reordering of writes to different memory addresses. For example, suppose line 4 and line 5 are reordered, `assert②` will be violated following the schedule `1-10-2-11-3-12-5-13-14-4-18`.

For the state of the art bug reproduction solutions [14, 21, 32, 41], having a small runtime perturbation is challenging, even for

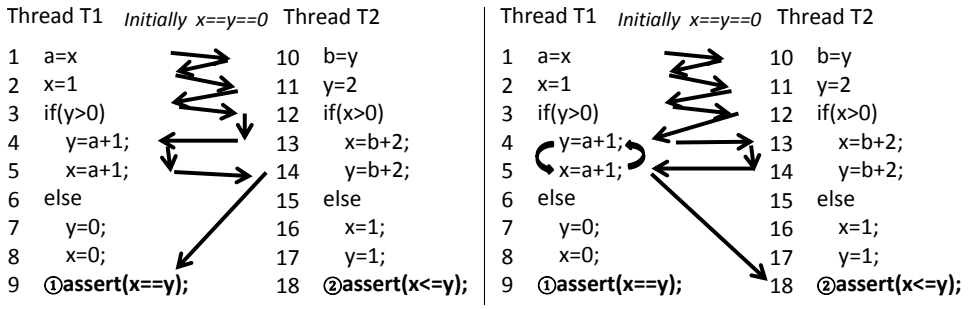

Figure 2. Example: concurrency errors on sequential consistent (left) and partial store order (right) memory models.

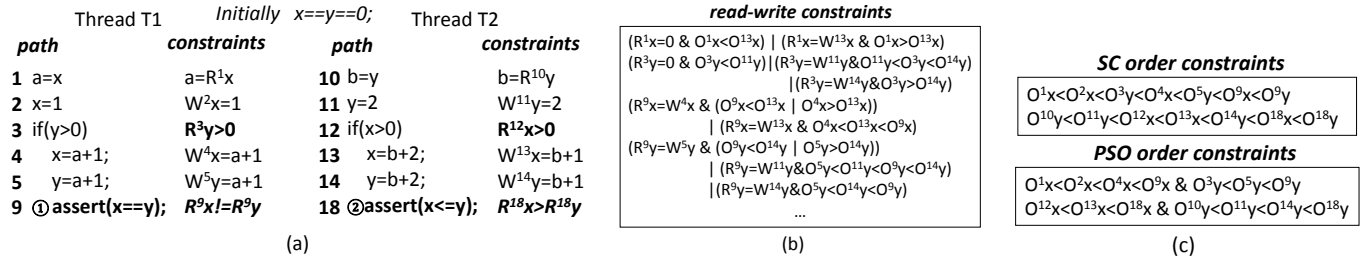

Figure 3. CLAP constraint modeling of the example program in Figure 2.

this simple example of less than 20 lines. For instance, to replay either of the assertions, there are at least 12 race pairs that need to be tracked. A more subtle but critical point is that the PSO bug might never be captured if one is not careful enough in adding locks in tracking the race orders. The memory fencing effect of locks can prevent the reordering in Figure 2 (right) to happen in test runs. And, if the runtime monitoring is disabled in production runs, the bug will surface to bite.

Both of these two assertion violations can be reproduced by the value-based approaches [23, 24, 36, 40], however, at the price of expensive logging of the value trace. Because both of the two buggy executions contain 12 reads and writes, there are 12 corresponding value cells (one per each read/write) needed to be recorded and stored into the value logs at runtime. Yet, the search space of the shared memory dependencies based on the value trace is still enormous, and the search problem has been shown to be still NP-complete in theory [11].

We next show how CLAP reproduces these two bugs without recording any race order or program state and without using any extra synchronization.

2.2 CLAP

In CLAP, we formulate the problem of reproducing concurrency bugs as a constraint solving problem, the goal of which is to compute a schedule (*i.e.*, an ordering) of the shared data accesses in the execution such that the bug can be reproduced. For simplicity, we shall refer to such access to shared data as shared access point (SAP), the read-SAP as Read, and the write-SAP as Write. At runtime, CLAP logs only the execution path of each individual thread. Then, offline, CLAP performs symbolic execution along the thread paths to collect and encode all the necessary execution constraints over the order of the SAPs. During the symbolic execution, since the value returned by each Read is unknown, we first mark it by a fresh symbolic value and later match it with a Write using the constraints.

Figure 3 shows our constraint modeling of the example program for both SC and PSO. There are two type of unknown variables: R_v^i – the value of read access to v (here v is x or y) at line i, and O_v^i – the order of the corresponding access to v in the to-be-computed schedule. For the value of each Write, given the program input, it could be either a concrete value or a value computed from the symbolic values of the Reads. To aid the presentation, we also use a symbolic variable W_v^i to denote the value of Write to v at line i in the example.

Figure 3(a) shows the path constraints. For instance, the constraints for the violation of `assert②` are written as $R_y^3 > 0 \wedge R_x^{12} > 0 \wedge R_x^{18} > R_y^{18}$, meaning that, for this assertion to be violated, both the value returned by the read of y at line 3, R_y^3, and that by the read of x at line 12, R_x^{12}, should be larger than 0, and, of course, the value returned by the read of x at line 18, R_x^{18}, should be larger than that of the read of y at line 18, R_y^{18}.

Figure 3(b) shows the read-write constraints. The idea is to match each Read with a corresponding Write, following the rule that a read always returns the value by the most recent write (on the same data). For example, consider the read of x at line 1 (R_x^1), it may return either the initial value 0, or the value written by the write access at line 13 (W_x^{13}). If R_x^1 returns 0, it should be executed before W_x^{13}, and we shall have the order constraint $O_x^1 < O_x^{13}$. Otherwise, if R_x^1 returns the value by W_x^{13} (which is b+1), it should be executed after W_x^{13}, and we shall have the order constraint $O_x^1 > O_x^{13}$. Therefore, as shown in the first row, the read-write constraint for R_x^1 is written as (R_x^1=0 $\wedge$ $O_x^1 < O_x^{13}$) $\vee$ (R_x^1=W_x^{13} $\wedge$ $O_x^1 > O_x^{13}$).

Figure 3(c) shows the memory order constraints, determined by the memory model. The memory order constraint for SC is the same as the program order among all the per-thread SAPs. For instance, we have $O_x^1 < O_x^2 < O_y^3$ in the SC order constraints, meaning that the statement at line 1 should be executed before line 2, and line 2 before line 3. For PSO, the memory order constraint is more relaxed compared to that of SC. Because reads and writes on different memory addresses are allowed to be re-ordered, the strict program order constraint is only applied to the SAPs on the same shared variable. For instance, we only have $O_x^1 < O_x^2$, but not $O_x^2 < O_y^3$, as line 2 and line 3 are accessing different data.

