A Trace Simplification Technique for Effective Debugging of Concurrent Programs

Nicholas Jalbert
EECS Department, UC Berkeley, CA, USA
jalbert@cs.berkeley.edu

Koushik Sen
EECS Department, UC Berkeley, CA, USA
ksen@cs.berkeley.edu

ABSTRACT

Concurrent programs are notoriously difficult to debug. We see two main reasons for this: 1) concurrency bugs are often difficult to reproduce, 2) traces of buggy concurrent executions can be complicated by fine-grained thread interleavings. Recently, a number of efficient techniques have tried to address the former reproducibility problem; however, there is no effective solution for the latter trace simplification problem. In this paper, we formalize and prove the trace simplification problem is NP-hard. We then propose a heuristic algorithm, Tinertia, that transforms a buggy execution trace into an easier-to-understand simplified trace. Tinertia works by automatically and iteratively increasing the granularity of the thread interleavings in the buggy trace. Tinertia cannot guarantee optimal simplification; however, we empirically show that our algorithm often generates optimally simplified traces. Moreover, we show that in the simplified trace, the locations of preemptive context switches point to the cause of the bug. We have implemented Tinertia in a tool for C/C++ programs using Pthreads and applied it to 11 benchmarks with up to 37,000 lines of code.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Experimentation, Reliability

1. INTRODUCTION

Software is becoming increasingly concurrent to take advantage of the multicore trend in hardware. Unfortunately, concurrent programs are notoriously difficult to debug compared to their sequential counterparts. We see two main reasons for this: 1) Bugs due to concurrency happen under very specific thread schedules and are often not reproducible during regular testing. As a result, the cyclic debugging techniques that are effective for sequential programs cannot be directly applied to concurrency bugs. 2) The traces generated by concurrent program executions can be complicated and difficult to understand. We believe that fine-grained thread interleavings are chief among these complications, especially when concurrency bugs are discovered by random scheduling based testing techniques [8, 29, 27, 13] or by recording real-world production executions.

For the reproducibility problem, a number of light-weight and efficient techniques have been proposed to record and replay a concurrent execution [22, 26, 6, 25, 34, 21, 16, 23, 1]. A record and replay system dynamically tracks the execution of a concurrent program, recording the non-deterministic choices made by the scheduler. A trace is produced which allows the program to be re-executed, forcing it to take the same schedule. If captured in a trace, a concurrency bug can be replayed consistently during debugging.

Despite advances on the reproducibility problem, the second problem remains—a trace of a buggy execution can be complicated by the fine-grained interleaving of various threads. Fine-grained thread interleavings significantly increase the number of potential thread interactions one must reason about to understand a trace. Hence, it is often desirable to create a simpler execution trace that shows the same bug but increases the granularity of thread interleavings.

To address this problem, we propose an algorithm, Tinertia, to automatically transform the trace of a buggy concurrent execution into a simpler trace with a coarser-grained thread interleaving that exhibits the same bug. The simplified trace can then be used in debugging instead of the more complicated original one, potentially relieving the programmer of some of the burden associated with reasoning about fine-grained dependencies among threads.

In this paper, we show that the general problem of simplifying a trace is NP-hard. Tinertia is a heuristic algorithm that runs in time polynomial in the size of the trace; it computes a locally optimal simplification instead of the globally optimal simplification. Tinertia simplifies a buggy trace by greedily performing merges and removals on a trace to generate simpler intermediate traces. An intermediate trace is then executed to validate that it is feasible and exhibits the bug. Thus, the Tinertia algorithm applies a purely experimental approach in the spirit of Delta Debugging [4] to the problem of trace simplification.

We have implemented Tinertia for C/C++ programs with Pthreads in a tool called Thrille and applied this tool to 11 benchmark programs having 250 to over 37,000 lines of code with known or seeded bugs. Our experiments show

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$10.00.

```
// Thread 1              // Thread 2
flag = 1;                while (! flag)
x = 1;                       sleep();
if (x == 3)              x = 3;
   error();              flag = 0;
```

Figure 1: Example Program.

```
1                          (T2, while (! flag))
2                          (T2, sleep())
3    (T1, flag = 1)
4                          (T2, while (! flag))
5    (T1, x = 1)
6                          (T2, x = 3)
7    (T1, if (x == 3))
8                          (T2, flag = 0)
9    (T1, error())
```

Figure 2: Example buggy trace.

Thrille produces nearly optimal simplified traces. Moreover, we show that the simplified traces have at most 2-3 preemptive context switches (excepting one benchmark) supporting the observation that most concurrency bugs can be caught with few preemptive context switches [17]. A manual check of a number of simplified traces confirmed the cause of the bug could be easily pinpointed by examining remaining preemptive context switches. Finally, we address the argument that a debugging tool like Thrille is unnecessary because one could use a context bounded model checker such as CHESS [17] that returns a simplified trace by default to find concurrency bugs; we empirically show that a combination of race directed random testing and simplification is more efficient than basic context bounded model checking for a large fraction of our benchmarks.

The contributions of this paper are as follows:

- We formally model the trace simplification problem and prove it is NP-hard.
- We propose a heuristic trace simplification algorithm, Tinertia, which greedily applies two types of simplifying operations to a buggy trace. We describe the rationale behind each operation and provide empirical evidence as to their effectiveness.
- We describe an implementation of the Tinertia algorithm for C/C++ programs. We experiment on 11 programs from the PARSEC and Inspect benchmark suites to show the efficacy of Tinertia empirically.

2. OVERVIEW

In this section, we informally describe the operation of our trace simplification algorithm, Tinertia, on an example. Consider the program in Figure 1. Assume all variables are initially 0, all statements execute atomically on a sequentially consistent memory model, and both threads run concurrently. Notice there is a data race over setting the value of variable x. If Thread 2 executes `x = 3` after Thread 1 executes `x = 1` but *before* Thread 1 checks the branch condition, the program will hit an error.

We define an *action* to be an atomic statement executed by a particular thread. For example, Thread 1 executing the statement `flag = 1` is an action. A *trace* of a program execution is a list of actions. Figure 2 is an example of a trace that exhibits a bug. A trace can be viewed as a description of a serialized execution of a multithreaded program that captures a thread interleaving. We call two consecutive actions in a trace a *context switch* if the actions belong to different threads. The trace in Figure 2 has seven context switches. A *non-preemptive context switch* is a context switch where the descheduled thread is disabled by the semantics of the program. A *preemptive context switch* or *preemption* is a context switch where the descheduled thread could have continued execution. All context switches in the trace in Figure 2 are preemptive context switches except for the context switch occurring between action 8 and action 9. This is a non-preemptive context switch because Thread 2 terminates and cannot legally execute any instruction.

