LEAN: Simplifying Concurrency Bug Reproduction via Replay-supported Execution Reduction

Jeff Huang Charles Zhang

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{smhuang, charlesz}@cse.ust.hk

Abstract

Debugging concurrent programs is known to be difficult due to scheduling non-determinism. The technique of multiprocessor deterministic replay substantially assists debugging by making the program execution reproducible. However, facing the huge replay traces and long replay time, the debugging task remains stunningly challenging for long running executions.

We present a new technique, LEAN, on top of replay, that significantly reduces the complexity of the replay trace and the length of the replay time without losing the determinism in reproducing concurrency bugs. The cornerstone of our work is a redundancy criterion that characterizes the redundant computation in a buggy trace. Based on the redundancy criterion, we have developed two novel techniques to automatically identify and remove redundant threads and instructions in the bug reproduction execution.

Our evaluation results with several real world concurrency bugs in large complex server programs demonstrate that LEAN is able to reduce the size, the number of threads, and the number of thread context switches of the replay trace by orders of magnitude, and accordingly greatly shorten the replay time.

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

Keywords Concurrecy Defect, Execution Reduction, Replay

OOPSLA'12, October 19–26, 2012, Tuscon, Arizona, USA.

1. Introduction

1.1 Motivation

Bug reproduction is of critical importance for software debugging. Yet, reproducing concurrency bugs is known to be difficult due to scheduling non-determinism. The technique of multiprocessor deterministic replay (MDR) is able to fully reproduce previous executions, attracting a significant research attention for debugging concurrent programs in the multicore era [1, 5, 8, 9, 13, 17, 18, 22, 30]. As MDR aims at faithfully reenacting an earlier execution, the core focus throughout the decades of research has been on reducing the runtime recording overhead while preserving the determinism. Several recent work [8, 9, 17, 30] demonstrates that the future of low overhead MDR is positive, via special hardware design [8, 17] or even clever software-level approaches [9, 30].

However, we argue that MDR alone is not often sufficient for debugging. Even with a zero-recording-overhead MDR support, the debugging task can remain stunningly challenging for concurrent programs. We identify two main reasons. First, most real world concurrent applications are large and complex. For any non-trivial real execution, the execution trace could be huge and complicated, containing millions (or even billions) of critical events [27] and hundreds of thousands of thread context switches [10, 11]. Facing the ocean of shared memory dependencies and thread interleavings, it is very hard for programmers to locate the bug by inspecting the huge amount of trace information, never to mention the space needs for storing the trace. Moreover, the performance of replay is often slow and hard to predict. As replay typically requires enforcing the scheduling behavior, it is often significantly slower (5x-39000x [1, 22]) than the native execution. For long running executions, the replaying phase may never end within a bounded time budget. It is very frustrating for programmers to wait but without knowing when the bug will be reproduced.

To make MDR more practical for supporting concurrent program debugging, we advocate *the simplification of the replay execution* and *the speeding-up of the replaying process*,

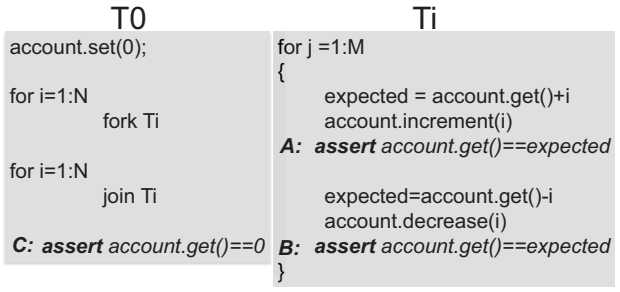

Figure 1. A typical test case for stressing testing an account function. A significant amount of computation in a buggy execution of this program may be redundant.

so that programmers can locate and understand concurrency bugs more effectively using a simplified reproducible buggy execution. To achieve this goal, we propose LEAN, a concurrency bug reproduction technique on top of MDR, that significantly reduces the complexity (size, threads, and context switches) of the replay trace and shortens the replay time without losing the determinism.

1.2 Key Observation

Our key observation is that most computations in the buggy execution are often redundant for reproducing the concurrency bug. As shown by Vaziri, Tip and Dolby [29], most concurrency bugs are exhibited by *only two threads* and *one or two shared variables*. The rest of the threads and shared variable accesses, *if not pre-requisite to understand the bug*, are redundant and can be removed from the execution. This observation is also empirically confirmed by a comprehensive study by Lu et al. [15] on real world concurrency bugs showing that the manifestation of more than 96% of the examined concurrency bugs involves no more than two threads, 66% of the non-deadlock concurrency bugs involve only one variable, and 97% of the deadlock concurrency bugs involve at most two resources. This observation also reflects the common wisdom demonstrated by years of industrial experience (IBM ConTest [6], Stress testing [19] and Microsoft Chess [20]) that most concurrency bugs in practice are triggered by a few threads and a small number of context switches. For example, the stress testing for exposing concurrency bugs typically forks as many threads as possible to repeatedly execute the same code. However, with correct interleaving, a few threads and repetitions are often sufficient to trigger the bug.

To further elucidate this observation, consider a simple, but common, test case for stress testing an account function in Figure 1. The parent thread T_0 forks a number of (N) children threads T_i ($i = 1, 2, \ldots, N$), each of which repeatedly validates two account functions a number of (M) times: increasing and decreasing the account by a certain amount (i). There are three assertions (A, B, C) in the program for the checking of correctness. When an assertion is violated, in the worst case, the buggy execution trace contains M threads (excluding T_0) and $M \times N$ iterations of increasing/decreasing operations on the account. However, in the best case, only two threads and two iterations are able to reproduce the bug. For instance, the account increasing function maybe non-atomic, and an erroneous interleaving happened between the 5th and 10th iterations of threads $T_{(2,3)}$, causing assertion A to be violated. To reproduce the error, the 5th and 10th iterations of threads $T_{(2,3)}$ (plus the erroneous interleaving) are sufficient. The rest of the computation is redundant and can be eliminated from the execution without affecting the bug reproduction.

1.3 Contributions

We propose a redundancy criterion to characterize the redundant computation in a buggy trace. The criterion ensures that, after removing a redundant computation, the resultant execution is able to reproduce the same concurrency bug, which preserves the debugging information but easier to reason about as the amount of computation is greatly reduced. Based on the criterion, LEAN simplifies the buggy execution by iteratively identifying and removing the redundant computation from the original execution trace (skipping the computation by controlling the execution) and, at the same time, enforcing the same schedule between threads in the reduced execution as that in the original buggy execution. The final result produced by LEAN is a simplified execution with all the redundant computation removed.

The key challenge we address is how to effectively identify the redundant computation. We further categorize the redundant computation into two dimensions: the *whole-thread redundancy* and the *partial-thread redundancy*. The whole-thread redundancy characterizes redundant threads of which the entire computation is redundant. For example, all the other threads except $T_{(0,2,3)}$ in our illustrating example are redundant threads and all their computation can be removed. The partial-thread redundancy characterizes the more fine-grained redundant instructions as part of each individual thread. For example, all the other iterations (except the 5th and 10th) of threads $T_{(2,3)}$ in our illustrating example belong to the partial-thread redundancy.

We develop two effective techniques based on delta-debugging [38] to identify the whole-thread redundancy and the partial-thread redundancy, respectively. To reduce the search space of delta-debugging, we utilize the parent-children relationship between threads to iteratively identify the whole-thread redundancy using the dynamic thread hierarchy graph. For the partial-thread redundancy, as it is ineffective to enumerate every combination of the instructions for each thread, we combine an adapted multithreaded program slicing technique [26] and a repetition analysis to remove irrelevant instructions and to identify the redundant iterations of computation. To further improve the effectiveness, we also provide an easy-to-use repetition analysis framework that allows the programmers to annotate repeti-

tive code segments of which some execution iterations are potentially redundant. All those redundant iterations are then automatically validated and filtered out by our technique.