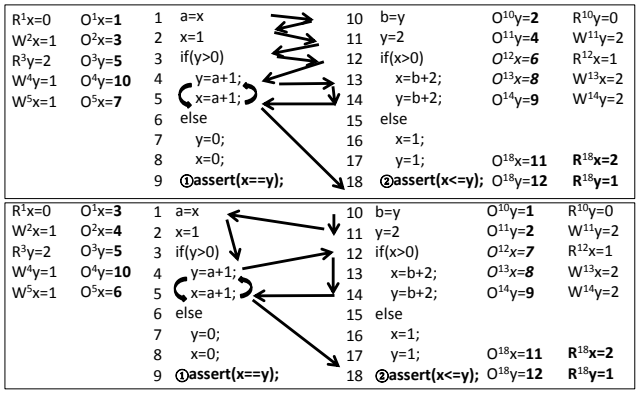

Figure 4. Two possible solutions returned by the solver for the PSO case. The first solution (top) is identical to the original schedule. The second (bottom) has the minimal thread context switches.

Taking all these constraints, CLAP invokes a SMT solver to solve them. Figure 4 shows two possible solutions returned by the solver for the PSO case (the result for SC is similar but simpler so we omit it). In the first solution (top), the computed schedule is identical to the original one. Following this schedule, at line 18, the values of `x` and `y` are `2` and `1`, respectively, and the violation of `assert`② can be reproduced. Better, in the second solution (bottom), the computed schedule has only four context switches, much fewer than that of the original schedule, but is still sufficient to reproduce the assertion violation. The power of our approach is that we can easily add additional constraints to always produce a solution like the second one, which has the minimal number of thread context switches. We will present more details on this property in Section 4.2.

The above example illustrates how CLAP works in a nutshell. We next answer the following two important questions:

- How to correctly model all the execution constraints? For example, for presentation easiness, we do not have any synchronization in the example program. How to model them? (§3)
- How difficult it is to solve the constraints? How to scale the constraint solving task in our approach? (§4)

3. CLAP Execution Constraint Modeling

From a high level view, we encode all the necessary execution constraints into a formula Φ containing two type of unknown variables: V - the symbolic value variables denoting the value returned by Reads, and O - the order variables denoting the order of SAPs in the schedule. Φ is constructed by a conjunction of five sub-formulae:

$$\Phi = \Phi_{path} \wedge \Phi_{bug} \wedge \Phi_{so} \wedge \Phi_{rw} \wedge \Phi_{mo}$$

where Φ_{path} denotes the path constraints, Φ_{bug} the bug predicate, Φ_{so} the inter-thread order constraints determined by synchronizations, Φ_{rw} read-write constraints over Reads and Writes, and Φ_{mo} the memory order constraints determined by the memory model. These constraints are complete because the intra-thread data and control dependencies are captured by Φ_{path} and Φ_{bug}, and the inter-thread dependencies are captured by Φ_{rw}, Φ_{so} and Φ_{mo}. We next present each of the constraints in detail.

3.1 Intra-Thread Constraints

The intra-thread constraints serve two purposes: they force every thread in the computed execution to follow the same control flow as the corresponding threads in the original execution, and they force the same bug to happen, *i.e.*, the same assertion to fail. Forcing the same control flow eliminates the search over all possible branches that common to the static techniques, thus simplifying the problem.

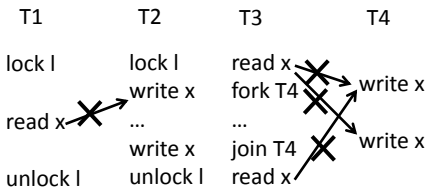

Figure 5. An illustration of the synchronization constraints

Path Constraints ***(Φ_{path})*** The path constraints consist of the conjunction of the path conditions for all threads. The thread path conditions are collected by a symbolic execution of the program following the recorded thread path profiles. On each branch instruction, a new constraint that specifies the condition of the branch taken by the thread, is generated and added to the path constraint.

Bug Manifestation Constraint ***(Φ_{bug})*** In our modeling, the bug is not limited to a crash or a segfault, but general to all properties over the program state. Φ_{bug} is modeled as a predicate over the final program state. For example, a null pointer dereference *x.f()* can be defined as $V_x = NULL$, and a buffer overflow error can be defined as $V_{len} > V_{size}$. In practice, the predicate could be extracted from the core dump when the program crashed, or from the program assertion when the assertion is violated at runtime, or from any other properties checked at runtime.

3.2 Inter-Thread Constraints

Inter-thread constraints are of two kinds: the synchronization constraints that govern the control flow between multiple threads, and memory constraints that govern the data flow between threads. We cover the synchronization constraints and then the memory constraints. For the memory order constraints Φ_{mo}, we also prove that our approach is applicable to both SC and a range of relaxed memory models such as TSO and PSO.

Synchronization Order Constraints (Φ_{so})

We model the synchronization order constraints according to the locking and the partial order semantics. The locking semantics determines that two sequences of Reads/Writes protected by the same lock should not be interleaved, while the partial order semantics determines that one SAP should always happen before the other. For example, consider the program in Figure 5. The read of `T1` cannot be mapped to the first write of `T2`, because it cannot be executed between the two writes due to the lock. The first read of `T3` cannot be mapped to any of the two writes of `T4`, because it must happen before them due to the *fork* operation; similarly, the second read of `T3` cannot be mapped to the first write of `T4`, because the second write of `T4` always executes between them due to the *join* operation.

Φ_{so} is encoded as the conjunction of the locking constraints and the partial order constraints. We extract the synchronization operations for each thread and group them by the lock, signal, or thread they operate on. We use the memory address in the symbolic execution to identify locks and signals. For thread objects, because each thread creates its children threads in a deterministic order (i.e., following the program order), we create a consistent identification for all threads based on the parent-children order relationship. For example, suppose a thread t_i forks its jth child thread, this child thread is identified as $t_{i:j}$, and when the thread $t_{i:j}$ forks its kth child thread, the new thread is then identified as $t_{i:j:k}$. Readers can find more details in our previous work [13].