Using terminology from [30], we refer to a maximal set of consecutive actions by the same thread as a *thread execution interval* (TEI). The first two actions of the trace in Figure 2 where Thread 2 executes an iteration of the spin loop is an example of a TEI of Thread 2. Every trace is composed of one or more TEIs, and the number of context switches in a trace is one less than the number of TEIs.

The input to Tinertia is a trace of a buggy multithreaded execution and the output is a simpler trace which exhibits the same bug. We describe the output trace as a *variant-trace* of the original input trace. A bug can be any distinguishable behavior of a program; typical bugs include deadlocks, segmentation faults, and assertion failures.

Our metric for trace simplicity (and interleaving granularity) is the number of context switches in a trace: a trace with fewer context switches is simpler than a trace with more context switches. Given a buggy trace τ, we would like to generate the simplest variant-trace of τ. That is, we would like to generate the variant-trace with the minimal number of context switches of all variant-traces. In Section 3, we show this trace simplification problem is NP-hard.

Tinertia is a heuristic algorithm that iteratively applies a simplifying operation to a trace to generate a simpler intermediate trace. An intermediate trace is validated by executing it. If the intermediate trace describes a feasible program execution and exhibits the bug, i.e. it is a variant-trace, Tinertia keeps the intermediate trace and continues to apply simplifying operations to it. If the intermediate trace is invalid or does not exhibit the bug, the simplifying operation is undone and a different simplifying operation is tried. Tinertia terminates when no simplifying operation can be applied to produce a valid variant-trace.

The core simplifying operations performed by Tinertia are the removal of the last TEI of a thread and the merging of two TEIs. We further subdivide the merging operation into two different forms— Two-Stage Consolidate Up and Consolidate Down. We now describe the simplifying operations that Tinertia performs on a trace:

- **Remove Last:** This operation is applied to a specific thread t in the trace and generates a simpler trace by removing the last TEI of thread t. This operation can remove extraneous threads and TEIs which do not contribute to the bug. If this operation is successful, at least one context switch is removed.
- **Two-Stage Consolidate Up:** This operation is applied to a specific action a in the trace and only has an effect if a is the last action of a TEI. Let thread t be the thread that executes action a. This operation is two-stage in the sense that it first operates at the TEI granularity and then at the action granularity. In the first stage, the operation finds the next TEI after a

```
1                                  (T2, while (! flag))
2                                  (T2, sleep())
3     (T1, flag = 1)
4                                  (T2, while (! flag))
5     (T1, x = 1)
6                                  (T2, x = 3)
7     (T1, if (x == 3))
8     (T1, error())
```

Figure 3: The variant-trace generated by the Remove Last simplifying operation.

which is executed by thread t in the trace. This operation then generates a new trace by removing that TEI and inserting it directly after a. If this move produces a valid variant-trace, the operation terminates.

If the first stage is not successful, the second stage is executed. The moved TEI is returned to its original trace position. Then the maximal (potentially empty) prefix of actions of that TEI is removed and inserted after a such that a valid variant-trace is produced.

If successful, the first stage of the operation removes at least one context switch; this can eliminate context switches unrelated to the bug that resulted from scheduling nondeterminism. The second stage removes no context switches overall, but can eliminate preemptive context switches by extending a TEI until the executing thread becomes disabled.

- **Consolidate Down:** This operation is symmetric to the first stage of Consolidate Up. It is applied to a specific action a in the trace and only has an effect if a is the first action in a TEI. Let thread t be the thread that executes action a. This operation finds the most recent TEI of thread t which was executed previous to a, and then generates a new trace by removing that TEI and inserting it directly before a. Each successful application of this operation eliminates at least one context switch. This operation can remove spin loops in traces where thread t_1 is waiting for thread t_2 to act; in this case, moving t_1's TEI to the point after t_2 performs its action causes t_1 to no longer wait.

`Tinertia` first applies the Remove Last operation to each thread until no progress can be made. Then a forward pass is made over the trace, and the Two-Stage Consolidate Up operation is applied to each action in the trace. Following that, `Tinertia` applies the Consolidate Down operation to each action in a reverse pass. Once this is completed, `Tinertia` starts again with Remove Last operation, and continues through the phases until no progress can be made.

We now describe the application of the `Tinertia` algorithm to the buggy trace in Figure 2. `Tinertia` first applies the Remove Last operation to each thread in the trace until no progress can be made. Figure 3 shows the resulting variant-trace. The algorithm was able to successfully remove the action corresponding to Thread 2 resetting the flag (action 8 in Figure 2) because this action is extraneous and does not affect Thread 1 hitting `error()`. This operation removes two context switches.

Next, `Tinertia` takes the variant-trace that resulted from applying the Remove Last operation and attempts to apply the Two-Stage Consolidate Up operation to each action in turn, going from the first action in the trace to the last action. The result of this pass is the variant-trace in Figure 4.

```
1                                  (T2, while (! flag))
2                                  (T2, sleep())
3     (T1, flag = 1)
4     (T1, x = 1)
5                                  (T2, while (! flag))
6                                  (T2, x = 3)
7     (T1, if (x == 3))
8     (T1, error())
```

Figure 4: The variant-trace generated by the Two-Stage Consolidate Up simplifying operation.

```
1     (T1, flag = 1)
2     (T1, x = 1)
3                                (T2, while (! flag))
4                                (T2, x = 3)
5     (T1, if (x == 3))
6     (T1, error())
```

Figure 5: The variant-trace generated by the Consolidate Down simplifying operation. This will be the simplified trace returned by `Tinertia`.

When applied to the second action in Figure 3, the first stage of the Two-Stage Consolidate Up operation fails to produce a valid variant-trace. In this case, it attempts to move the TEI consisting of action 4 and place it directly after action 2. This, however, is an invalid trace because the flag has not been set yet. Thread 2 will execute the spin loop again, and will not execute the statement `x = 3` at the appropriate point. The second stage of Two-Stage Consolidate Up also does not make progress, as the only non-empty prefix of the moved TEI is the entire TEI.

However, a valid variant-trace is generated when the Two-Stage Consolidate Up operation is applied to action 3 of the trace of Figure 3. In this case, the operation moves the TEI consisting of action 5 to directly after action 3. Thread 1 setting `x` (action 5) and Thread 2 checking the flag (action 4) are independent and thus commute. This variant-trace has two fewer context switches and is the only progress that the Two-Stage Consolidate Up operation makes in this pass.

`Tinertia` then takes the trace that resulted from the Two-Stage Consolidate Up pass in Figure 4 and attempts to apply the Consolidate Down operation to each action in the trace, starting at the last action and working toward the beginning. The result of this pass can be seen in Figure 5.