Note that the redundancy criterion is black-box in nature. It does not rely on any data or control dependency information of the program, and is completely based on the bug reproduction property. This allows us to explore more simplification opportunities than those white-box approaches such as program slicing [7, 12, 21] that rely on the program/system dependence graphs for removing the statements that are irrelevant to the fault.

We have implemented LEAN on top of our replay system LEAP [9] for Java programs. Our evaluation results on a set of real concurrency bugs in popular multithreaded benchmarks as well as several large complex concurrent systems demonstrate that LEAN is able to significantly reduce the complexity of the reproducible buggy execution and shorten the replay time without losing the determinism. LEAN produces a simplified execution typically within 20 iterations that deterministically reproduces the concurrency bug. LEAN is able to reduce the size of the replay trace by as large as 324x, the number of threads and thread context switches by 99.3% and 99.6%, and shorten the replay time by more than 300x. We believe LEAN is able to significantly improve the debugging effectiveness of MDR for reproducing concurrency bugs and to reduce the debugging effort for concurrent programs.

We highlight our contributions as follows:

1. We present a trace redundancy criterion and characterize two dimensions of redundancy for concurrency bug reproduction with the MDR support.

2. We present a technique and the design of a tool prototype to automatically simplify the buggy trace by removing the two dimensions of trace redundancy.

3. We evaluate our technique on a set of real concurrency bugs in several large complex Java programs and the results demonstrate the effectiveness of our technique.

The remainder of this paper is organized as follows: Section 2 presents the fundamentals of our trace redundancy criterion; Section 3 presents our technique; Section 4 presents our implementation and Section 5 presents a case study of simplifying the reproduction of a real concurrency bug; Section 6 reports our experimental results; Section 7 discusses related work and Section 8 concludes this paper.

2. Fundamentals

The cornerstone of this work is a redundancy criterion for characterizing the redundant computation in a buggy multithreaded execution w.r.t. to a concurrency bug. We first introduce the common program modeling used by DMR techniques to support concurrency bug reproduction and then present our redundancy criterion based on the model.

2.1 Concurrent Program Execution Modeling

Debugging is often based on a reproducible buggy execution trace. A *trace* captures a concurrent program execution as a sequence of events $\delta = \langle e_i \rangle$ that contains all the critical computation (i.e., thread synchronizations and the read/write accesses to the shared variables), in their execution order. During execution, these events are monitored by MDR techniques to support the deterministic replay. More formally, an event e can be of the following forms [24]:

- FORK(σ,t_p,t_c) denotes a thread t_p starts a thread t_c while executing the statement σ;
- JOIN(σ,t_p,t_c) denotes a thread t_p waits for the termination of a thread t_c while executing the statement σ;
- READ(σ,v,t) denotes that thread t reads a shared variable v while executing the statement σ;
- WRITE(σ,v,t) denotes that thread t writes to a shared variable v while executing the statement σ;
- ACQ(σ,l,t) denotes the acquisition of a lock l by thread t while executing the statement σ;
- REL(σ,l,t) denotes the releasing of a lock l by thread t while executing the statement σ;
- SND(σ,g,t) denotes the sending of a message with unique ID g by thread t while executing the statement σ;
- RCV(σ,g,t) denotes the reception of a message with unique ID g by thread t while executing statement σ.

As threads execute concurrently, events by different threads may interleave. A *preemptive interleaving* occurs if two successive events from the same thread in the trace are interleaved by events from other threads, but they could have been executed continuously without the interleaving. Preemptive interleaving is non-deterministic, because it depends on the behavior of the thread scheduler and the timing variations between threads [22]. And because of the non-determinism, preemptive interleavings are the root cause of many concurrency bugs.

The *schedule* of an execution is the projection of the trace on the thread ID, which captures all the preemptive interleavings in the trace. If a schedule contains no preemptive interleaving, we say it is *sequential* and, otherwise, *non-sequential*. Since the schedule contains the execution order information of all the critical computation across threads, it can be used to deterministically reproduce the program execution [2].

Starting with an initial state Σ^0 and, following a schedule ξ, the program can reach a final state Σ^f. We say ξ exhibits a bug if Σ^f satisfies a predicate, say ϕ, that denotes the bug. The bug predicate is defined as follows:

DEFINITION 1. *(**Bug predicate**) A bug predicate, ϕ, characterizes the exhibition of a bug in the program execution over the final program state. The bug is exhibited in the execution iff $\phi(\Sigma^f)$ is evaluated to be true.*

Following different schedules, however, Σ^f may be different and may or may not satisfy ϕ. We call the bug a *sequential bug* if any sequential schedule is able to exhibit it, and a *concurrency bug* if only a non-sequential schedule can exhibit it. To reproduce a concurrency bug, essentially, a MDR technique captures or computes the schedule ξ in the buggy run and then enforces the same ξ in all replay runs to exhibit the bug.

2.2 A Model of Trace Redundancy

From a high level view, LEAN simplifies the concurrency bug reproduction by controlling the program execution to skip instructions in the program that are redundant to reproducing the bug. Generally speaking, an instruction (or a group of instructions) as a part of the program execution cannot be arbitrarily skipped, as it may result in two possible negative consequences: the program malfunctions, or the bug disappears. The program might malfunction if the skipped instruction is an indispensable part of the program logic, while the bug might disappear if the skipped instruction is involved in the buggy interleavings that are related to the bug. Either consequence will make the reduced execution not useful for debugging.

We propose a redundancy criterion for the concurrency bug reproduction that ensures none of the two consequences will happen if a redundant instruction is skipped. The basic idea is that, after removing the redundancy, the reduced execution is still sufficient for understanding the bug, i.e., the same bug is reproduced. A subtle problem in defining the criterion is that, in practice, we may not have such a bug predicate ϕ as that defined in Definition 1. In practice, we often use assertions or rely on runtime exceptions to determine whether a bug is exhibited or not. However, the assertions or exceptions may be insufficient to distinguish between the behavior of the bug manifestation and the behavior of program malfunction, in which case the program is no longer working properly as expected due to the removal of a necessary instruction. For example, the assertion that characterizes the bug in the original execution may always be violated after removing a certain instruction. Although the reduced execution manifests the violation of the assertion, it is not useful for debugging because the assertion is not able to characterize the same bug as that in the original execution.

We tackle this issue from the perspective of thread interleavings. For a concurrency bug, essentially, it is some nondeterministic buggy interleavings that cause the bug (assuming the input is deterministic). For debugging, programmers want to understand how the bug occurs with these buggy interleavings. If the program executes sequentially and behaves correctly, the bug should not manifest. On the other hand, if the program malfunctions after removing an instruction, either the program cannot proceed to execute the buggy statement or the bug predicate ϕ is always satisfied regardless of the buggy interleavings. Therefore, we define the redundancy criterion as follows:

DEFINITION 2. *(***Trace redundancy criterion***) Consider a trace δ that exhibits a concurrency bug (δ drives the program to a state satisfying the bug predicate ϕ) and a subset E of the events in δ. Let $\delta \backslash E$ denote the trace δ with the events in E removed. We define E is redundant if after removing the events in E, the following two conditions are satisfied:*

I. $\delta \backslash E$ can still drive the program to a state that satisfies ϕ;

II. any sequential schedule of the reduced execution does not satisfy ϕ.

We assume ϕ characterizes a concurrency bug. The soundness of this criterion is easy to follow. First, Condition I and Condition II together ensure that the reproduced bug is a concurrency bug, because ϕ is satisfied under the original buggy schedule (excluding the events in E), but not a sequential schedule. Second, consider condition II, since ϕ is evaluated but not satisfied (i.e., the bug does not manifest) under a sequential schedule[1], the program does not malfunction after removing the events in E. Otherwise, either ϕ is not evaluated or ϕ is always satisfied. Hence, the same concurrency bug is reproduced under Conditions I and II.

It is worth noting that the trace redundancy is not defined over a single event but a subset of events in the trace, which correspond to a group of instructions in the program execution. The reason is that redundant instructions are not independent but may be closely related to each other. A group of instructions may be redundant but any single instruction of them may not. For example, suppose an erroneous interleaving between the 5th and 10th iterations of threads $T_{(2,3)}$ manifests the bug in Figure 1. The whole computation of thread T_1 is redundant, but any single instruction of T_1 alone is not. Without any dependence information between the instructions, removing the trace redundancy is essentially a combinatorial optimization problem, which is exponential to the number of instructions in the original buggy execution.