Locking constraints The locking semantics is concerned with the *lock* and *unlock* operations only. For each lock object, we first extract the set of *lock/unlock* pairs that operate on it, following the program order locking semantics, *i.e.*, an unlock operation is paired with the most recent lock operation on the same lock by the same thread. For each *lock/unlock* pair, we then enumerate all the other pairs in the set and add their corresponding order constraints with the chosen pair. Specifically, let S denote the set of *lock/unlock* pairs on a certain lock and consider a pair a_l/a_u. The constraint is written as follows:

$$\bigwedge_{\forall a'_l/a'_u \in S} O_{a_u} < O_{a'_l} \quad \vee$$

$$\bigvee_{\forall a'_l/a'_u \in S} (O_{a_l} > O_{a'_u} \wedge \bigwedge_{\forall a''_l/a''_u \in S} O_{a''_l} > O_{a_u} \vee O_{a''_u} < O_{a'_l})$$

The constraint above states that the *lock* operation a_l acquires either the initial lock, or the lock released by another *unlock* operation a'_u. In the first case, the order of the *unlock* operation a_u should be smaller than that of all the other *unlock* operations. In the second case, the order of a_l should be larger than that of a'_u, and for any other *lock/unlock* pair a''_l/a''_u, either the order of a''_l is larger than that of a_u, or the order of a''_u is smaller than that of a'_l. The total size of the locking constraints for each lock object is $2|S|^2 + 2|S|$.

Partial order constraints The partial order semantics is related to the thread *fork/join* and *wait/signal* operations. For *fork* and *join*, their order constraint is simple to model, because they can only be mapped to one unique operation. For *fork*, it is the corresponding *start* operation of the newly forked thread, and for *join*, it is the *exit* operation of the joined thread. Therefore, we can simply add the constraints that the order of a *fork* operation is smaller than that of its corresponding *start* operation. Similarly, the order of a *join* operation is larger than that of its corresponding *exit* operation. For *wait* and *signal* operations, the constraint is slightly more complex. Because a *wait* operation could have multiple candidate *signal* operations that it could be mapped to, we have to enumerate all the candidates. Also, because a *signal* operation can only signal at most one *wait* operation, we need to constrain the number of operations a *signal* can be mapped to.

To model this constraint, we introduce a set of binary variables for each *signal* operation. Each binary variable denotes whether the *signal* operation is mapped to a *wait* operation or not. We then constrain the sum of these binary variables to be less than or equal to one. Consider a *wait* operation a_{wt}, and let SG denote the set of *signal* operations that operate on the same *signal* variable as that of a_{wt} and by a thread different from that of a_{wt}. We model the constraint as follows:

$$(\bigvee_{\forall a_{sg} \in SG} O_{a_{sg}} < O_{a_{wt}} \wedge b_{a_{wt}}^{a_{sg}} = 1) \bigwedge \sum_{x \in WT} b_x^{a_{sg}} \leq 1$$

In the constraint above, WT denotes the set of wait operations the signal operation a_{sg} can be mapped to, and $b_x^{a_{sg}}$ the binary variable that indicates whether a_{sg} is mapped to a wait operation, x, or not. The total size of the constraints is $2|SG||WT| + |SG|$.

Memory Model Constraints

A crucial factor of the constraint modeling is the memory model under which the buggy execution occurred, as it determines what values each Read could return. The memory model is a parameter to our system in the sense that we take a declarative specification of the memory model, and combine it with the concrete program actions from the trace to produce a set of constraints that determine which reads can see values from which writes in the program. Conceptually, this approach is taken from previous work like MemSAT [3]; indeed, we could employ specifications in that style directly, since our thread path constraints and SAPs correspond to the elements of that model.

However, in this work, we are focusing on memory models for bus-based shared memory machines in which there is a global order among memory operations as they appear to the main memory. This allows us to simplify and use O as a total order; however, this is not an inherent limit and we could employ other kinds of models directly. Specifically, in CLAP, we currently implement Sequential Consistency (SC), Total Store Order (TSO), and Partial Store Order (PSO). We next discuss how Reads and Writes are constrained with respect to the memory order O, and then how O is constrained to model SC, TSO and PSO.

Read-Write Constraints (Φ_{rw}) For a Read, it may be mapped to a Write by the same or a different thread, depending on the order relation between the Writes. Consider a Read r on a shared variable s, and let W denote the set of Writes on s. We use O_r to denote the order of r, V_r the value returned by r, and O_{w_i} the order of the Write w_i in W. Φ_{rw} is written as:

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

The constraint above states that, if a Read is mapped to a Write, for this Write, its order is smaller than that of the Read, and there is no other Write that is between them. In our constraint construction, we first group all the Reads/Writes by the accessed memory address, and then encode the read-write constraint for each Read. Let N_r and N_w denote the number of Reads and Writes on a certain shared address, the size of the read-write constraints is $4N_rN_w^2$, which is polynomial to the size of SAPs. Note that this constraint is identical to the corresponding constraints in [3], with the change of O to be a single global order.

Memory order constraints (Φ_{mo}) For the sequential consistent memory model, the memory order for SAPs is the same as the program order. Consider two consecutive SAPs a and b by a thread t in the program order, their order constraint is written as $O_a < O_b$. We encode this constraint for each pair of successive accesses by each thread, following the recorded path profile. The size of the constraints is linear to the total number of SAPs in the execution.

For TSO, it does not require the program order and allows reordering Read and Write on different addresses. The hard constraint is that the original program order among all the Writes and among all the Reads is preserved. Hence, for all Writes and for all Reads by the same thread, we model the same order relation as the program order. In addition, for Reads, we need to make sure each Read returns the value written by the most recent Write on the same data. Therefore, for each Read, we first find the two Writes that 1) access the same address as that by the Read, and 2) are immediately before and after the Read, respectively, in the program order. We then model the order between the Read and the two Writes to be the same as that in the program order. Compared to TSO, PSO further relaxes the order relation between Writes and between Reads on different addresses. Our constraint model for PSO is hence similar to TSO, except that the order relation between Writes and between Reads on different addresses are removed.

Soundness With respect to different models, although the execution order of the SAPs by each thread may or may not be the same as the program order, we prove that our approach of computing a schedule among the SAPs is sound to SC/TSO/PSO memory models. More formally, we have the following theorem:

Theorem 1. *For SC/TSO/PSO, there always exists a schedule of the SAPs in which each load return the value by the most recent store to the same address, and following which the same program state can be achieved as that of the original buggy execution.*

Proof For all these models, once a store is visible to a second processor, it is instantly visible to all processors. It is impossible for two processors to observe different orders for any pair of stores. Hence, there always exists a total ordering of all the stores under these models. For loads, their order in the schedule can be determined by placing them after the stores whose value they accessed in the original execution and before the subsequent store, such that a load always return the value by the most recent store.

4. Constraint Complexity and Scalable Solving

Our constraint solving task is much easier than conventional ones involving complex constraints such as strings and non-linear mathematics. In CLAP, the solver only needs to compute a solution for the order variables that essentially maps each Read to a certain Write in a discrete finite domain (*i.e.*, the set of Writes on the same data as that of the Read), subject to the order constraints. To further scale CLAP, we have also developed two core techniques to improve the performance of constraint solving: preemption bounding and parallel solving. We next conduct a brief analysis of the constraint complexity, followed by the detailed discussion of the two techniques.