Consolidate Down is able to eliminate Thread 2's initial iteration of the spin loop. When the operation is applied to action 5 in the trace in Figure 4, it moves Thread 2's TEI corresponding to the execution of the spin loop past the setting of `flag`. This eliminates the need for Thread 2 to spin at all. Notice the actual trace generated by Consolidate Down has Thread 2 spinning once after the flag is set, while Thread 2 will not spin at all when the trace is validated by execution. We discuss how we implement this *variant-trace validity approximation* in Section 4.2.1. The result of this pass is that one context switch is removed.

`Tinertia` now attempts to again apply each operation to the trace in Figure 5. No operation will be able generate a valid variant-trace, so `Tinertia` terminates and returns the variant-trace in Figure 5 as the simplified trace. In this example, the `Tinertia` algorithm returns the variant-trace that is minimal in the number of context switches (with five fewer context switches than the original trace).

Delta Debugging is an automated search strategy for simplifying inputs which cause failure [36]. The Delta

Debugging-based algorithm proposed by Choi et al. in [4] is related to `Tinertia`; however, the goals of the two algorithms are orthogonal. Their algorithm takes as input a passing trace p and failing trace f and generates a new passing trace p' and new failing trace f' such that every difference between p' and f' is necessary to cause the bug. The insight in that work is the differences between p' and f' will illuminate the cause of the bug. However, there are no facilities in their algorithm to try to produce a simpler trace; the simplicity of the resulting p' and f' depend heavily on the input traces. Indeed, one can imagine fruitfully combining the two algorithms by first simplifying the failing trace and then using the simplified trace as input to the Delta Debugging algorithm.

The Execution Reduction system described in [30] by Tallam et al. has a similar goal to `Tinertia`, i.e. to simplify a buggy trace. Their work targets event-driven systems and does a thorough tracking of dynamic dependencies between threads to identify and remove irrelevant threads and TEIs. `Tinertia`, on the other hand, takes a purely experimental approach, which could result in better simplification in cases where dynamic dependencies semantically commute. Their primary metric of simplification is the length of a trace. We argue that the number of context switches, i.e. the granularity of the thread interleavings, is an important metric of the simplicity of a trace, and describe three operations which can be applied to a trace to reduce the number of context switches. One could usefully combine the Execution Reduction system with `Tinertia`–first run Execution Reduction on a buggy trace to remove unnecessary threads and TEIs and then run `Tinertia` on the resulting trace to increase the granularity of the thread interleavings.

3. ALGORITHM

Now that we have informally examined the operation of the `Tinertia` algorithm on a small example, in this section we formally model the problem of trace simplification, and situate the `Tinertia` algorithm within our formal model.

3.1 Background Definitions

Consider a multithreaded program execution. Let T be the set of threads that are in the execution and let S be the set of instructions executed by the threads in the execution. We call the execution of an instruction $s \in S$ by a thread $t \in T$ an *action* and denote it by the pair (t, s). We use $t(a)$ and $s(a)$ to denote the thread and instruction of the action a, respectively. A multithreaded program execution can be seen as a sequence of actions. We call such a sequence a *trace* and denote it by τ. We will use $\tau[i]$ to denote the i^{th} action in the trace τ and $\tau[j:k]$ to denote the subsequence $\tau[j]\tau[j+1]\ldots\tau[k]$. In the rest of the discussion, assume that we are given a trace $\tau = a_1a_2\ldots a_n$.

We refer to a maximal set of consecutive actions by the same thread in a trace as a *thread execution interval* (TEI). A TEI is a subsequence $\tau[j:k]$ of trace τ, defined as follows:

- $j = 1$ or $t(\tau[j-1]) \neq t(\tau[j])$,
- $k = n$ or $t(\tau[k+1]) \neq t(\tau[k])$, and
- $\forall l \in [j,k]$, $t(\tau[j]) = t(\tau[l])$

We say a trace of a multithreaded execution exhibits a bug if at the end of the execution the program state satisfies a predicate, say ϕ, that denotes the bug. For the rest of this discussion, assume trace τ exhibits a bug by satisfying ϕ at the end of its execution. The goal of `Tinertia` is to compute a "simplified" trace τ_{min} such that the state of the program after executing τ_{min} satisfies ϕ. We next define the notion of simplification formally.

We say that two actions a_i and a_j are related by the *program-order* relation, denoted by $a_i \rightarrow_p a_j$, iff $i < j$ and $t(a_i) = t(a_j)$. Let us call a trace τ' a *variant-trace* of τ if the following conditions hold:

- τ' is a permutation of a subset of the actions of τ,
- τ' denotes the prefix of a trace of a feasible multithreaded program execution,
- the program state after executing τ' satisfies ϕ, and
- τ' conforms to the program-order relation, i.e. if $a_i \rightarrow_p a_j$ in τ and a_j is present in τ', then a_i appears before a_j in τ'.

We can also associate a *cost* function with a trace—the cost of a trace is equal to the number of context switches in the trace. Note that $cost(\tau)$ is equal to the number of TEIs in τ minus one.

3.2 Problem Definition

As described earlier, we are given a trace $\tau = a_1a_2\ldots a_n$ of a multithreaded execution that satisfies the bug predicate ϕ at the end of the execution. We want to find a variant-trace τ_{min} such that $cost(\tau_{min})$ is the minimum of the cost of all variant-traces of τ. Formally, let $\mathcal{T}_{min} = \{\tau' \mid \tau'$ is a variant-trace of τ and for all variant-traces τ'' of τ, $cost(\tau') \leq cost(\tau'')\}$. The goal of our algorithm is to return an element of $\mathcal{T}_{min}$. Such a trace minimizes the number of context switches in the multithreaded execution. In our view, such a trace is simpler than the original buggy trace.

Unfortunately, the problem of finding an element of $\mathcal{T}_{min}$ is NP-hard. We next prove this claim.

THEOREM 3.1. *The problem of finding an element of $\mathcal{T}_{min}$ is NP-hard.*

Proof. (Sketch) Consider a pair $(A, \preceq)$ where A is a set of actions and $\preceq$ is a partial-order relation on A. We require that $\preceq$ minimally relates all pairs of actions $a_i, a_j \in A$ such that a_i and a_j are from the same thread. Inter-thread dependencies may also appear in $\preceq$. An observed buggy trace is a linear-order on A such that the trace conforms to $\preceq$, i.e., if a_i appears before a_j in τ then it must not be the case that $a_j \preceq a_i$. Assume that the bug is exhibited by any trace which is a linear-order on A and conforms to $\preceq$. Then a trace simplification problem $(A, \preceq)$ is to come up with a linear-order on A that conforms to $\preceq$ and has the minimum number of TEIs. Note that we use $\preceq$ to represent the error predicate ϕ. In a real program, we can assume that $\preceq$ represents the reflexive transitive closure of the union of the data dependency and program-order relations. Given $(A, \preceq)$, it is easy to construct a program with an execution that satisfies ϕ iff the execution of the program has all actions in A and conforms to $\preceq$.