To facilitate more effective simplification, we further characterize the redundancy into two dimensions:

- *whole-thread redundancy* - all computation of a certain thread is redundant;
- *partial-thread redundancy* - redundant instructions as part of each individual thread.

This categorization utilizes the thread identity relationship between the computations. In practice, threads as separate control flows in the program are more likely to be independent to each other than the individual instruction. We can skip all the computation of the redundant thread. Compared to the whole-thread redundancy, the partial-thread re-

[1] Note that we do not need to check all sequential schedules but checking any one of them is sufficient to validate whether the concurrency bug is still a concurrency bug or not.

dundancy examines the more fine-grained instructions local to each individual thread. If an instruction by a certain thread is redundant, we can skip it during the execution of that thread. In our illustrating example, all the other threads except $T_{(0,2,3)}$ are redundant, which belong to the whole-thread redundancy, and most of the repetitions of threads $T_{(2,3)}$ are redundant, which belong to the partial-thread redundancy.

3. Automatic Redundance Removing

We propose two techniques to remove the trace redundancy for simplifying the concurrency bug reproduction. The first technique effectively validates and removes the whole-thread redundancy by adapting delta-debugging [38] using the thread hierarchy information. Sharing the essence of delta-debugging, our technique produces a 1-minimal set of threads [38] that are not redundant in the buggy execution. The second technique targets at effectively removing the partial-thread redundancy related to irrelevant instructions and repetitions. It combines a dynamic multithreaded slicing technique and a static repetition analysis to improve the simplification efficiency, as well as a simple annotation framework that integrates programmers' hints to further reduce the search space. The entire simplification process is deterministic. There is no interleaving non-determinism during simplification as we control all the thread scheduling (including the non-preemptive ones) during the replay.

3.1 Removing Whole-Thread Redundancy

Our general idea to identify and to remove the whole-thread redundancy follows the approach of hierarchical delta-debugging [16, 38]. We use a bisection method to pick candidate threads and test whether they can be removed from the execution or not. More specifically, we control the program to disable the selected candidate threads and validate the reduced execution for the two conditions defined in our redundancy criterion in Section 2.2. Our technique for removing the whole-thread redundancy is fully automatic. It does not require any user intervention.

There are two main challenges we address in our approach. First, threads may not be arbitrarily removed. For example, if a parent thread is removed, all its descendants are disabled. Second, after removing a redundant thread, we must compute the schedule of the remaining threads (in order to deterministically replay the reduced execution). Our approach contains two core treatments to address these two problems. First, we extract a dynamic thread hierarchy graph of the original buggy execution (TH-Tree) and perform the delta-debugging based on the TH-Tree, to make sure that if a parent thread is disabled, all its descendent threads are disabled. Figure 2 shows an example of the TH-Tree. For example, if T_1 and T_3 are selected, all their descendants (shown in the gray boxes in Figure 2) are also selected. Second, we compute the schedule for the remaining threads by projecting the trace on the thread ID without the IDs of the selected candidate threads and their descendants. The schedule is enforced in the validation run to test whether the bug can still be reproduced or not. In this way, the thread schedule of the remaining execution is the same as that in the original buggy execution, which preserves the erroneous interleavings.

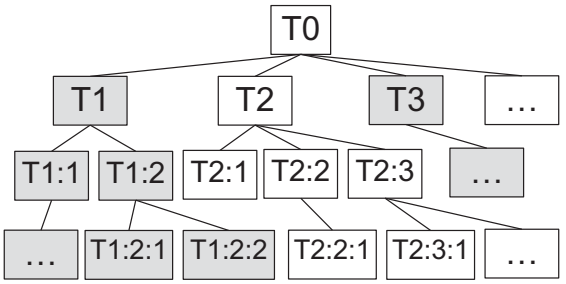

Figure 2. An example of dynamic thead hierarchy graph (TH-Tree). When $T_{1,3}$ are selected, all $T_{1,3}$ and their descendents (gray color) are disabled.

Algorithm 1 summarizes our algorithm. Given the original buggy trace, it produces a simplified trace (execution) containing only the 1-minimal set of threads that is able to reproduce the bug. The 1-minimal property means that, all remaining threads are necessary such that removing any one of them would cause the reduced execution to fail to reproduce the bug. Our algorithm starts by iterating on the height of the TH-Tree. In each iteration, we always pick the candidate threads with the same height. Starting from the threads with height 1 (the main thread is of height 0), we first select the candidate threads ($thread\_sets$) to be validated for the redundancy. If a parent thread is selected, its descendants are all disabled. We then process the selected threads using a delta-debugging algorithm, as shown in Figure 3. Each invocation of delta-debugging computes the 1-minimal set of threads (in the input threads denoted by c_x) that are necessary to reproduce the bug. The set c_x in the $ddmin$ algorithm corresponds to the selected threads. The $validate$ procedure (Algorithm 2) corresponds to the test function in delta-debugging. It tests whether the two conditions in the redundancy criterion are both satisfied or not after disabling the selected threads: (1) the bug is reproduced with the computed schedule of the remaining threads; (2) the bug is not reproduced with a sequential schedule. If both conditions are true, it means that the selected threads are redundant and they are removed from the execution. This process is repeated for all levels of threads in the TH-Tree, until no new thread can be removed.

Unique thread identification In our algorithm, an additional problem we need to address is how to consistently identify threads across runs, as the validation run requires matching the selected threads. We take a similar approach as that in jRapture [25] to identify threads and their children. The key observation is that each thread should create

Let ***validate*** and c_x be given such that ***validate***(c_x)=**X***(fail)*. The algorithm computes c'_x =***ddmin***(c_x)=***ddmin***$_2(c'_x, 2)$ such that $c'_x \subseteq c_x$, ***validate***(c'_x)=**X,** and c'_x is 1-minimal.

$$ddmin_2(c'_x, n) = \begin{cases} ddmin_2(\Delta_i, n) & if \exists i \in \{1, \dots, n\}.\ \boldsymbol{validate}(\Delta_i) = \mathbf{X} \\ ddmin_2(\nabla_i, max(n-1, 2)) & else\ if \exists i \in \{1, \dots, n\}.\ \boldsymbol{validate}(\nabla_i) = \mathbf{X} \\ ddmin_2(c'_x, min(|c'_x|, 2n)) & else\ if\ n < |c'_x| \\ c'_x & otherwise. \end{cases}$$

where $\nabla_i = c'_x - \Delta_i$, $c'_x = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\Delta_i \approx |c'_x|/n$.

Figure 3. The delta-debugging algorithm. The function $validate$ return true if the two conditions in the redundance criterion are both satisfied. For conciseness, the input trace is ignored in the $ddmin$ algorithm.

Algorithm 1 RemoveWholeThreadRedundancy(δ)

Input: δ - the original trace $\langle e_i \rangle$
Output: δ' - the simplified trace with all redundant threads removed
$TH_Tree \leftarrow$ExtractThreadHierarchyGraph(δ)
$height \leftarrow the\_height\_of(TH\_Tree)$
for $level \leftarrow 1 : height$ **do**
  $thread\_set \leftarrow get\_threads(TH\_Tree, level)$
  $minimal_threads \leftarrow$DeltaDebugging($\delta$,$thread\_set$)
  $redundant\_threads \leftarrow (thread\_set \setminus minimal\_threads)$ and their descendents
  remove $redundant\_threads$ from TH_Tree
  remove all events by $redundant\_threads$ in δ
end for
return δ

Algorithm 2 Validate(δ,$disabled\_threads$)

Input: δ - a trace $\langle e_i \rangle$
Input: $disabled\_threads$ - a set of disabled threads
$\delta' \leftarrow$ remove all events by $disabled\_threads$ in δ
$\xi \leftarrow get\_schedule(\delta')$
$\xi_{seq} \leftarrow get\_sequential_schedule(\delta')$
if IsBugReproduced(δ',ξ) **then**
  if IsBugNotReproduced(δ',ξ_{seq}) **then**
    return true
  end if
end if
return false

its children threads in the same order, though there may not exist a consistent global order among all threads. We therefore create a consistent identification for all threads based on the parent-children order relationship. More specifically, starting from the main thread (T_0), each thread maintains a thread-local counter for recording the number of children it has forked so far. And everytime a new thread is forked, it is identified with its parent thread ID associated with the counter value. For instance, suppose a thread t_i forks its jth child thread, this child thread will be identified as $t_{i:j}$. The thread identifier in Figure 2 illustrates this identification strategy. Note that the identification is performed only once on the initial TH-Tree and remains consistent for all the simplification runs.