4.1 Constraint Complexity

Let N_{br} denote the number of conditional branching instructions in the execution. Let N_{sync} denote the number of synchronizations. And let N_{sap}, N_r, and N_w denote the number of SAPs, read-SAPs, and write-SAPs, respectively. Among the synchronizations, let N_l the number of *lock/unlock* pairs, and N_{wt}/N_{sg} the number of *wait/signal* operations.

Recall Section 3 that our constraint formulae consist of the path constraints Φ_{path}, the bug manifestation constraint Φ_{bug}, the synchronization constraints Φ_{so}, the read-write constraints Φ_{rw}, and the memory order constraints Φ_{mo}. The size of Φ_{path} is equal to N_{br} because each branching instruction generates a new conjunction clause over the symbolic value variables. Recall Section 3.2, the size of Φ_{so} is $2N_l^2$+$2N_l$+$2N_{sg}N_{wt}$+N_{sg}; assuming all SAPs are accessing a single shared variable, the worst case size of Φ_{rw} is $4N_rN_w^2$, and the worst case size of Φ_{mo} is equal to N_{sap}+N_{sync} (for all the three memory models). Because N_{sap}=N_r+N_w+N_{sync} and normally $N_{sap} \gg N_{sync}$, the size of Φ_{rw} is far larger than Φ_{mo} and Φ_{so}. Therefore, the total size of the constraints can be approximated as N_{br}+N_{sap}^3.

In sum, the worst case complexity of our constraints is linear to the number of conditional branches and cubic to the number of shared data accesses in the execution.

4.2 Thread Context Switch Constraint

Researchers have observed that most real world concurrency bugs can be manifested by a small number of thread context switches [28]. In CLAP, this observation can be directly encoded as additional constraints to bound the search space of the solver.

Recall that each SAP is assigned with an order variable, representing its position in the computed schedule. Our basic idea is to use the order difference between consecutive SAPs within the same thread to determine whether a context switch occurs between them. If the execution of two consecutive SAPs is not interleaved by other threads, their order difference will be equal to one; otherwise, the difference will be larger. To obviate modeling the non-preemptive context switches (as they always occur) and to create a uniform constraint for different memory models, we group a sequence of SAPs into segments, and use the number of interleaved segments to approximate the size of real context switches. This is a good approximation in practice because the context switch number is often small.

We first extract all the synchronization operations that cause non-preemptive context switches, including *wait*, *join*, *yield*, and *exit*. For brevity, we call them *must-interleave* operations. We use these operations to divide the SAPs into segments for each thread. Each segment contains only one must-interleave operation which leads or ends the segment. Note that the must-interleave operations are not allowed to be reordered (because they are synchronizations). In the final schedule, the leading (ending) operation in each segment will always have the smallest (largest) order among all the SAPs in the same segment. For each segment, we then use the difference between the orders of the ending and the leading operations to determine whether a context switch occurs or not in the segment. Let S_t denote the set of segments by thread t, a_l and a_e the leading and ending SAP in a segment s, and N_{cs} the specified context switch bound. The constraint is written as:

$$\Sigma_{t\in T}\Sigma_{s\in S_t}\begin{cases}1 & \text{if } O_{a_e}-O_{a_l} > |s|-1\\ 0 & \text{otherwise.}\end{cases} \leq N_{cs}$$

The above formula states that the total number of interleaved segments for all threads is bounded by N_{cs}.

Minimal thread context switches The method above not only bounds the search space, but can also be used to produce the schedules with minimal number of context switches. Specifically, we can start from the constraint with zero thread context switch, and increment the context switch number when the solver fails to return a solution. We repeat this process until a solution is found. In this way, we can always produce a schedule with the fewest thread context switches among all the bug-reproducing schedules.

4.3 Parallel Constraint Solving Algorithm

Our core idea to parallelize the constraint solving is to treat the memory order constraints Φ_{mo} separately from the other constraints. We first generate candidate schedules that satisfy Φ_{mo}, and then employ the solver to validate each candidate schedule, *i.e.*, checking if it satisfies all the other constraints. This method has two salient features. First, we can generate different schedules and validate them in parallel. Each single schedule generation and validation is independent and fast (requiring only a linear scan of the SAPs and the constraints). The whole constraint solving task can then be divided into many independent subtasks that each works on a candidate schedule. Second, we can generate the schedules with the increasing number of context switches, allowing us to bound the search space similar to that of the bounded dynamic schedule exploration approaches [27, 28].

A key challenge in this approach is how to avoid generating duplicated schedules. To address this problem, we represent the context switches in a schedule by a set of *context switching points* (CSP). A CSP is a location in the schedule where a context switch occurs. It can be uniquely identified in an ordered way by a triple ($t1$,k,$t2$), denoting that a thread $t1$ is interleaved by another thread $t2$ immediately before the kth SAP of $t1$. Different CSPs are then combined together into a CSP set, and the set is used to guide the schedule generation procedure.

Preemption-bounded schedule generation Given a context switch number c (= 0,1,2,...), we first generate all the CSP sets of size c. For each CSP set (including the empty set), we fork a separate process that generates the corresponding schedules. Each generated schedule is then validated by the solver to determine its correctness. Our algorithms for generating schedules for SC and for TSO/PSO are mostly the same with only slight difference. The complexity of our algorithm is linear to the total number of SAPs. The size of the schedules is bounded by $\binom{c}{N}(N+c)!$, where c is the number of context switches and N is the total number of SAPs.

SC We associate each thread with a stack that contains the SAPs of the thread in the program order. Starting from the main thread, each schedule-generation process attempts to pop up one SAP from the thread's stack, subject to a condition: the input set S contains no CSP ($t1$,k,$t2$) that matches the current thread (t) and the current SAP (i). If the condition is not met ($t = t1$ and $i = k$), it means that a context switch should happen at this point, so we jump to the stack of the interleaving thread ($t2$). This process is repeated until all SAPs are popped up.

Note that each process usually generates more than one schedule. This is because, when the stack of the current thread becomes empty, we have multiple remaining threads to jump to and each choice will generate a different schedule. In that case, we remove the current thread and fork multiple children processes to continue the schedule generation. Each child process jumps to the stack of a different thread in the remaining thread set and repeat the process.

TSO/PSO As Reads/Writes are allowed to be reordered in TSO and PSO models, we can no longer use a stack to represent the memory order constraints. Instead, we use a tree structure (called SAP-tree) to represent the order relation between SAPs for each thread. Each node in the SAP-tree represents a SAP and the parent-child relation between nodes represents a store-load data dependence or a memory-model determined order relation. In TSO, the parent-child relation is applied on all Writes in the program order, while in PSO it is applied on the Writes for each memory address.