In the vertex cover problem, we are given an undirected graph with vertices V and edges E. The task is to select the minimum set of vertices $W \subseteq V$ such that every edge in the graph is adjacent to one of the vertices in W. The vertex cover problem is well known to be NP-complete.

We now reduce vertex cover to the trace simplification problem. We assume we are given an arbitrary graph $G =$

(V, E) on which we must find the minimum vertex cover. We now construct a trace simplification problem $(A, \preceq)$ whose solution would correspond to the minimum vertex cover. For each vertex in $v \in V$, we create a unique thread t_v. For each vertex v we then generate two actions: (t_v, i_1) and (t_v, i_2) in A. Since in our problem setting, we need a total-order over all the actions from a given thread, we add the relation $((t_v, i_1), (t_v, i_2))$ to $\preceq$ for each vertex v. For each edge uv in E, we add $((t_u, i_1), (t_v, i_2))$ and $((t_v, i_1), (t_u, i_2))$ to $\preceq$.

Let τ be a linear-order on A that conforms to $\preceq$ and minimizes the number of TEIs. Let T be the set of threads such that for each $t \in T$, τ has two TEIs from the thread t. Let W be the set of the vertices that correspond to the threads in T. Then W is the minimum vertex cover for G because, by construction, each thread must have at least one TEI in τ. If there is an edge between two vertices in G, at least one of the threads corresponding to the two vertices defining the edge must have two TEIs in τ by the way we constructed the relation $\preceq$. Thus, the set of vertices whose corresponding threads have more than one TEI in τ is a vertex cover, and it is minimal because τ has a minimal number of TEIs. This shows that trace simplification problem is NP-hard. □

3.3 `Tinertia` Algorithm

Given the problem of trace simplification is NP-hard, we do not expect to find an efficient algorithm for it. Therefore, we propose `Tinertia`, a heuristic algorithm.

Let us define three primitive operations on a trace:

- *remove*(τ, i) is the trace obtained by removing $\tau[i]$ from τ. For example, if $\tau = a_1 \dots a_{i-1} a_i a_{i+1} \dots a_n$, then *remove*$(\tau, i) = a_1 \dots a_{i-1} a_{i+1} \dots a_n$.
- *insert*(τ, i, a) is the trace obtained by inserting the action a immediately after the action $\tau[i]$. For example, if $\tau = a_1 \dots a_i a_{i+1} \dots a_n$, then *insert*$(\tau, i, a) = a_1 \dots a_i a a_{i+1} \dots a_n$.
- *move*(τ, i, j) is the trace *insert*$($*remove*$(\tau, j), i, \tau[j])$

Note that the operations *remove* and *move* could be applied repeatedly to obtain any trace that is a permutation of a subset of the actions of the original trace.

We next define four composite operations that we will use to describe our algorithm.

1. Operation *move-up-tei*(τ, i) moves the next TEI after the action $\tau[i]$ of thread $t(\tau[i])$ immediately after the action. Formally, *move-up-tei*(τ, i) is valid iff $\tau[i]$ is the last action of a TEI in τ and $\tau[j : k]$ is the next TEI of the thread $t(\tau[i])$ after the action $\tau[i]$. Then let $\tau_{j-1} = \tau$ and $\tau_l =$ *move*$(\tau_{l-1}, i + l - j, l)$. Then *move-up-tei*$(\tau, i) = \tau_k$. Note that if *move-up-tei*(τ, i) is valid and a variant-trace of τ, then *cost*$(\tau) \geq$ *cost*$($*move-up-tei*$(\tau, i)) + 1$.
2. Operation *move-up-actions*(τ, i, m) moves the prefix $\tau[j : m]$ of the next TEI after the action $\tau[i]$ of thread $t(\tau[i])$ immediately after the action. Formally, *move-up-actions*(τ, i, j) is valid iff $\tau[i]$ is the last action of a TEI and $\tau[j : m]$ is a prefix of the next TEI of the thread $t(\tau[i])$ after the action $\tau[i]$. Then let $\tau_{j-1} = \tau$ and $\tau_l =$ *move*$(\tau_{l-1}, i + l - j, l)$. Then *move-up-actions*$(\tau, i) = \tau_m$. Note that if *move-up-actions*(τ, i) is valid and a variant-trace of τ, then *cost*$(\tau) =$ *cost*$($*move-up-actions*$(\tau, i, m))$.

Algorithm 1 Greedy Simplification Algorithm

```
 1: Input: a trace $\tau$ and a bug predicate $\phi$
 2: $\tau_{cur} \leftarrow \tau$
 3: repeat
 4:    $\tau_{old} \leftarrow \tau_{cur}$
 5:    for $i = |\tau_{cur}|$ to 1 do
 6:       if remove-last-tei$(\tau_{cur}, i)$ is a variant-trace of $\tau$ then
 7:          $\tau_{cur} \leftarrow$ remove-last-tei$(\tau_{cur}, i)$
 8:       end if
 9:       $i \leftarrow$ minimum of $(i - 1)$ and $|\tau_{cur}|$
10:    end for
11:    for $i = 1$ to $|\tau_{cur}|$ do
12:       if move-up-tei$(\tau_{cur}, i)$ is a variant-trace of $\tau$ then
13:          $\tau_{cur} \leftarrow$ move-up-tei$(\tau_{cur}, i)$
14:       else if move-up-actions$(\tau_{cur}, i, m)$ is a variant-trace of
          $\tau$ for some maximal $m$ then
15:          $\tau_{cur} \leftarrow$ move-up-actions$(\tau_{cur}, i, m)$
16:       end if
17:    end for
18:    for $i = |\tau_{cur}|$ to 1 do
19:       if move-down-tei$(\tau_{cur}, i)$ is a variant-trace of $\tau$ then
20:          $\tau_{cur} \leftarrow$ move-down-tei$(\tau_{cur}, i)$
21:       end if
22:    end for
23: until cost$(\tau_{old}) \leq$ cost$(\tau_{cur})$
24: return $\tau_{cur}$
```

3. Operation *move-down-tei*(τ, i) could be defined symmetrically. It moves the previous TEI before the action $\tau[i]$ of thread $t(\tau[i])$ immediately before the action. Formally, *move-down-tei*(τ, i) is valid iff $\tau[i]$ is the first action of a TEI and $\tau[j : k]$ is the previous TEI of the thread $t(\tau[i])$ before the action $\tau[i]$. Then let $\tau_{k+1} = \tau$ and $\tau_l =$ *move*$(\tau_{l+1}, i - 2, j)$. Then *move-down-tei*$(\tau, i) = \tau_j$. Note that if *move-down-tei*(τ, i) is valid and a variant-trace of τ, then *cost*$(\tau) \geq$ *cost*$($*move-down-tei*$(\tau, i)) + 1$.
4. Operation *remove-last-tei*(τ, i) removes the last TEI of thread $t(\tau[i])$ if $\tau[i]$ is the first action of the last TEI of the thread $t(\tau[i])$. Formally, *remove-last-tei*(τ, i) is valid iff $\tau[i : j]$ is a TEI for some j and $\tau[i : j]$ is the last TEI of the thread $t(\tau[i])$. Then let $\tau_{j+1} = \tau$ and $\tau_l =$ *remove*(τ_{l+1}, l). Then define *remove-last-tei*$(\tau, i) = \tau_i$. Note that if *remove-last-tei*(τ, i) is valid and a variant-trace of τ, *cost*$(\tau) \geq$ *cost*$($*remove-last-tei*$(\tau, i)) + 1$.