3.2 Removing Partial-Thread Redundancy

To identify the partial-thread redundancy, we may directly apply delta-debugging on the granularity of the individual instructions. However, this naive approach is ineffective because enumerating and validating every combination of the instructions for each individual thread could be very expensive. To improve the efficiency, our technique combines the multithreaded dynamic slicing with a repetition analysis to identify the redundant computation local to each individual thread. The dynamic slicing tracks the data and control dependencies between instructions in the execution trace and removes those instructions that are irrelevant to the bug. The repetition analysis is a heuristic that targets at removing the redundancy related to repetitions. To further improve the effectiveness of repetition analysis, LEAN also provides a simple framework that allows programmers annotating the repetitive code segments, which significantly reduces the search space of trace simplification. We next describe them in detail.

3.2.1 Multithreaded dynamic slicing

The dynamic dependence graph (DDG) is the classical model for slicing single-threaded executions, which captures the dynamically exercised Read-After-Write (RAW) and control dependencies. Each node in the DDG represents an execution instance of a statement (an instruction) while the edges represent the dependences. For multithreaded execution, Tallam et al. [26] proposes a dynamic slicing modeling for data race detection. Their model extends the DDG to consider the additional data dependencies on shared variable accesses.

Our slicing model for concurrency bug reproduction is similar to but more strict than the model by Tallam et al. [26]. To guarantee the deterministic bug reproduction, in addition to the shared variable read/write dependencies, we also need to consider the dependencies on synchronization

operations. Specifically, given the buggy execution, we construct a multithreaded dependence graph (MDG) that consists of the DDG for each individual threads as well as the following dependence relations between the instructions a_i and a_j by different threads:

- *Synchronization dependencies*
  - REL→ACQ: a_i is the REL operation that releases the lock acquired by the ACQ operation a_j;
  - FORK→START: a_i is the FORK operation that forks the thread whose START operation (a dummy instruction introduced as the first operation of a thread) is a_j;
  - EXIT→JOIN: a_i is the EXIT operation of a thread (a dummy instruction introduced as the last operation of a thread) that the JOIN operation a_j joins;
  - SND→RCV: a_i is the SND operation that sends the message received by the RCV operation a_j;
- *Shared variable dependencies* - a_i and a_j are consecutive on the same shared variable:
  - WRITE→READ: a_i is a WRITE and a_j is a READ;
  - READ→WRITE: a_i is a READ and a_j is a WRITE;
  - WRITE→WRITE: both a_i and a_j are WRITE accesses.

Note that the WRITE→WRITE dependency must be included in the MDG, to ensure the correctness of MDR [10]. Otherwise, a read in the replaying phase may return the value written by a different write from that in the original buggy execution, which may cause the failure of MDR.

Algorithm 3 shows our dynamic slicing algorithm for removing the partial-thread redundancy. We first construct the MDG that includes both the DDG for each thread in the execution and the synchronization and shared variable dependencies. Starting from the buggy instruction which violates the bug predicate, we perform a backward analysis that keeps only the instructions with a direct or a transitive dependency relation to the buggy instruction. All the other instructions are marked to be irrelevant to reproducing the bug and are skipped in the simplified execution.

Algorithm 3 DynamicMultithreadedSlicing(δ,α_f)

1: **Input**: δ - the full execution trace after removing all redundant threads
2: **Input**: α_f - the buggy instruction
3: **Output**: δ' - the simplified trace
4: $mdg \leftarrow$ ConstructMultithreadedDependencyGraph(δ)
5: $mdg' \leftarrow$ ReverseEdge(mdg)
6: $relevant\_instructions \leftarrow$ DepthFirstSearch(α_f) on mdg'
7: $\delta' \leftarrow$ remove the instructions from δ that are not in $relevant\_instructions$
8: return δ'

3.2.2 Repetition analysis

Redundancy is often caused by repetitions. Specifically, we observe that a large portion of redundant computation by each individual thread is rooted by the repetitive code blocks (RCBs) that contain repeated operations in loops. The operations inside a RCB are expected to execute a few iterations upon the loop condition with no break operation. The loop variable is often a primitive data (e.g., integers) that used as a counter for counting the number of iterations so far. We propose a static repetition analysis to identify RCBs in the program. The RCBs are used as a pool of potential redundant computation that we may simplify. Each execution iteration of a RCB is considered as potentially redundant. After validating the redundancy of an iteration using our redundancy criterion, we can remove all computation of this iteration from the execution.

Our repetition analysis is based on a simple intra-procedural loop analysis. For each loop, we consider two conditions to mark it as a potential RCB. First, the loop condition contains only primitive or concrete data and the loop variable is only incremented or decremented once in each iteration. Second, there is no break operation inside the loop (exceptions are allowed). Despite the simplicity, our experiments show that this analysis is effective and efficient for identifying redundant computation caused by RCBs.

Algorithm 4 RemoveRepetitionRedundancy(p,δ)

1: **Input**: p - the program
2: **Input**: δ - the trace after slicing
3: **Output**: δ' - the final simplified trace
4: $statements \leftarrow$ GetRepetitiveCodeBlocks(p)
5: $threads \leftarrow get\_threads(\delta)$
6: **for** t in $threads$ **do**
7: **for** σ in $statements$ **do**
8: $all\_iterations \leftarrow get\_iterations(\delta,t,\sigma)$
9: $minimal_iterations \leftarrow$DeltaDebugging($\delta$,$all\_iterations$)
10: remove $(all_iterations \backslash minimal_iterations)$in δ
11: **end for**
12: **end for**
13: return δ

Algorithm 4 shows our algorithm for removing the partial-thread redundancy caused by repetitions. This algorithm is applied after slicing the buggy trace. We first identify the RCB that contains potential redundant computation. We then perform delta-debugging on each iteration of the RCB for each thread, to validate the redundancy of the computation corresponding to the iteration.

A framework for repetition analysis LEAN also provides an option for the programmers to annotate RCBs, which can help significantly improve the effectiveness of our automatic repetition analysis. Our general observation is that programmers often have the knowledge of whether some

Ti

```
for j =1:M
{
@rcb-begin
      expected = account.get()+i
      account.increment(i)
 A: assert account.get()==expected

      expected=account.get()-i
      account.decrease(i)
 B: assert account.get()==expected
@rcb-end
}
```

Figure 4. Some iterations of the code block demarcated by `@rcb-begin` and `@rcb-end` are specified as potentially redundant.

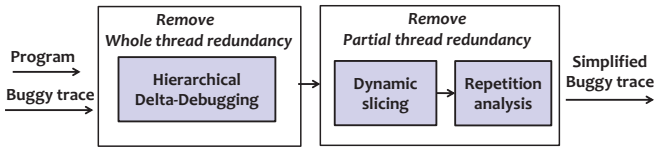

Figure 5. An overview of LEAN

code blocks is repetitive or not (in particular, in writing the test drivers). This piece of information is in fact easy for the programmers to specify (e.g., using simple annotations), but very difficult to identify by any automatic approach because of the absence of the repetition criterion. More importantly, without any further intervention, we can help programmers automatically validate whether some executions of the RCBs are redundant or not, and eliminate them from the buggy execution if they are redundant.