Similar to that of SC, the schedule generation process for TSO/PSO repeatedly removes an ancestor node (which has no parent) from a SAP-tree, until the SAP-trees for all threads become empty. Whenever there are multiple ancestor nodes in the tree (because an ancestor node with multiple children nodes may be removed in the previous step), we fork the same number of subprocesses each of which continues the schedule generation starting with one of the ancestor nodes.

5. Implementation

We have implemented CLAP on top of LLVM and KLEE-2.9 [5] with the STP solver [10]. To adapt KLEE to CLAP, we made four key modifications. First, KLEE works only with sequential programs, thus we extended it to support multiple threads. Specifically, we modified KLEE to spawn a new instance for each new thread and added the necessary extension to uniquely identify threads (see Section 3). Second, we changed KLEE to only follow the recorded path profile for each thread, and to only collect the path constraints without solving them. At the end of the run, we unify the constraints collected from each thread as the path constraints. Third, we adapted KLEE to return a new symbolic value for the load instructions that access shared memory addresses. Fourth, we modified the constraint solving utility, Kleaver, to incorporate the path constraints with the other execution constraints. We wrote our own constraint generation engine based on the SAPs collected from the thread local paths during the symbolic analysis phase.

Thread Local Path Collection Path monitoring is a pluggable component in our approach. Ideally, we can use hardware monitoring techniques such as the work of Vaswani *et al.* [31] which has negligible (around 0.6%) overhead. Our current implementation is an extension of the classical Ball-Larus algorithm [4, 20] based on a LLVM function pass. It works for multithreaded C/C++ programs that use PThreads and incurs 9.3%-269% runtime overhead in our experiment. From a high level view, we break the whole path into a sequence of segments, each of which is a Ball-Larus (BL) profile. A new segment starts when a new function is called, or an intra-procedural path is re-entered. We analyze the control flow graph of each function and insert instrumentations at the following points: the entrance/exit of each function, the beginning of basic blocks that have a back edge, and the branching points analyzed by the Ball-Larus algorithm. Each function is uniquely labeled (represented by a number), and each BL path inside the same function is also uniquely labeled (computed as the sum of the encoded edge weights at runtime). During the execution, we collect the whole path profile for each thread by recording the sequence of the labels for the function calls and for the BL paths. To distinguish between the BL paths that have the same label but are in different functions, we also log the exit of each function to demarcate the sequence of BL paths. The recorded labels are then decoded to produce the whole path profile, and to guide our symbolic analysis.

Shared Memory Access Identification Identifying shared data accesses is orthogonal to our approach but important for reducing the size of the constraints. Naively, we would mark all loads and stores as shared data accesses. This would produce a huge amount of unnecessary constraints, since the constraints with only the thread local data accesses are essentially redundant. A better way is to detect shared data accesses at runtime, where the accessed address is available for every memory operation. To overcome the virtual address recycling issue, we also need to record malloc/free operations to identify truly shared memory locations. However, this method does not fit our design as it inevitably causes additional program slowdown. In CLAP, we perform a static thread sharing analysis based on the Locksmith [30] race detector to identify shared variable accesses, including also the treatment of shared pointers and heap locations. Though being conservative, it is very effective to determine thread-local locations and, more importantly, does not introduce any runtime cost.

Deterministic Bug Reproduction Our application level thread scheduler is implemented based on Tinertia [16]. We add instrumentations before each SAP in the program and employ the dynamic thread interpolation to intercept PThread library calls. Through the inserted instrumentations, we are able to enforce the thread execution order of the SAPs to strictly follow that of the computed bug-reproducing schedule. Whenever a thread is going to execute a SAP, we first check the schedule to decide whether it is the correct turn for the thread to continue execution. If not, we put the thread in a postponed thread queue and make it wait until all the SAPs before it have been executed.

Challenges and Treatments

External Function Calls A known challenge in symbolic execution is the existence of external function calls that make the path constraints incomplete. Although KLEE tried its best to simulate the external environments (i.e., file systems and system libraries), it still suffers from this problem when facing unresolved external calls. However, this problem is not fundamental to CLAP because, for bug reproduction, we can record the runtime input/return values of all the external calls, and use the value pairs to construct the constraints of the external interfaces. A negative side of this method is that it would over-approximate the behavior of the external functions, which reduces the scheduling space explored by CLAP. In practice, to avoid limiting the capability of CLAP too much, we choose to first resolve the external functions as much as possible. In addition, we try to avoid flagging as symbolic the variables that have value flows to unresolved external calls. In our experiments, the external functions are seldom related to symbolic variables, and we did not face much difficulty in this problem.

Symbolic Address Resolution Another known issue in symbolic execution is the resolution of symbolic memory addresses. As KLEE does not perform any pointer tracking, when facing reads or writes to symbolic addresses, it would exhaustively search all allocated objects and forks execution for each possible base object. This usually takes a long time and also produces quite a number

of unnecessary memory states. In CLAP, since the symbolic analysis phase follows the recorded thread execution path, we only need to explore one memory state for each thread. To avoid the state explosion, we choose to delay the object resolution for symbolic addresses to the constraint solving phase. Specifically, during symbolic execution, we keep track of the base object for each memory operation. For each base object, we maintain an ordered list of writes to symbolic addresses performed so far. Each write is remembered as a pair consisting of the location that was updated, and the expression that was written to that location. For any subsequent read, the loaded value is resolved from the ordered list with a set of constraints. As an example, suppose the ordered list of writes to an array is $\langle a[0] = 0, a[1] = 0, \ldots, a[n] = 0, a[i1] = x, a[i2] = y\rangle$. For a read $b = a[j]$, we create the constraint $(j = i2 \wedge b = y) \vee (j \neq i2 \wedge j = i1 \wedge b = x) \vee (j \neq i2 \wedge j \neq i1 \wedge b = 0)$, and add it to the path constraints.

Input Non-determinism Currently, CLAP assumes that the program input is deterministic. CLAP does not record the program input as we mainly address the problem of scheduling non-determinism in this work (which is more significant and difficult). If the program input is non-deterministic, CLAP might not be able to reproduce the bug. Nevertheless, similar to the treatment of external functions, this problem can be addressed by recording and enforcing the same input value during the bug reproduction execution.