Algorithm 1 is a formal description of the `Tinertia` algorithm. `Tinertia` takes as input a trace τ and a bug predicate ϕ which is satisfied by the state of the program after the execution of τ. All simplifying operations are applied to τ_{cur}, and whenever the application of an operation produces a valid variant-trace τ', that trace is stored into τ_{cur}.

`Tinertia` begins by initializing τ_{cur} to the input trace τ. The algorithm then enters the main loop. In lines 5–10, the algorithm does a reverse pass over trace τ_{cur}, applying the *remove-last-tei* operation to to each index of the trace. This is equivalent to the reverse pass of the Remove Last simplifying operation described in Section 2. If the application of *remove-last-tei*(τ_{cur}, i) produces a valid variant-trace of τ_{cur} for some index i, it is unnecessary to execute the last TEI of thread $t(\tau_{cur}[i])$ to produce the bug.

In lines 11–17, `Tinertia` does a forward pass over the variant-trace generated by the previous pass. This is a pass of the Two-Stage Consolidate Up simplifying operation described in Section 2. For each action in the trace τ_{cur}, the algorithm first performs the *move-up-tei* operation. If *move-up-tei*(τ_{cur}, i) produces a valid variant-trace, two TEI were successfully merged, and the simplified variant-trace is saved. If the application of *move-up-tei* fails, the algorithm then performs *move-up-actions*(τ_{cur}, i, m) for some maximal m. That is, the maximal (potentially empty) prefix of the TEI following $\tau_{cur}[i]$ is merged with the TEI of $\tau_{cur}[i]$. This operation can remove a preemptive context switch and replace it with a non-preemptive context switch.

The final pass of the main loop is performed in lines 18–22. This is a reverse pass of the Consolidate Down simplifying operation over the variant-trace generated by the previous two passes. The operation *move-down-tei* is applied to each action in the trace. If the application of *move-down-tei* creates a valid variant-trace, two TEI were successfully merged.

Tinertia executes the main loop until no composite operation applied to any action of the trace τ_{cur} can produce a valid variant-trace. Tinertia then terminates and returns τ_{cur} as the simplified trace.

THEOREM 3.2. *The worst-case time complexity of Algorithm 1 is* $O(|\tau|^3)$.

Proof. (Sketch) Each of the for loops at line 5, 11, and 18 has an upper bound of $|\tau|$ iterations. $cost(\tau)$ decrements by at least one in each iteration of the outer loop at line 3; so, the outer loop can run for a maximum of $|\tau|$ iterations. Therefore, $O(|\tau|^2)$ is an upper bound on the number of simplifying operations and variant trace validity checks, which cost $O(|\tau|)$. Therefore the entire algorithm is $O(|\tau|^3)$. □

THEOREM 3.3. *The trace τ_{cur} returned by Algorithm 1 is a local minimum in the following sense. There exists no i such that $remove(\tau_{cur}, i)$ is a variant-trace of τ and $cost(remove(\tau_{cur}, i)) < cost(\tau_{cur})$. And, there exists no i and j such that $move(\tau, i, j)$ is valid and a variant-trace of τ and $cost(move(\tau_{cur}, i, j)) < cost(\tau_{cur})$.*

4. IMPLEMENTATION

We have implemented the Tinertia algorithm in a tool called Thrille for C/C++ programs that use Pthreads. Our implementation is divided into two components: trace transformation and trace validation. The trace transformation component iteratively generates simplified traces; the trace validation component determines if a generated trace is a valid variant-trace.

Thrille takes as input a program and a replayable buggy trace. In general, deterministic replay requires controlling all sources of non-determinism in the program. For our benchmarks, it is sufficient to fix inputs, control the scheduler, and deterministically seed random number generators.

Thrille controls scheduler non-determinism by serializing program execution and selectively allowing threads to execute. Dynamic library interposition is used to intercept the synchronization events in a program. When a synchronization call is made, Thrille takes over execution and decides which thread to schedule next.

In programs with data races, bugs may manifest under finer-grained interleavings than is possible to achieve while scheduling only at synchronization events. To capture and reproduce these bugs, we compile the target program (or program module) using the LLVM tool chain [14]. During compilation, we execute a pass which instruments all load and store instructions with a call into Thrille. For each of our benchmarks, we run race detection to identify potentially racing memory accesses and then during simplification Thrille makes scheduling decisions at these accesses.

4.1 Trace Transformation

The trace transformation component repeatedly applies the Remove Last, Two-Stage Consolidate Up, and Consolidate Down simplifying operations to generate simpler intermediate traces.

If merging two TEIs fails in the first stage of Two-Stage Consolidate Up, we then must find the maximal prefix of the second TEI that can be merged with the first TEI in stage two. For efficiency, we approximate this by automatically examining the trace of the failing execution and generating a new trace that merges all actions that were executed from the second TEI in the failing trace.

4.2 Trace Validation

The trace validation component executes a trace τ' to determine if τ' is a valid variant-trace of the original buggy trace τ. Minimally, this component must be able to replay a trace and detect any program errors.

In Thrille, a trace is simply a listing of which thread to execute at each synchronization or racing memory operation (and not, say, a record of all memory reads and writes). Our implementation models Pthread semantics to determine what executions are legal. A trace is executed until an error is detected or the program terminates normally. The trace validation component detects errors including deadlocks, segmentation faults, and assertion failures. We note assertions allow for arbitrarily precise descriptions of bugs.