Our framework is easy to use. Programmers simply mark the beginning and the end of the RCB by `@rcb-begin` and `@rcb-end`, respectively. For example, programmers may mark the RCB for thread T_i in a way as shown in Figure 4. We then perform delta-debugging on each iteration of the code, and filter out most redundant iterations. Also, this framework is flexible. New annotations may be added after each round of simplification, when programmers get more information about the bug from the intermediate simplified execution.

4. Implementation

To evaluate our technique, we have implemented LEAN on top of LEAP [9], our MDR framework for applications written in Java[2]. Figure 5 shows an overview of LEAN. Given the target concurrent program and the buggy execution trace, LEAN first removes the whole-thread redundancy from the trace using our adapted hierarchical delta-debugging algorithm (Algorithm 1). It then further simplifies the resultant execution by removing the partial-thread redundancy using our dynamic multithreaded slicing algorithm (Algorithm 3) and the repetition analysis (Algorithm 4). The final output produced by LEAN is a simplified buggy execution in which all the redundant computation is skipped in the replayed execution.

For the delta-debugging, we faithfully implemented the algorithm described in Figure 3. Our slicing implementation is based on the Indus framework [23], which we adapt for dynamic multithreaded execution traces. In addition to the data dependencies across threads, slicing also takes care of all the data and control dependencies internal to each individual thread in the execution.

For supporting MDR, LEAN collects in the trace the following types of events in a global order: read/write accesses to shared variables, lock acquisition/release, thread fork/join, and wait/notify events. To support recording long running programs which produce large traces, LEAN does not put the entire trace in the main memory but saves it to a database.

To disable an instruction, we instrument the program to insert control statements before the statement (Jimple statement in Soot[3]) which corresponds to the instruction. For example, to disable a thread, we insert control instrumentation before *Thread.start()* and *Thread.join()* to make sure that the disabled thread is not executed and joined by any other thread. We distinguish the dynamic thread by assigning a unique ID to each thread instance (explained in Section 3.1). For the partial-thread redundancy, we also maintain a thread local counter for each annotated RCBs, to denote the iteration instance of each thread in executing the RCB.

To control the thread schedule, we reuse the application-level scheduler of LEAP. The thread IDs of all the events in the trace form a global schedule. After disabling a thread, we simply remove the thread ID from the global schedule. To enforce a sequential schedule, we control the execution of a thread until it terminates or cannot continue execution (i.e., waiting for a lock or joining for the termination of another thread) and then randomly pick an enabled thread to proceed. For removing the partial-thread redundancy, we also associate each event in the trace with its corresponding statement in the program. User annotated RCBs are interpreted as special statement blocks. To generate the remaining schedule after disabling a certain iteration of a RCB, we first remove the corresponding events in the trace according to the RCB and the per-iteration information, and then compute the schedule by performing a projection of the remaining trace on the thread ID.

[2] Our technique is general to concurrent programs written in any programming language.

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

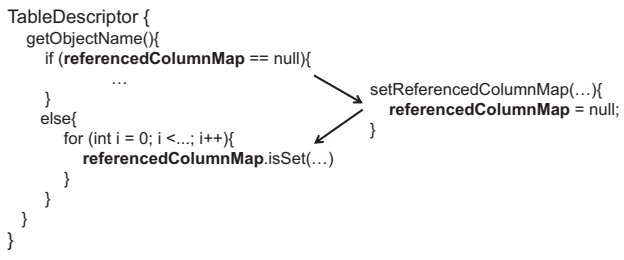

Figure 6. A real concurrency bug #2861 in Derby. The thread interleaving following the solid arrow on the shared data `referencedColumnMap` crashed the program with `NullPointerException`.

5. A Case Study

In this section, we present a case study of a real concurrency bug reproduction in Apache Derby DBMS[4]. We illustrate how LEAN simplifies the bug reproduction w.r.t. the whole-thread redundancy and the partial-thread redundancy in a detailed view.

5.1 Description of Derby Bug #2861

Figure 6 shows the concurrency bug #2861 we study in the Apache bug database[5]. The bug is concerned with a thread safety issue in the `org.apache.derby.iapi.sql.dictionary.TableDescriptor` class. The shared data `referencedColumnMap` is checked for `null` at the top of the `getObjectName` method and later dereferenced if it is not null. Due to an erroneous interleaving, another thread can set `referencedColumnMap` to `null` in the `setObjectName` method and causes the program to crash by throwing a `NullPointerException`. Figure 7 shows a driver program (also documented in the bug database) for triggering the bug. Ignore all the gray areas for the moment; these are statements inserted by LEAN. The driver program starts N threads each creating (lines 41-45) and then dropping (lines 48-51) a separate view against the same source view, repeated M times. Because of the non-determinism, the bug is very difficult to manifest with a small N and M. In our experiment with N=2 and M=2 on an eight-core Linux machine, we did not observe a single run of failure after 1000 runs. With a larger number of threads and repetitions, the probability of triggering the bug is increased. When we set N=10 and M=10, we were able to trigger the bug in three out of 1000 runs.

With the help of a MDR system such as LEAP, we are able to deterministically reproduce the bug once it manifests. The problem is that the bug reproduction run is too complicated, with too many threads (11) and thread context switches (6,439). The size of the execution trace (which contains the critical events only) is as large as 94.1M, and it took LEAP as long as 466 seconds to reproduce the bug. Given such a bug reproduction run, it is still challenging for the programmers to understand the bug by inspecting the trace.

```
1  TestEmbeddedMultiThreading {
2    main(String args[]){
3        int numThreads = Integer.parseInt(args[0]);
4        int numIterations = Integer.parseInt(args[1]);
5
6        //register the embedded driver and create the test database
7        EmbeddedDriver driver = new EmbeddedDriver();
8        conn = DriverManager.getConnection("jdbc:derby:DERBY2861");
9        stmt = conn.createStatement();
10       sql = "CREATE VIEW viewSource AS SELECT col1, col2 FROM
11             schemamain.SOURCETABLE“
12       stmt.execute(sql);
13       stmt.close();
14
15       //create test threads
16       Thread[] threads = new Thread[numThreads];
17       for (i = 0; i < numThreads; i++)
18           threads[i] = new Thread(new ViewCreatorDropper(
19             "schema1.VIEW" + i, "viewSource", "*", numIterations));
20
21       //start test threads
22       for (int i = 0; i < numThreads; i++)
23           threads[i].start();          → if(shouldStartThread(threads[i]))
                                              threads[i].start();
24
25       //wait for threads to terminate
26       for (int i = 0; i < numThreads; i++)
27           threads[i].join();           → if(shouldJoinThread(threads[i]))
                                              threads[i].join();
28   }
29 }
30 ViewCreatorDropper implements Runnable {
31   ViewCreatorDropper(String viewName, String sourceName,
32                      String columns, int iterations) {
33       m_viewName = viewName;
34       m_sourceName = sourceName;
35       m_columns = columns;
36       m_iterations = iterations;
37   }
38   run(…){
39       for (i = 0; i < m_iterations; i++)
40       {   @rcb-begin               → if(shouldExecuteIteration(i))
41                                      {
42           //create view
43           stmt = conn.createStatement();
44           sql = " "CREATE VIEW " + m_viewName + " AS SELECT "
45              + m_columns + " FROM " + m_sourceName“;
46           stmt.execute(sql);
47           stmt.close();
48
49           //drop view
50           stmt = conn.createStatement();
51           sql = " " "DROP VIEW " + m_viewName“;
52           stmt.execute(sql);
53           stmt.close();
54       @rcb-end                     →  }
55       }
56 }
```

Figure 7. A real world test driver for triggering the concurrency bug in Figure 6. The statements inserted by LEAN to simplify the execution are shown in the gray areas.

5.2 How LEAN Simplifies the Bug Reproduction

LEAN simplifies the reproduction of this bug by removing the redundant computation in the reproducible buggy execution. Although there are ten testing threads each of which repeats ten times in the buggy execution, we can observe that, in the best case, two testing threads each with one iteration is sufficient to trigger the bug. All the other eight threads and nine iterations are redundant and can be removed from the bug reproduction run.