6. Experiments

We have evaluated CLAP on a variety of real world multithreaded C/C++ applications with known or seeded bugs collected from [16, 39], including *pbzip2-0.9.4*, a parallel implementation of bzip; *aget-0.4.1*, a parallel FTP/HTTP downloading utility; *ctrace*, a multithreaded tracing library; *pfscan*, a parallel file scanner; *swarm*, a parallel sort implementation; *bbuf*, a shared bounded buffer implementation; and the *apache-2.2.9* web server. To assess the limit of CLAP, we also examined with *simp_race*, a simple racey program [16], and *racey*, a benchmark for deterministic replay systems [38]. To evaluate CLAP for reproducing bugs on relaxed memory models, we also examined with the implementations of three classical mutual exclusion algorithms - *dekker*, *bakery* and *peterson*.

Setup CLAP works in three phases: 1) online path collection; 2) offline constraint generation and solving; 3) bug replay execution. In phase 1, the thread paths are dumped to disk when the bug occurs. Due to the rare erroneous thread interleaving, most concurrency bugs are difficult to manifest. To trigger the bug in our experiment, we typically inserted timing delays at key places in the program and ran it many times until the bug occurred, and we added the corresponding assertion to denote the bug manifestation. In phase 2, the constructed constraints were first saved to a file which is then provided to the solver to compute a schedule. In phase 3, CLAP enforced the replay to follow the computed schedule to reproduce the bug. All our experiments were conducted on an eight-core 3GHz machine with 16GB memory and Linux 2.6.22.

6.1 CLAP Bug Reproduction Effectiveness

Table 1 summarizes our experimental results. Overall, CLAP is highly effective in reproducing concurrency bugs. For all the eleven evaluated bugs and injected violations including three relaxed memory model related failures, CLAP is able to reproduce all of them by producing a schedule with a small number of thread context switches. Most of the computed schedules contain less than three preemptive context switches, except for the *racey* benchmark, in which at least 276 context switches are needed to reproduce the injected violation. The size of the constraints range from 341 to more than 400M clauses with 26 to 2M unknown variables, and the total time for constructing and solving these constraints ranges from 2s to around 50 mins. We next discuss the results for several interesting applications in detail.

pbzip2-0.9.4 contains a known order violation bug frequently studied in concurrency defect analysis techniques [16, 18, 19, 35, 39]. The main thread communicates with a set of consumer threads through a FIFO queue with a mutex protecting the data accesses. The bug occurs intermittently when the main thread nullifies the mutex before some consumer threads are still using it, causing program crashes. The buggy execution contains four threads compressing a 80K file. There are 18 variables identified as shared (including all the global variables and the variables associated with the FIFO queue) by the static escape analysis [30] and are marked as symbolic. Upon crash, it executed a total number of 4K instructions with 65 SAPs and 473 conditional branches (excluding calls to external libraries). It took CLAP 4s to construct the symbolic constraints, containing 102 unknown variables and 5K clauses. The STP solver took around 5s to compute a bug-reproducing schedule with two preemptive thread context switches.

apache-2.2.9 bug #45605 is a multi-variable atomicity violation between a set of listener threads and worker threads on accessing a shared queue data structure, causing an assertion violation that finally crashes the server. We started four clients to simultaneously send a number of requests until the assertion was violated. The collected path profiles contain 28 threads executing a total number of 6.8M instructions with 962K branches. We identified a total number of 22 variables as shared symbolic variables in the buggy run. To bound the search space, we constrained the size of preemptive thread context switches to be less than three. It took CLAP around 13 minutes to collect and encode the execution constraints, which contain 81K unknown variables and 10M clauses. CLAP solved the constraints in 344s and produced a bug-reproducing schedule with three preemptive thread context switches.

racey is a specially designed benchmark [38] with numerous intentional races that make it very likely to produce a different output if a different race occurs. We assessed the bug reproduction capability of CLAP by applying it to reproduce the same output in racey. There are three shared variables: a 64-union array upon which the core computation between threads is operated, and two volatile variables for coordinating the start of the threads. To avoid out of memory error, we set the loop iterations (*MAX_LOOP*) to 500000. The test execution contained 3 threads, 7.1M branches, 93M instructions, and 1.36M SAPs. It took CLAP around 15 minutes to collect the constraints containing 513M clauses and 2M unknown variables. CLAP took 38 minutes to solve the constraints and successfully computed a schedule with 276 context switches. Since racey is a benchmark specifically designed to have many races, it is an outlier in our data. It does not follow the observation that most bugs require few thread switches to reproduce, and hence it causes worst-case behavior in our system. Note that real programs have much better results.

Relaxed memory model bugs

Dekker's and Peterson's algorithms, and Lamport's Bakery algorithm all work well for the SC model, but not for TSO and PSO models. We evaluated with them to demonstrate the capability of CLAP for reproducing concurrency bugs on relaxed memory models. To trigger the bugs, we simulated the memory model effects by actively controlling the value returned by shared data loads in a similar style to [9]. For TSO, we simulated a FIFO store buffer for each thread, and for PSO, we simulated multiple FIFO store buffers, with one per shared variable.

Bakery We forked four worker threads, each of which increments a shared integer variable by one in the critical section. Due to the

Program	LOC	#Threads	#SV	#Inst	#Br	#SAPs	#Constraints	#Variables	Time (secs) symbolic	Time (secs) solve	#cs	success?
sim_race	75	5	2	103	18	23	341	26	0.8	0.5	0	**Y**
pbzip2	1.8K	4	18	4203	473	65	5324	102	4	5	2	**Y**
aget	1.2K	4	30	39401	4017	1951	1366923	2485	125	89	1	**Y**
bbuf	381	5	11	2643	189	64	2784	75	3	5	1	**Y**
swarm	2.2K	3	13	840193	73728	1265	1226098	1776	27	64	2	**Y**
pfscan	925	3	13	2261926	287713	2864	7255156	3101	332	160	3	**Y**
apache	643K	28	22	6806939	961779	81237	10153562	15534	770	344	3	**Y**
racey	200	3	3	93035842	7133586	1361588	513086300	2010082	857	2280	276	**Y**
bakery	73	5	3	1181	182	218	16355	331	1	8	1	**Y**
dekker	48	3	3	223	28	39	699	52	1	2	1	**Y**
peterson	44	3	3	215	28	35	696	48	1	2	1	**Y**

Table 1. Overall results - Columns 3-6 report the number of threads (*#Threads*), the number of shared variables (*#SV*), the number of executed instructions (*#Inst*) and branches (*#Br*) in the original buggy execution. Columns 7 reports the number of shared data accesses (*#SAPs*) in the schedule. Columns 8-9 report the size of the constraints (*#Constraints*) and the number of unknown variables in it (*#Variables*). Columns 10-11 report the symbolic analysis time (*Time-symbolic*) for collecting the constraints, and the constraint solving time (*Time-solve*) using the STP solver on a single core. For all the evaluated programs, CLAP was able to compute a bug-reproducing schedule, and most of the computed schedules contain less than 3 context switches. The total time for collecting and solving the constraints ranges from 2s to 50mins.

bug, the mutual exclusion for executing the critical section does not hold, and threads accessing the shared variable can race with each other, which may produce a wrong final result of the shared integer. The recorded path profiles of the buggy execution contain 5 threads with a total of 1181 instructions and 182 branches. It took CLAP less than 1s to collect the constraints that contain 331 unknown variables and 16K clauses for both TSO and PSO. CLAP solved the constraints in 8s and produced a bug-reproducing schedule with one preemptive thread context switch.