4.2.1 Approximating Variant-Trace Validity

In our initial experiments, we observed that a strict variant-trace validity check unnecessarily discards many intermediate simplified traces. This is because reordering TEIs can reorder memory dependencies and change the control flow within a TEI. Such reordering can result in a simplified trace which still exhibits the bug, but is not necessarily a variant-trace of the original trace, i.e. the modified trace is not a strict linear order of a subset of actions of the original trace. Thrille implements an approximation of the variant-trace validity check which is designed to tolerate variations between a trace and the execution induced by the trace. During execution, Thrille attempts to execute each TEI in the order in which it appears in the trace. The following inconsistencies can occur:

- **Control Flow Changes:** A thread may deviate from its expected execution path. If this occurs, the thread is executed until its execution rejoins the expected path, in which case the remaining actions of the TEI are completed. The thread may also block or enter livelock without rejoining the TEI, in which case the next TEI in the trace is executed.
- **Disabled Threads:** A thread may block before executing its whole TEI. If this occurs, the next TEI in the trace is executed.
- **Condition Variables:** A previously missed signal may no longer be missed, in which case a waiting thread is randomly chosen to signal.
- **No Bug:** At the end of a trace, if the program has not terminated but also does not exhibit the desired

bug, we continue execution in a non-preemptive fair way [18] until a bug is found or execution terminates.

When execution terminates, Thrille performs an approximation of the variant-trace validity check. Let τ'_{actual} be the trace of the actual execution induced by the trace τ' after all inconsistencies are accounted for. If τ'_{actual} exhibits the bug and $cost(\tau'_{actual}) \leq cost(\tau')$, then τ' is considered a valid variant-trace and simplification continues on τ'_{actual}. If τ'_{actual} does not satisfy both of these properties, τ' is considered an invalid variant-trace. Note this approximation guarantees termination of the algorithm and that any returned simplified trace will exhibit the same bug as the original trace.

5. EXPERIMENTAL EVALUATION

We evaluated Thrille on 11 C and C++ multithreaded benchmarks. Our set of benchmarks consists of programs from the PARSEC benchmark suite [3] and the Inspect benchmark suite [35]. The following programs came from PARSEC: *blackscholes* is an option pricing simulation, *canneal* implements a simulated annealing algorithm to minimize the routing cost of a chip design, *dedup* is a data stream compression program, *streamcluster* is an online clustering kernel, and *x264* is a threaded video compression library.

The following programs came from the Inspect benchmark suite: *bbuf* is a toy implementation of a shared buffer, *bzip2* and *pbzip2* are both multithreaded compression programs, *ctrace* is a multithreaded tracing library, *pfscan* is a parallel file scanner, and *swarm* is a parallel sort implementation.

To generate the initial buggy traces, we implemented a variant of race directed random testing [28], where we considered both data races and lock contentions. The benchmarks *pbzip2* and *swarm* had bugs which we could reproduce in our trace generation step. For the other programs, we seeded bugs by modifying synchronization operations. We chose this approach because we do not propose a novel way to discover bugs, and thus the exact nature of the bug is less important. We ensured that all seeded bugs were concurrency-related, i.e. all programs with seeded bugs successfully ran under normal circumstances and the bugs manifested non-deterministically under rare schedules.

All experiments were performed on a dual socket quad-core Xeon server with 8GB of RAM. All results are averaged over 30 runs. Each run consisted of generating a new buggy trace with our race directed random testing implementation and then applying Thrille to the trace.

The goal of these experiments was to evaluate the following four hypotheses:

1. The locally optimal simplified traces generated by the Tinertia algorithm are close to the global optimal in terms of number of context switches.
2. Each simplifying operation performed by the Tinertia algorithm contributes to the overall simplification.
3. Remaining preemptions in simplified traces are useful for pinpointing concurrency bugs in the program.
4. A debugging tool like Thrille can be useful even though context bounded model checking finds simplified buggy traces by default.

5.1 Optimality of Simplification

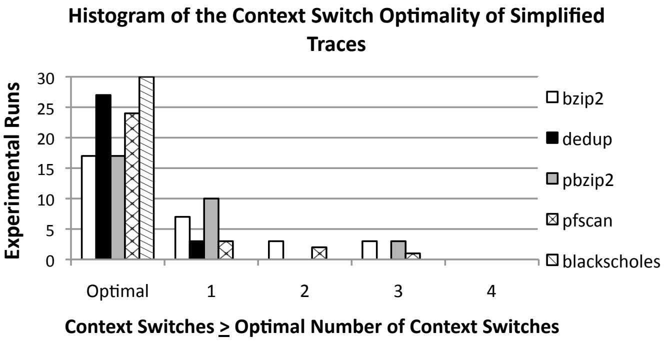

Figure 6: Optimality of simplified traces.

Table 1 shows the results of the experiments. The second column shows the size of each benchmark in lines of C and C++ code. Columns 3–7 show the average characteristics of the start traces. Size is the number of synchronization and memory operations at which Thrille made a scheduling choice. Thr is the number of threads which execute at least one action in a trace, and Ctxt is the number of context switches in a trace. The Non and Pre columns further break down the number of context switches. Non reports the number of non-preemptive context switches in the trace; Pre reports the number of preemptive context switches.

Columns 8–12 report the averages of these characteristics for the simplified traces. Note that due to the variant-trace validity approximations described in Section 4.2.1, certain characteristics (e.g. Size) of a trace can actually increase. For all benchmarks except for *ctrace*, Thrille generated simplified traces which averaged 2-3 preemptive context switches. Because a program should behave correctly irrespective of whether a preemptive context switch is made, we expect that any remaining preemptions in a simplified trace are necessary to cause the concurrency bug. Therefore, a simplified trace with few preemptive context switches will significantly reduce the debugging effort by reducing the number of places in the trace where we need to look for the cause of the bug.

Columns 13–15 report the average percent reduction of the different types of context switches due to simplification. For all benchmarks, Thrille was able to generate simplified traces with 92% or more of the preemptions removed (column 15).

The percent reduction in overall number of context switches varies more widely (column 13); the maximum possible reduction is dependent on both the design of the benchmark program and how the bug manifests. To validate our hypothesis that Tinertia generates nearly optimal traces with respect to number of context switches, we manually generated buggy traces with an optimal number of context switches for five of our benchmarks. Note that we did not do this for all benchmarks because creating the optimal trace for each benchmark is a tedious and time-consuming process. For each simplified trace generated in our experiments from the examined benchmarks, we binned the difference between the number of context switches in the simplified trace and the number of context switches in the optimal traces. Figure 6 shows a histogram of the results of this study.