[4] http://db.apache.org/derby/

[5] https://issues.apache.org/jira/browse/DERBY-2861

Taking the original buggy execution as the input, LEAN first identifies and removes the redundant threads in the execution using Algorithm 1. Figure 8 illustrates the simplification process. Because the dynamic thread hierarchy graph in the buggy execution contains only one level of threads, the entire simplification process invokes the delta-debugging procedure only once, which directly applies on threads $T_{(1,2,...,10)}$. To skip a thread, LEAN controls the execution of the program by inserting a condition checking before *Thread.start()* and *Thread.join()* (as shown in the gray areas at lines 23 and 27 in Figure 7). A thread is not started or joined if it is removed. After four rounds of simplification, threads $T_{(2,3)}$ remain in the reduced execution and all the other threads are removed. This process took 1,841 seconds in our experiment. After removing the redundant threads, 75.1M(79.8%) of the events in the original buggy trace were removed and the size of the remaining trace was reduced to 19M.

After removing the whole-thread redundancy, LEAN then further processes the reduced buggy execution to remove the partial-thread redundancy. It first performs the dynamic slicing to remove irrelevant instructions using Algorithm 3. As slicing tracks all the dynamic data dependencies across threads as well as all the intra-thread data and control dependencies in the remaining buggy execution, it took LEAN 553 seconds to finish the slicing process in our experiment, and an additional 6.2M(6.6%) of the events were removed from the trace. Similar to the control of threads, we simply insert control statements before the irrelevant instructions to skip their executions.

LEAN then continues to simplify the reduced buggy execution by removing the redundant repetitions using Algorithm 4. Our automatic repetition analysis successfully identified the RCB at lines 42-53 in the test thread, as demarcated by `@rcb-begin` and `@rcb-end` at lines 41 and 54 in Figure 7. To control the execution of a certain iteration i of the RCB, we insert a control statement before the RCB with i as the input parameter (as shown in the gray area at line 40), determining whether the ith iteration is enabled or not. Figure 9 illustrates the simplification process for LEAN to remove the redundant execution iterations of the RCB of threads $T_{(2,3)}$. After ten rounds of simplification, the 7th iteration of T_2 and the 4th iteration of T_3 remain and all the other iterations are removed. This process took around 200 seconds in our experiment. An additional 11.6M (12.3%) of the events were removed and the size of the final buggy trace was reduced to around 2.01M.

In total, it took LEAN 2,593 seconds to simplify the original buggy execution. The final simplified execution was able to reproduce the same bug and was significantly simpler than the original buggy execution. The simplified trace size was reduced by 47x (from 94.1M to 2.01M), containing only 3 threads ($T_{(0,2,3)}$) and 433 thread context switches, and its replay time by LEAN was shortened by 46x (from

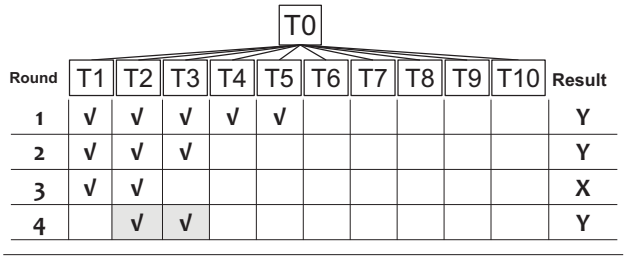

Figure 8. Illustration of delta-debugging for removing the whole thread redundancy. T_i denotes the ith test thread created by the main thread T_0. After four rounds of simplification, threads $T_{(2,3)}$ remain and all the other threads are removed.

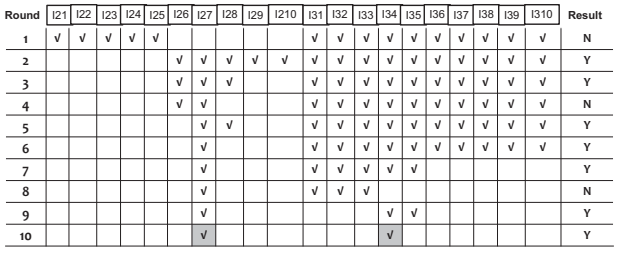

Figure 9. Illustration of delta-debugging for removing the redundant repetitions for the remaining threads $T_{(2,3)}$. I_{ij} denotes the jth iteration of thread T_i where i=2,3 and j=1,2,...,10. After ten rounds of simplification, the 7th iteration of T_2 and the 4th iteration of T_3 remain and all the other iterations are removed.

446 to 10.2 seconds). Moreover, all the instrumentations and the thread scheduler in LEAN are transparent to the programmers, such that the debugging task can be performed on the simplified buggy execution in a normal debugging environment.

6. Experiments

The goal of our technique is to improve the effectiveness of the MDR support for debugging concurrent programs, via removing the redundancy from the reproducible buggy trace. Accordingly, our evaluation aims at answering the following two research questions:

RQ1. Effectiveness - Is LEAN effective in simplifying real buggy traces? How much reduction of the replay time and the trace complexity (i.e., size, threads, and context switches) can our approach achieve?

RQ2. Efficiency - How efficient is LEAN for identifying and removing the trace redundancy?

All experiments were conducted on two eight-core 3.00GHz Intel Xeon machines with 16GB memory and Linux 2.6.22 and JDK1.7.

Program	Original Trace				Simplified Trace			
	Size	#Thread	#CS	Replay Time	Size	#Thread	#CS	Replay Time
BuggyPro	460K	34	1,003	1.27s	13.2K(↓**97.1%**)	4(↓**88.2%**)	28(↓**97.2%**)	39ms(↓**97%**)
Tsp	44.1M	5	9,190	280s	22.1M(↓**49.9%**)	3(↓**40.0%**)	4,588(↓**50.0%**)	115s(↓**58.9%**)
ArrayList	1.72M	451	2,381	6.5s	6.4K(↓**99.6%**)	3(↓**99.3%**)	10(↓**99.6%**)	20ms(↓**99.7%**)
LinkedList	2.20M	451	2,564	7.2s	6.8K(↓**99.7%**)	3(↓**99.3%**)	10(↓**99.6%**)	22ms(↓**99.7%**)
OpenJMS	128.9M	36	7,287	606s	1.82M(↓**98.5%**)	7(↓**80.5%**)	415(↓**94.3%**)	16.3s(↓**97.3%**)
Tomcat	38.2M	13	3,543	206s	1.26M(↓**96.7%**)	4(↓**69.2%**)	111(↓**96.9%**)	3.3s(↓**98.4%**)
Jigsaw	20.1M	11	2,322	154s	416K(↓**98.0%**)	3(↓**72.7%**)	64(↓**97.2%**)	2.4s(↓**98.4%**)
Derby	94.1M	11	6,439	466s	2.01M(↓**97.8%**)	3(↓**72.7%**)	433(↓**92.5%**)	10.2s(↓**97.6%**)

Table 2. Experimental results - RQ1: Effectiveness

Program	SLOC	Input/#Threads#Iterations
BuggyPro	348	race exception/33/-
Tsp	709	map4/4/-
ArrayList	5,979	not-atomic bug/450/-
LinkedList	5,866	not-atomic bug/450/-
OpenJMS-0.7.7	262,842	order violation bug/20/10
Tomcat-5.5	339,405	bug#37458/10/10
Jigsaw-2.2.6	381,348	NPE bug/10/10
Derby-10.3.2.1	665,733	bug#2861/10/10

Table 1. Evaluation benchmarks

6.1 Benchmarks

We quantify our technique using a set of widely used third-party concurrency benchmarks with known bugs. We configure the program inputs to generate buggy traces of different sizes and complexity. To understand the performance of our technique on real applications in practice, we also include several large concurrent server systems in our benchmarks. The first column in Table 1 shows the benchmarks used in our experiments. *BuggyPro* is a multithreaded benchmark from the IBM ConTest benchmark suite [6], *ArrayList* and *LinkedList* are open libraries from Suns JDK 1.4.2, and *Tsp* is a parallel branch and bound algorithm for the travelling salesman problem from ETH [31]; The remaining four benchmarks are real server systems. *OpenJMS-0.7.7* is an enterprise message-oriented middleware server, *Tomcat-5.5* is a widely used JSP and Servlet Container, *Jigsaw-2.2.6* is W3C's leading-edge web server platform, and *Derby-10.3.2.1* is a widely used open source Java RDBMS from Apache. Column 2 (*SLOC*) reports the source lines of code of our evaluation benchmarks. The sizes range from a few hundred lines to over 600K lines of code. Column 3 (*Input/#Threads#Iterations*) reports the input data (the bug, the number of threads, and the iterations, if available) configured in the recorded execution of the benchmark.