Dekker/Perterson We forked two threads each of which loops twice in the critical section for incrementing a shared integer. Due to the bug, the final result of the shared integer could be wrong. The recorded path profiles of *dekker* contain 3 threads with a total of 223 instructions and 28 branches. It took CLAP less than 1s to collect the constraints that contain 52 unknown variables and 699 clauses for both TSO and PSO. CLAP solved the constraints in 2s and produced a bug-reproducing schedule with one preemptive thread context switch. The result for *peterson* is similar, as Peterson's algorithm is a slightly simplified version over Dekker's algorithm.

6.2 CLAP Runtime Performance

We compared the runtime performance of CLAP with an implementation of LEAP [14], which is one of the state of art record-replay techniques that track shared memory dependencies. Because none of the concurrency bug reproduction tools [1, 21, 29, 32, 40, 41] is available, we choose LEAP for the reason that it incurs minimum implementation bias and it puts the quantifications of various runtime characteristics of CLAP into perspective. We ran each benchmark under three different settings – natively (without instrumentation and logging), with LEAP, and with CLAP, and we measured the corresponding execution time and log size.

Table 2 reports the results. All data were average over five runs. As expected, since LEAP requires synchronizations to record the shared variable access orders, its overhead is large when there are intensive shared memory dependencies in the execution. Because most of these benchmarks have frequent shared data accesses (especially *racey*, in which the majority of memory operations are on shared data), both the runtime overhead for recording the shared data accesses and the space needed for storing the log are significant for LEAP. The runtime overhead of LEAP ranges from 21.4% in *pbzip2* to as large as 4289% in *racey*, and the corresponding space cost ranges from 19.5K-68.2M. Compared to LEAP, CLAP incurred much less runtime overhead and space cost, because it only records the thread local paths and does not use any synchronization. The runtime overhead of CLAP ranges from 9.3%-269%, which achieves 10% to 93.9% reduction compared to LEAP. For the space cost (1.1K-3.81M), the improvement by CLAP is also significant, with 72% to 97.7% reduction compared to LEAP.

Program	Schedules #worst	Schedules #gen(#cs)	Schedules #good	Time par	Time seq
sim_race	$> 10^6$	128(0)	3	0.3s	0.5s
pbzip2	$> 10^{15}$	140(2)	8	0.3s	5s
aget	$> 10^{40}$	13725(1)	16	16s	89s
bbuf	$> 10^{20}$	324(1)	36	0.6s	5s
swarm	$> 10^{30}$	30855(2)	27	19s	64s
pfscan	$> 10^{50}$	118714(3)	12	63s	160s
apache	$> 10^{100}$	5634627(3)	15	195s	344s
racey	$> 10^{10000}$	2528316(2)	**0**	**2h**	2280s
bakery	$> 10^{25}$	1071(1)	22	1.5s	8s
dekker	$> 10^6$	31(1)	12	0.2s	2s
peterson	$> 10^6$	28(1)	11	0.2s	2s

Table 3. The performance of parallel constraint solving

6.3 CLAP Parallel Constraint Solving Performance

We have evaluated the performance of our parallel constraint solving algorithm on an eight-core machine. We repeated the schedule generation process with a larger context switch number (recall Section 4.3) until we found at least one correct schedule that satisfies the constraints. Each task of generating and validating one schedule is handled by a separate thread. Because normally there exist a set of correct schedules and multiple threads may work on them simultaneously, we typically have found multiple correct schedules before the whole process is terminated.

Table 3 reports the results. Columns 2-4 report the worst number of possible schedules (computed according to the theoretical results in [25, 27]), the number of generated schedules before we stopped (#cs–the largest number of context switches among these schedules), and the number of correct schedules among the generated ones. Column 5 reports the total amount of time it took for finding these correct schedules. As the table shows, although the worst number of different schedules is exponential, CLAP successfully generated correct schedules for most of the benchmarks within 200s.

For comparison, Column 6 shows the constraint solving time of the sequential version (as also reported in Table 1). For most benchmarks, using our parallel algorithm is much faster than the

Program	Native	Time			Space		
		LEAP (Overhead%)	CLAP (Overhead%)	Reduction%	LEAP	CLAP	Reduction%
sim_race	2ms	4ms (-)	4ms (-)	-	448B	126B	↓**72%**
bbuf	2ms	6ms (-)	4ms (-)	↓**33%**	12.2K	1.1K	↓**91%**
swarm	68ms	0.770s (**1032%**)	0.101s (**48.5%**)	↓**87%**	9.20M	215.6K	↓**97.7%**
pbzip2	0.140s	0.170ms (**21.4%**)	0.153ms (**9.3%**)	↓**10%**	19.5K	1.8K	↓**91%**
aget	0.231s	0.490s (**112%**)	0.270s (**17%**)	↓**45%**	683.8K	24.3K	↓**96.4%**
pfscan	0.135s	1.537s (**1172%**)	0.260s (**92.6%**)	↓**83.1%**	1.61M	330.5K	↓**79.5%**
apache	0.185s	0.248s (**34%**)	0.220 (**19%**)	↓**11.3%**	15.4M	2.30M	↓**85%**
racey	0.262s	11.5s (**4289%**)	0.705 (**269%**)	↓**93.9%**	68.2M	3.81M	↓**94.4%**

Table 2. Runtime and space overhead comparison between CLAP and LEAP

sequential solution. For example, for the constraint of *apache*, it took the sequential solver 344s to return a correct schedule, while with the parallel algorithm, CLAP generated 27 correct schedules in 195 seconds. The only exception is *racey*, which we use as a stress test for worst case behavior. Using the parallel algorithm, we did not find a correct schedule within two hours. The reason is that a correct schedule for *racey* should contain at least 276 thread context switches, but in two hours we did not even finish enumerating the schedules with only two context switches, due to the large number of shared data accesses in the execution.

6.4 Discussion

Long running traces Our evaluation results demonstrate that CLAP has good scalability with fairly substantial traces in real-world executions. For very long runs, reproducing the failure is more challenging that the solver may not be able to find a solution within a reasonable time budget. In such cases, we need to break up the execution so that each execution segment has tractable size of constraints. Checkpointing is a common technique used in such contexts. We plan to integrate CLAP with checkpointing in future.