For all the examined benchmarks, over 90% of the simplified traces were within 2 context switches of optimal. Moreover, the Tinertia algorithm does nontrivial work in all cases. Average percentage reduction in context switches ranges from 31% for *blackscholes* to over 90% for all other examined benchmarks (column 13 in Table 1). These re-

Program	LOC	Start Trace					Simplified Trace					% Reduction		
Name		Size	Thr	Ctxt	Non	Pre	Size	Thr	Ctxt	Non	Pre	Ctxt	Non	Pre
bbuf	255	199.6	11.0	168.4	57.1	111.3	175.7	11.0	21.5	19.8	1.8	87.2	65.4	98.4
blackscholes	919	23.0	9.0	20.6	14.0	6.6	23.0	9.0	14.0	14.0	0.0	32.1	0.0	100.0
bzip2	4294	74.5	6.0	60.3	9.3	51.0	48.9	3.4	4.7	1.9	2.9	92.2	79.9	94.4
canneal	2822	111.0	13.0	52.4	22.0	30.4	111.0	13.0	24.0	22.0	2.0	54.2	0.0	93.3
ctrace	763	296.6	2.0	160.2	11.9	148.3	324.5	2.0	9.1	2.3	6.7	94.3	80.4	95.5
dedup	2571	78.1	12.4	56.2	14.2	42.0	46.5	3.4	3.4	0.2	3.2	94.0	98.6	92.5
pbzip2	1489	457.5	7.0	128.6	14.6	114.0	70.9	3.6	3.8	2.2	1.6	97.0	84.9	98.6
pfscan	750	116.3	9.0	99.4	7.0	92.4	56.6	3.3	3.3	0.8	2.5	96.6	88.0	97.3
streamcluster	1250	1505.3	9.0	1284.2	1012.8	271.4	961.1	9.0	627.0	627.0	0.0	51.2	38.1	100.0
swarm	1636	827.6	5.0	613.4	106.3	507.1	805.1	5.0	107.1	105.1	2.0	82.5	1.1	99.6
x264	37739	659.1	10.1	148.5	18.6	129.9	582.3	10.1	18.2	16.0	2.2	87.8	13.8	98.3

Table 1: Experimental results. Data is averaged over 30 runs for each benchmark.

```
pbzip2.cpp:
void *consumer (void *q) {
       // initialization
887:   for (;;)
888:   {
889:      pthread_mutex_lock(fifo->mut);
          // check if compression is done
933:      pthread_mutex_unlock(fifo->mut);
          // do compression work
976:   }}

void queueDelete (queue *q) {
1041:   if (q == NULL)
1042:      return;
1044:   if (q->mut != NULL) {
1046:      pthread_mutex_destroy(q->mut);
1047:      delete q->mut;
1048:      q->mut = NULL; }}
```

Figure 7: Real Segmentation Fault in *pbzip2*.

sults suggest that the locally optimal traces calculated by the Tinertia algorithm are nearly optimal in a global sense, validating our first hypothesis. Traces with a coarser thread interleaving granularity significantly reduce the number of potential thread interactions one must reason about when diagnosing a bug, hence we believe these optimality results argue strongly for the Tinertia algorithm.

5.2 Contribution of Simplifying Operations

During each run of Thrille, we recorded the contribution to simplification (in terms of context switches removed) attributable to each simplifying operation. This data showed that Two-Stage Consolidate Up was a major contributor to simplification in most of the benchmarks, but both the Remove Last and Consolidate Down operations were responsible for greater than 40% of the simplification of at least one benchmark. We conclude that each of the simplifying operations performed in Tinertia contributes to the overall calculation of the simplified trace.

5.3 Pinpointing Bugs with Simplified Traces

Our third hypothesis is that the preemptions in a simplified trace are useful for pinpointing the concurrency bug. To validate this hypothesis, we examined the results of several of our experimental runs. We found that preemptions often pointed directly to the cause of the bug. We illustrate this observation for one of our benchmarks.

Figure 7 shows the real bug in the *pbzip2* benchmark. Improper synchronization can allow the main thread to execute clean up code before all worker threads have terminated. If a worker thread attempts to grab a mutex that has been destroyed and set to NULL by the main thread, the program will crash with a segmentation fault.

In the run we examined, the initial trace of the segmentation fault consisted of 79 context switches, 77 of which were preemptive and 2 of which were non-preemptive. Given this trace, Thrille generated a simplified trace which had 4 context switches, 1 of which was preemptive and 3 of which were non-preemptive. In the simplified trace, the main thread calls the queueDelete clean up method, passing fifo as the argument. The preemption occurs directly after the main thread sets q->mut to NULL in line 1048. A worker thread is then scheduled and attempts to lock the NULL mutex on line 889. This results in a segmentation fault on our test server. We note the preemptive context switch in the simplified trace occurs exactly in the buggy code.

5.4 Performance

Table 2 reports the runtime characteristics of the Thrille tool. All numbers are averaged over the 30 runs for each benchmark. Bug Type (column 2) is a description of how each bug manifests. Iters (column 3) reports the average number of iterations taken through the main loop of the Tinertia algorithm. Execs (column 4) reports the number of times Thrille re-executes the program validating an intermediate trace. Time (sec) is the average time it takes in seconds to simplify a trace. For all benchmarks, average simplification time took less than 30 minutes (often much less). The Thrille implementation is unoptimized, but given the average time spent debugging, we believe that even in its current form the benefits of debugging with a simplified trace outweigh the time costs of simplification. We also note that similar running times have been reported by other effective debugging tools in the literature [24].

Further, as traces grow longer, the Tinertia algorithm can be modified to operate on a fixed size suffix of the trace. This would place an upper bound on the number of re-executions while still allowing the programmer to reap most of the benefits of debugging with a simplified trace.

5.5 Comparison with Context Bounded Model Checking

One could argue there is no need for a debugging tool like Thrille because one could use a context bounded model checker such as CHESS [17] to find concurrency bugs. When these model checkers find a bug, they return a simplified trace by default. We now evaluate our hypothesis that the Thrille tool can still be useful by comparing the efficiency of the combination of directed random testing and simplification with basic context bounded model checking.

To compare Thrille with context bounded model checking, we implemented the basic CHESS algorithm described

Program Name	Bug Type	Iters	Execs	Time (sec)
bbuf	deadlock	2.7	298	19.4
blackscholes	deadlock	2.0	54	9.6
bzip2	segfault	2.9	64	162.5
canneal	deadlock	2.2	138	29.9
ctrace	deadlock	2.1	250	21.4
dedup	segfault	2.0	49	90.8
pbzip2	segfault	2.5	78	41.1
pfscan	segfault	2.7	109	12.2
streamcluster	segfault	2.7	2586	769.3
swarm	assert fail	2.5	1300	1651.2
x264	deadlock	2.1	187	1107.9

Table 2: `Thrille` Runtime Statistics

in [17]. We ran our CHESS implementation with a preemption bound of 2 on all of our benchmarks until a bug was found or we had explored 10,000 interleavings. The basic CHESS algorithm is incomplete in programs with data races, therefore we adopted a strategy similar to [19] and allowed CHESS to schedule at the same potential data races at which our race directed random testing implementation could schedule. This ensured our CHESS implementation could expose the same bugs as our race directed random testing implementation. We also used a fair scheduling implementation to prevent divergence on our benchmarks [18].