6.2 RQ1: Effectiveness

The goal of our first research question is to evaluate how effective our technique is for simplifying the buggy execution traces of real concurrent programs. To generate the data necessary for investigating this question, we proceed as follows. For each benchmark, we first run it multiple times with random thread schedule until the bug manifests and use LEAN to collect the corresponding buggy trace of each run. For each trace, we then apply our technique to produce a simplified trace with the redundancy removed. During the simplification process, we first remove the whole-thread redundancy and then the partial-thread redundancy (consists of both slicing and the repetition analysis). The whole process is fully automatic with no user intervention. We measure the percentage of trace size reduction with respect to the two dimensions of redundancy. We also quantify the final simplification results in terms of the reductions of the trace size, the number of threads and the number of thread context switches, as well as the replay speedups. To demonstrate the simplification effectiveness of our approach, we also compared LEAN with an execution reduction technique ER [27] that uses the dependence graph for the simplification.

Table 2 reports our final simplification results. Columns 2-5 (*Size*, *#Thread*, *#CS*, *Replay Time*) report the size of the original trace, the number of threads, the number of thread context switches (including both non-preemptive and preemptive ones) in the original trace, and the replay time of the original trace, respectively, while Columns 6-9 report the corresponding statistics of the simplified trace. As the table shows, the size of the original trace ranges from 460KB (*BuggyPro*) to more than 128MB (*OpenJMS*) on disk, which take from 1.27 seconds to more than 10 minutes to replay to reproduce the bug. The original trace is also of significant complexity w.r.t. the number of threads and the number of context switches, ranging from 5 threads in *Tsp* to 451 threads in *ArrayList* and *LinkedList*, and from 1,003 context switches in *BuggyPro* to 9,190 context switches in *Tsp*. LEAN was able to greatly reduce the trace complexity for all the concurrency bugs in our experiments. The trace size is reduced by 49.9% (2x) in *Tsp* to as large as 99.7% (324x) in *LinkedList*, the number of threads is reduced by 40% to 99.3%, and the number of context switches is reduced by 50% to 99.6%. Moreover, the replay time is also greatly shortened after simplification, ranging from 58.9% (2.4x) in *Tsp* to 99.7% (327x) in *LinkedList*. In the four large server

Program	Whole Redundancy	Partial Redundancy	
		Slicing	Repetition
BuggyPro	445K(**96.9%**)	1.8K(**0.2%**)	-
Tsp	21.7M(**49.2%**)	0.4M(**0.7%**)	-
ArrayList	1.71M(**99.6%**)	-	-
LinkedList	2.19(**99.7%**)	-	-
OpenJMS	100.8M(**78.2%**)	7.3M(**5.7%**)	20.0M(**15.5%**)
Tomcat	23.6M(**61.9%**)	4.2M(**11.0%**)	9.1M(**24.0%**)
Jigsaw	16.0M(**79.4%**)	0.91M(**4.5%**)	2.7M(**13.4%**)
Derby	75.1M(**79.8%**)	6.2M(**6.6%**)	11.6M(**12.3%**)

Table 3. Decomposed effectiveness on trace size reduction

Program	ER	LEAN
BuggyPro	2.1%	97.1%
Tsp	0.0%	49.9%
ArrayList	2.9%	99.6%
LinkedList	3.0%	99.7%
OpenJMS	10.2%	98.5%
Tomcat	6.9%	96.7%
Jigsaw	4.6%	98.0%
Derby	2.5%	97.8%

Table 5. Comparison between LEAN and ER

applications, the replay time is consistently shortened by around 98% (64x).

Table 3 reports the simplification effectiveness w.r.t. each of the three components in terms of the trace size reduction. Column 2 reports the percentages of the whole-thread redundancy reduced by the hierarchical delta-debugging (HDD), while Columns 3-4 report that of the partial-thread redundancy, reduced by the slicing and the repetition analysis, respectively. In the small benchmarks, the percentage of whole thread redundancy ranges from 49.2% to 99.7%. LEAN did not identify much partial thread redundancy in these small benchmarks. Slicing removes only 0.2% and 0.7% redundancy, respectively, in *BuggyPro* and *Tsp*. For the real server programs, the percentage of whole-thread redundancy ranges from 61.9% to 79.8%. For the partial-thread redundancy, slicing and repetition analysis are both more effective than that for the small benchmarks. Slicing removes 4.5% to 11% redundant computation in the four large server programs, while the percentage of redundancy removed by the repetition analysis ranges from 12.3% to 15.5%. We note that the amount of redundancy in the buggy traces is closely related to the number of threads and the number of repetitions configured as input to the program. With more redundancy in the buggy trace, LEAN would have better simplification ratio. Nevertheless, we believe our result is representative as our experimental setup reflects the typical concurrency testing scenarios in the development cycle (such as the effective random testing in the IBM ConTest tool [6] and the stress testing in Chess [20]).

Comparing with the ER [27] The execution reduction (ER) technique proposed by Tallam et al. [27] also aims at reducing the trace size, for supporting the tracing of long running multithreaded programs. ER works by tracking a dynamic dependence graph of the execution events. The events are grouped into regions and threads such that the size of the dependence graph can be reduced. By analyzing the dependence graph, ER removes the regions of events or threads that are irrelevant to the fault. As ER relies on the dynamic dependence graph, it cannot remove the redundant computation that has data/control dependencies to the fault. While LEAN relies on the redundancy criterion and the dynamic verification, it is able to explore more simplification opportunities.

We compared the simplification effectiveness on the trace size reduction between LEAN and ER. Table 5 shows the result. For our evaluation benchmarks, LEAN is much more effective than ER. ER does not find many irrelevant events (the percentage of simplification ranges from 0.0%-10.2%), because almost all threads have data dependencies between each other on shared variables, while LEAN can effectively remove the redundant threads and the repetitive computation through the hierarchical delta-debugging and our repetition analysis.

6.3 RQ2: Efficiency

The goal of our second research question is to assess if our approach is efficient in simplifying the buggy trace. Since LEAN works in a black-box style (applying delta-debugging except for the dynamic slicing part) to iteratively simplify the trace, it may take a long time (many rounds) to produce the final simplification. As in each round it requires two replay runs to validate the redundancy (for the two redundancy conditions in our criterion), the efficiency of LEAN is an important concern for the usefulness in practice. Hence, during the trace simplification, we also record the number of delta-debugging rounds (for dealing with both the whole-thread redundancy and the partial-thread redundancy) and measure the time needed for each of the three components of LEAN to produce the final simplified trace. As we use the repetition analysis to identify the RCBs, we also report the statistics of the repetition analysis result to assess its usefulness in improving the simplification effectiveness of LEAN.