Recording synchronizations Previous work [32, 42, 43] has shown that recording synchronization operations is lightweight for many applications. Recording the synchronization order can also reduce the size of generated constraints, and it is easy for CLAP to do so. We do not record synchronizations in our current version of CLAP, because it would need extra synchronization operations, which could limit our ability to capture non-sequential bugs. Also, it could still be costly for a certain range of programs containing intensive high-level races on synchronizations.

7. Related Work

Lee *et al.* [23, 24] pioneered the use of offline symbolic analysis for deterministic replay on SC and TSO models at the hardware level. While being much inspired by their work, their solution does not meet our goal of designing a lightweight and software-only solution. Without the hardware support, their reliance on collecting load values for the SMT solvers to search for shared memory dependencies will create a significant space overhead and a considerable program slowdown. In contrast, we do not collect any values but only the paths taken by the threads, which has much lower recording overhead. Taking a leap from their symbolic analysis, we perform symbolic execution along the program paths to replace "value matching" with "path constraint satisfaction". This not only allows us to exploring more and possibly simpler schedules (*i.e.*, with minimal number of thread context switches), but more importantly, it enables us to encode the preemption bound into the constraint model, converting the NP-complete problem [11] to a polynomial search.

ODR [1] presents a high-level framework that is similar with CLAP: using constraint solving to figure out a schedule that satisfies the recorded information. However, ODR did not provide a concrete constraint solving algorithm and implementation that considers real-world memory consistency and synchronization. Its constraint solving approximates many issues that CLAP provides concrete and accurate solution to.

Wang *et al.* [33, 34] develop a verification framework for predictive concurrency trace analysis. Our modeling of the read-write constraints is similar to their symbolic model, but has several specializations treatments tailored to bug reproduction. For instance, we do not use any synchronization nor log any program state online, and we do not require a sequential consistent execution which is needed in their work to obtain a global trace.

ESD [40] performs symbolic execution to synthesize program failures without any runtime logging. A key difference between CLAP and ESD is that we explore only a single path (*i.e.*, the original path) that is guaranteed to exhibit the same failure, while ESD essentially has to search all program paths and thread interleavings to find the bug, which faces particular challenges in addressing programs with loops and recursive calls. To improve scalability, ESD relies on heuristics to synthesize races and deadlocks (which is unsound and could miss real bugs), while CLAP is built upon a sound modeling of the execution constraints.

Weeratunge *et al.* [36] present a technique that reproduces concurrency bugs by analyzing the failure core dump. Powered by execution indexing [37], their technique actively searches for a failure inducing schedule by comparing the core dump differences between the failing run and the passing run. Compared to [36], CLAP does not require the core dump information and hence is able to reproduce a wider range of concurrency errors.

CHESS [28] detects and reproduces concurrency bugs through dynamic exploration of thread interleaving. To mitigate the schedule explosion, CHESS employs a context-bounded algorithm [27] to explore schedules up to a certain small number of context switches. Our mechanism for encoding the number of thread interleavings shares the same spirit as CHESS. Differently, our technique is parallelizable because of the static nature, while CHESS is difficult to parallelize due to the dynamic exploration strategy.

PRES [29] proposes probabilistic replay that intelligently explores the thread interleavings through a feedback-based replayer. By continuously rectifying the schedule in the previous failing replay, the technique successfully trades multiple replay attempts for efficient online recording.

The idea of doing lightweight recording at runtime and making up for it with a solver has also been successfully applied by Cheung *et al.* [6] for replaying long-running single-threaded programs. The technique achieves low recording overhead by logging only the branch choices at runtime and using symbolic analysis to reconstruct the inputs and memory states.

8. Conclusion

Reproducing concurrency bugs is notoriously challenging due to non-determinism. We have presented a new technique, CLAP, that achieves significant advances over previous approaches. CLAP

does not log any runtime shared memory dependency or program state, works for sequential consistent as well as a range of relaxed memory models, and produces the bug-reproducing schedule in parallel with minimal thread context switches. With all these properties, we believe CLAP is promising for production settings.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback which has substantially improved the content and presentation of this paper. We are also grateful to Thomas Ball for his invaluable comments on CLAP. The work of the first two authors is supported by Hong Kong RGC GRF grants 622208 and 622909.

References

[1] Gautam Altekar and Ion Stoica. ODR: output deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] The SPARC Architecture Manual V9. *SPARC International, Inc*. 1994.

[3] Emina Torlakand, Mandana Vaziri, and Julian Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, 2010.

[4] Thomas Ball and James Larus. Efficient path profiling. In *MICRO*, 1996.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[6] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *ESEC/FSE*, 2011.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[8] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, 2006.

[9] Cormac Flanagan and Stephen Freund. Adversarial memory for detecting destructive races. In *PLDI*, 2010.

[10] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[11] Phillip Gibbons and Ephraim Korach. Testing shared memories. 1997.

[12] Derek Hower and Mark Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.

[13] Jeff Huang, and Charles Zhang. LEAN: Simplifying concurrency bug reproduction via Replay-supported Execution Reduction. In *OOPSLA*, 2012.

[14] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.

[15] Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *SAS*, 2011.

[16] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *FSE*, 2010.

[17] Daniel Jiménez. Fast path-based neural branch prediction. In *MICRO*, 2003.

[18] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[19] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[20] James R. Larus. Whole program paths. In *PLDI*, 1999.

[21] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: hybrid program analysis for determinism. In *PLDI*, 2012.

[22] Michael D. Bond and Milind Kulkarni. Respec: efficient online multiprocessor replay via speculation and external determinism. *Technical report, Ohio State University*, 2012.

[23] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. Offline symbolic analysis to infer total store order. In *HPCA*, 2011.

[24] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. Offline symbolic analysis for multi-processor execution replay. In *MICRO*, 2009.

[25] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *ESEC-FSE*, 2007.

[26] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently. In *ISCA*, 2008.

[27] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[28] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[29] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multi-processors. In *SOSP*, 2009.

[30] Polyvios Pratikakis, Jeffrey Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[31] Kapil Vaswani, Matthew J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *CGO*, 2005.

[32] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *ASPLOS*, 2011.

[33] Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

[34] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace based symbolic analysis for atomicity violations. In *TACAS*, 2010.

[35] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jaganathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[36] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, 2010.

[37] Bin Xin, William Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *PLDI*, 2008.

[38] Min Xu, Rastislav Bodik, and Mark Hill. A "flight data recorder" for full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[39] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[40] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, 2010.

[41] Jinguo Zhou, Xiao Xiao, and Charles Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *ICSE*, 2012.

[42] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS*, 2010.

[43] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. In *SPE*, 2004.