Table 3 shows the results from the comparison. Column 2 shows the average number of program executions to both find a bug using race directed random testing and then simplify the bug using `Thrille`. Column 3 shows the average time in seconds to find a bug and then simplify it. Column 4 shows the number of program executions before our CHESS implementation found a bug or hit the 10,000 execution cutoff. Column 5 shows the total time of the CHESS search, and column 6 reports whether our CHESS implementation found a bug before the execution cut-off.

In 2 of our benchmarks (*pbzip2* and *streamcluster*), the basic CHESS algorithm was clearly the more efficient choice for finding simplified buggy traces. On the *ctrace* benchmark, `Thrille` and CHESS were roughly tied. In the other 8 benchmarks, our CHESS implementation was either unable to discover a bug within 10,000 program executions, or was not competitive with the combination of race directed random testing and simplification. We feel these results argue that, for certain types of programs and bugs, randomized testing with simplification can be more efficient than systematic approaches like context bounded model checking.

Program Name	Avg. Execs	Avg. Time	CHESS Execs	CHESS Time	Bug?
bbuf	300	19.5	2081	183.1	yes
blackscholes	85	12.8	10000	1250.8	no
bzip2	66	166.2	10000	25025.5	no
canneal	464	79.0	10000	1789.4	no
ctrace	252	21.5	165	22.0	yes
dedup	51	92.5	10000	14447.4	no
pbzip2	83	41.9	21	3.9	yes
pfscan	110	12.3	10000	2135.2	no
streamcluster	2591	769.9	7	4.8	yes
swarm	1334	1659.8	10000	3392.2	no
x264	193	1177.0	10000	137484.6	no

Table 3: Comparison between `Thrille` and CHESS

6. OTHER RELATED WORK

Refer to the end of Section 2 for discussions of the two work most closely related to ours.

Tzoref et al. use machine learning techniques to automatically discover potentially buggy program locations in multithreaded programs [32]. Other work show that perturbing the thread scheduler in a concurrent program can increase test coverage and find bugs [8, 29, 27, 13].

Model checking is a promising technique to find concurrency bugs in programs before they manifest in the wild; however, the cause of the bug can be difficult to pinpoint in an error trace returned by a model checker. A number of research have tried to minimize an error trace and extract useful counterexamples when a bug is found [2, 11, 10].

Statistical sampling techniques can find bugs in the sequential setting [15], and extensions have been proposed to discover concurrency bugs [31]. Program slicing [33, 37] is a popular debugging approach that determines which parts of a program are relevant to a particular statement (e.g. a bug). Precise slicing for concurrent programs is undecidable in general but a number of work have investigated efficient approximate approaches for debugging [12, 20, 9].

Different trace simplification ("shrinking") techniques have been shown useful in debugging functional concurrent languages like Erlang [5].

A large body of research exists on record and replay systems for parallel software [22, 26, 6, 25]. Some of these systems make use of the specialized hardware to make record more efficient [34, 21, 16, 7]. Other systems lower record costs by probabilistically reproducing failures or recording a subset of information required to reproduce a multithreaded execution and doing offline work [23, 1].

7. CONCLUSION

Debugging a concurrent program is a time consuming and frustrating process. We believe that useful debugging techniques can be developed through the application of model checking and program analysis techniques. Our work is a small, effective step towards this goal. `Thrille` is open source, and can be downloaded at http://www.github.com/nicholasjalbert/Thrille.

8. ACKNOWLEDGMENTS

We would like to thank Shaon Barman, Sarah Bird, Andrew Waterman, and our anonymous reviewers for their valuable comments. We would also like to thank Chang-Seo Park and Christos Stergiou for contributions to the early implementation of `Thrille`. This work supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), and by NSF Grants CNS-0720906 and CCF-0747390.

9. REFERENCES

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 193–206. ACM, 2009.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *30th SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 97–105. ACM, 2003.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural

Implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[4] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ACM SIGSOFT international symposium on Software testing and analysis (ISSTA)*, pages 210–220. ACM, 2002.

[5] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *14th ACM SIGPLAN international conference on Functional programming (ICFP)*, pages 149–160. ACM, 2009.

[6] J. deok Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[7] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 85–96. ACM, 2009.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. In *ACM-ISCOPE conference on Java Grande*, page 181. ACM, 2001.

[9] D. Giffhorn and C. Hammer. Precise slicing of concurrent programs. *Automated Software Engg.*, 16(2):197–234, 2009.

[10] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, 2006.

[11] A. Groce and W. Visser. What Went Wrong: Explaining Counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[12] J. Krinke. Context-sensitive slicing of concurrent programs. In *9th European software engineering conference/11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pages 178–187. ACM, 2003.

[13] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting Atomic-Set Serializability Violations in Mulithreaded Programs through Active Randomized Testing. In *32nd International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2010.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code generation and optimization (CGO)*, page 75. IEEE, 2004.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 15–26. ACM, 2005.

[16] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *35th International Symposium on Computer Architecture (ISCA)*, pages 289–300. IEEE, 2008.

[17] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 446–455. ACM, 2007.

[18] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 362–371. ACM, 2008.

[19] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 267–280. USENIX Association, 2008.

[20] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to Java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, 2006.

[21] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32nd annual international symposium on Computer Architecture (ISCA)*, pages 284–295. IEEE, 2005.

[22] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *ACM/ONR workshop on Parallel and distributed debugging (PADD)*, pages 1–11. ACM, 1993.

[23] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *22nd symposium on Operating systems principles (SOSP)*, pages 177–192. ACM, 2009.

[24] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *7th joint meeting of the European software engineering conference/ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pages 33–42. ACM, 2009.

[25] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[26] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 258–266. ACM, 1996.

[27] K. Sen. Effective random testing of concurrent programs. In *The 22nd IEEE/ACM international conference on Automated software engineering (ASE)*, pages 323–332. ACM, 2007.

[28] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 11–21. ACM, 2008.

[29] S. D. Stoller. Testing Concurrent Java Programs using Randomized Scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4):142 – 157, 2002. RV'02, Runtime Verification 2002 (FLoC Satellite Event).

[30] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing Of long-running multithreaded programs via dynamic execution reduction. In *International Symposium on Software testing and analysis (ISSTA)*, pages 207–218. ACM, 2007.

[31] A. Thakur, R. Sen, B. Liblit, and S. Lu. Cooperative Crug Isolation. In *7th International Workshop on Dynamic Analysis (WODA)*, pages 35–41, 2009.

[32] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *International Symposium on Software testing and analysis (ISSTA)*, pages 27–38. ACM, 2007.

[33] M. Weiser. Program Slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[34] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *30th annual international symposium on Computer architecture (ISCA)*, pages 122–135. ACM, 2003.

[35] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker For Mulithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.

[36] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[37] X. Zhang, R. Gupta, and Y. Zhang. Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. In *26th International Conference on Software Engineering (ICSE)*, pages 502–511. IEEE, 2004.