Table 4 shows the experimental results for our research question RQ2. Columns 2-3 and 5-6 report the number of simplification rounds (including the failed runs) and the time taking LEAN to remove the whole-thread redundancy and the redundant repetitions, respectively, from the original trace (the same trace as that in Table 2). Generally, the number of rounds is dependent on the amount of redundancy, while the simplification time is dependent on the amount of redundancy as well as the length of the original trace. For the small benchmarks, LEAN took 2 to 18 rounds for validating the whole-thread redundancy, which took 8 to 99 seconds of the execution time. For the large systems, since their traces

Program	HDD		Slicing	Repetition		RCB	
	#Rounds	Time	Time	#Rounds	Time	All	Real
BuggyPro	6	8s	155ms	-	-	4	0
Tsp	2	199s	12s	-	-	3	0
ArrayList	18	55s	2s	-	-	-	-
LinkedList	18	58s	2s	-	-	-	-
OpenJMS	13	4,265s	330s	11	152s	1	1
Tomcat	5	1,082s	308s	12	55s	1	1
Jigsaw	4	630s	210s	10	37s	1	1
Derby	4	1,841s	553s	10	200s	1	1

Table 4. Experimental results - RQ2: Efficiency

are much larger, LEAN took 4 to 13 rounds and 630 to 4,265 seconds to remove the whole-thread redundancy, and 10 to 12 rounds and 37 to 200 seconds to remove the redundant repetitions. Column 4 reports the time needed for slicing the trace (including both the construction time of the multithreaded dependence graph (MDG) and the analysis time for slicing the MDG). Because slicing considers all the instructions in the buggy execution, it is more expensive for large server programs (which have longer and more complex traces) than that for the small benchmarks. The slicing time for the four large server programs in our experiments ranges from 210 to 553 seconds.

Summary Compared to the original replay time, the simplification time is typically 4x-8x longer (except *Tsp*, which is in fact shorter). However, considering the significant trace simplification ratio, we believe the time cost is acceptable (even for the large systems). Moreover, as the simplification task is fully automatic (transparent to programmers) and can be easily parallelized, programmers do not need to worry about the simplification procedure. For very long running executions, programmers may also choose to set a time bound for the simplification. When the simplification does not finish within the time bound, programmers can still have the partially simplified trace (sharing the spirit of delta-debugging).

On the aspect of repetition analysis, Columns 7-8 report the total number of identified RCBs and the number of real RCBs among them in each benchmark. For the small benchmarks, our analysis identified 4 RCBs in *BuggyPro* and 3 in *Tsp*, but none of them are truly redundant. Our analysis does not report any RCB in *LinkedList* and *ArrayList*. For the large systems, our analysis successfully identified all the RCBs in the test drivers. In testing real concurrent systems, there is often a large number of repetitions (in order to increase the bug finding possibility). We note that the repetition analysis plays an important role in effectively reducing this kind of partial-thread redundancy, though (as our result suggests) the precision of our repetition analysis is not optimized.

7. Related Work

Concurrency bug reproduction has attracted extensive research efforts in the multicore era. However, there is little research that targets at simplifying the concurrency bug reproduction. We discuss several key related work in this section.

Thread interleaving simplification Chio and Zeller [3] first proposed a delta-debugging technique to isolate failure-inducing thread schedules in concurrent programs. The proposed technique is useful for identifying the context switch points that might cause the bug. Jalbert and Sen [11] first proposed a dynamic trace simplification technique that uses replay to reduce the thread context switches in the buggy trace. Our SimTrace [10] algorithm further improves the efficiency of trace simplification by statically exploring the trace dependence graph. In general, these techniques do not reduce the redundant computation in the replayed execution or the trace.

Program slicing Program slicing [28, 34, 39] has been widely used for debugging, that determines which parts of a program are relevant to a given program point of interest (i.e., the buggy statement). A number of efficient approaches [7, 12, 21] are also proposed for slicing concurrent programs. Essentially, slicing relies on the program/system dependence graphs (PDG/SDG) for the simplification, and it cannot simplify beyond the data and control dependencies in the program or the trace. Differently, our notion of redundancy is based on the bug reproduction property, which is not limited by the PDG/SDG, allowing us to explore more simplification opportunities. For instance, a computation may still be redundant even if it has data or control dependency to the buggy state, as long as without it the same bug can be reproduced. Weeratunge et al. [33] also propose a novel dual slicing approach to locate and explain the root cause of a concurrency failure by comparing the failing and correct schedules. Different from dual slicing, our approach aims at producing a simplified buggy execution, without the need of a correct schedule.

Multiprocess deterministic replay MDR has witnessed significant progress in recent years. The state of art software-only approaches, DoublePlay [30] supports low overhead

full-program replay by offloading the recording processes to extra cores, and our MDR system LEAP [9] enables the lightweight recording using a new type of local order to track the shared memory dependency. Many hardware approaches are also proposed to greatly reduce the recording overhead with special hardware design. Rerun [8] exploits episodic memory race recording to achieve efficient logging (around 4B per 1000 instructions), while DeLorean [17] promises much smaller log sizes and higher replay speeds by investigating the total sequence of chunk commits.

Offline search To further reduce the overhead to make concurrency bug reproduction applicable for the production use, researchers have also explored the idea of offline search with only partial runtime information. PRES [22] presents a feedback-based approach to use an intelligent replayer during diagnosis time to explore the unrecorded non-deterministic space. ODR [1] and Respec [13] develop lightweight online solutions and rely on offline search to achieve output-determinism (which is sufficient for bug reproduction). ESD [37] further reduces the runtime tracing overhead by symbolically exploring the complete thread scheduling decisions via execution synthesis. Weeratunge et al. [32] present an approach to generate a failure inducing schedule by comparing the core dumps at offline (utilizing the execution indexing technique [36]).

Replay log reduction and checkpointing Lee et al. [14] develop a novel log reduction technique that selectively records extra information and utilizes it to achieve reduction. A key ingredient of their work is the unit-based loop analysis (similar to our repetition analysis) that reduces redundant loop iterations based on unit annotation for loops. Another direction to simplify bug reproduction is through checkpointing [4, 35, 40]. By memoizating the program state that is near the buggy execution point, these techniques can help significantly reduce the replay time. A limitation of checkpointing for debugging is that it only reproduces a partial causal chain to the bug. In cases of bugs that contaminate program state from the beginning of the execution, programmers may need the full execution history to locate the root cause of bug.

8. Conclusion

Debugging concurrent programs has been a long-standing challenging problem. We have presented a novel technique LEAN to simplify the concurrency bug reproduction by removing the redundant computation from the buggy trace with the replay-supported execution reduction. Our experimental results show that LEAN is able to significantly reduce the complexity of the reproducible buggy execution and shorten the replay time. With LEAN, we believe the effectiveness of debugging concurrent programs can be greatly improved.

Acknowledgement

We thank the anonymous reviewers for their constructive comments. This research is supported by RGC GRF grants 622208 and 622909.

References

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

[2] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT*, 1998.

[3] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA*, 2002.

[4] William R. Dieter and James E. Lumpp Jr. A user-level checkpointing library for posix threads programs. In *FTCS*, 1999.

[5] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[6] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. *IPDPS*, 2003.

[7] Dennis Giffhorn and Christian Hammer. Precise slicing of concurrent programs. *Automated Software Engg.*, 2009.

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

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

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

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

[12] Jens Krinke. Context-sensitive slicing of concurrent programs. In *ESEC/FSE*, 2003.

[13] 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.

[14] Kyu Hyung Lee, Yunhui Zheng, Nick Sumner, and Xiangyu Zhang. Toward generating reducible replay logs. In *PLDI*, 2011.

[15] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ASPLOS*, 2008.

[16] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *ICSE*, 2006.

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

[18] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multi-processor replay. In *ASPLOS*, 2009.

[19] Madan Musuvathi and Shaz Qadeer. Chess: systematic stress testing of concurrent software. In *Proceedings of the 16th international conference on Logic-based program synthesis and transformation*, 2007.

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

[21] Mangala Gowri Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 2006.

[22] 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.

[23] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *Int. J. Softw. Tools Technol. Transf.*, 2007.

[24] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[25] John Steven, Pravir Ch, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *ISSTA*, 2000.

[26] Sriraman Tallam, Chen Tian, and Rajiv Gupta. Dynamic slicing of multithreaded programs for race detection. In *ICSM*, pages 97–106, 2008.

[27] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA*, 2007.

[28] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 1995.

[29] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[30] 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.

[31] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA*, 2001.

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

[33] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. In *ISSTA*, 2010.

[34] Mark Weiser. Program slicing. In *TSE*, 1984.

[35] K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. Micro-checkpointing: Checkpointing for multithreaded applications. In *IOLTW*, 2000.

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

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

[38] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 2002.

[39] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *PLDI*, 2004.

[40] Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ml. *J. Funct. Program.*, 2010.