An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs

Jeff Huang, Charles Zhang

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

Abstract. One of the major difficulties in debugging concurrent programs is that the programmer usually experiences frequent thread context switches, which perplexes the reasoning process. This problem can be alleviated by trace simplification techniques, which produce the same computation process but with much fewer number of context switches. The state of the art trace simplification technique takes a dynamic approach and does not scale well to large traces, hampering its practicality. We present a static trace simplification approach, SimTrace, that dramatically improves the efficiency of trace simplification through reasoning about the computation equivalence of traces offline. By constructing a dependence graph model of events, our trace simplification algorithm scales linearly to the trace size and quadratic to the number of nodes in the dependence graph. Underpinned by a trace equivalence theorem, we guarantee that the results generated by SimTrace are sound and no dynamic program re-execution is required to validate the trace equivalence. Our experiments show that SimTrace scales well to traces with more than 1M events, making it attractive to practical use.

Keywords: Trace simplification, Debugging, Concurrent program

1 Introduction

Software is becoming increasingly concurrent due to the prevalence of multicore hardware. Unfortunately, the unique non-deterministic nature of concurrency makes debugging concurrent programs notoriously difficult. For instance, according to a recent report [6], the average bug fixing time of concurrency bugs (137 days) is 2.58 times longer than that of sequential ones in the MySQL[1] project. In our experience, the challenge of debugging concurrent programs comes from two main sources. First, concurrency bugs are hard to reproduce, as they may only manifest under certain specific thread interleavings. Due to the non-deterministic thread interleavings, a concurrency bug manifested in an earlier execution may "disappear" when the programmer attempts to reproduce it [12]. Second, concurrency bugs are hard to understand. A buggy concurrent program execution often contains many thread context switches. Since most programmers are used

[1] http://www.mysql.com/

to thinking sequentially, they have to jump frequently from the context of one thread to another for reasoning about a concurrent program execution trace. These frequent "context switches" of the programmers' minds significantly impair the efficiency of debugging concurrent programs [9].

Researchers have studied the first problem for several decades [3] and numerous effective approaches [7, 11, 15, 14, 8] are proposed for reproducing concurrency bugs. For the second problem, Jalbert and Sen [9] have recently proposed a *dynamic* trace simplification technique, Tinertia, for reducing the number of thread interleavings in a buggy execution trace. From a high level perspective, Tinertia iteratively transforms an input trace that satisfies a certain property to another trace satisfying the same property but with smaller number of thread context switches. Tinertia is valuable in improving the debugging efficiency of concurrent programs as it prolongs the sequential reasoning of concurrent program executions and reduces frequent "context switches". However, since Tinertia is a dynamic approach, it faces serious efficiency problems when used in practice. To reduce every single context switch, Tinertia has to re-execute the program at least once to validate the equivalence of the transformed trace. It is very hard for Tinertia to scale to large traces as the program re-execution typically requires controlling the thread scheduler to follow the scheduling decisions in the transformed trace, which is often 5x to 100x slower than the native execution [15]. From the time complexity point of view, the total running time of Tinertia is cubic to the trace size.

In this paper, we present a *static* trace simplification technique, SimTrace, that dramatically improves the efficiency of trace simplification through the offline reasoning of the computation equivalence of traces. The key idea of SimTrace is that we can statically guarantee the trace equivalence by leveraging the dependence relations between events in the trace. By presenting a formal modeling of dependence relation, we show a theorem of trace equivalence that any rescheduling of the events in the trace respecting the dependence relation is equivalent to the given trace. The trace equivalence is not limited to any specific property but general to all properties that can be defined over the program state. Underpinned by the trace equivalence theorem, SimTrace is able to perform the trace simplification completely offline, without any dynamic re-execution to validate the intermediate simplification result, which significantly improves the efficiency of the trace simplification.

In our analysis, we first build a *dependence graph* that encodes all the dependence relations between events in the trace. The dependence graph is a direct acyclic graph in which each node in the graph represents a corresponding event or event sequence by the same thread in the trace, and each edge represents a data dependence or partial order relation between two events or event sequences. The dependence graph is sound in that it encodes a complete set of dependence relations between the events. And the trace equivalence theorem guarantees that any topological sort of the dependence graph produces an equivalent trace to the original trace.

Taking the advantage of the dependence graph, we reduce the trace simplification problem to a *graph merging problem*, of which the objective is minimizing the size of the graph. To address this problem, we developed a graph merging algorithm that performs a sequence of merging operations on the graph. Each merging operation is applied on two consecutive nodes by the same thread in the graph, and it consolidates the two nodes if a merging condition is satisfied. The merging condition is that the edge connecting the two merged nodes is the only path connecting them in the graph, which can be efficiently checked by computing the reachability relation between the two nodes.

Finally, SimTrace performs a topological sort on the reduced dependence graph and generates the simplified trace. The total running time of SimTrace is linear in the size of the trace and quadratic in the number of the nodes in the initial dependence graph. SimTrace is very efficient in practice, since the size of the initial dependence graph is often much smaller than that of the original trace. Moreover, guaranteed by the sound dependence graph model, SimTrace is completely static and does not require any re-execution of the program for validating the equivalence of the simplified trace.

For the general trace simplification problem of generating equivalent traces with *minimum* context switches, Jalbert and Sen [9] have proved that this problem is NP-hard and there is unlikely an efficient algorithm for solving it in polynomial time. Like Tinertia, SimTrace does not guarantee the globally optimal simplification but a local optimum. However, our evaluation results using a set of multithreaded programs show that SimTrace has good performance that is able to significantly reduce the context switches in the trace. For instance, for an input trace of the *Cache4j* subject with 1,225,167 events, SimTrace is able to reduce the number of context switches from 417 to 33 (with 92% reduction percentage) in 592 seconds. The overall reduction percentage of SimTrace ranges from 65% to 97% in our experiments.

Being a static analysis technique, SimTrace is complementary to Tinertia. For the sake of efficiency, our modeling of the dependence relation does not consider the runtime value dependencies between events in the trace and hence may be too strict in preventing further trace simplification. As Tinertia utilizes the runtime verification regardless of the dependence relation, it might be able to explore more simplification opportunities that are beyond the strict dependence relation. A good match between SimTrace and Tinertia for the trace simplification is to apply SimTrace as a front-end and use Tinertia as a back end. By working together, we can achieve both the trace simplification efficiency and effectiveness at the same time, i.e., to more efficiently generate a simplified trace with fewer context switches.

To sum up, the key contributions of this paper are as follows:

- We present an efficient static trace simplification technique for reducing the number of thread context switches in the trace.
- We show a theorem of trace equivalence that is general to all properties defined over the program state. This theorem provides the correctness guar-

antee of the static trace simplification without any dynamic program re-execution to validate the intermediate simplification result.

- We present a sound graph modeling of the dependence relation between events in the trace, which allows us to develop efficient graph merging algorithms for the trace simplification problem.
- We evaluate our approach on a number of multithreaded applications and the results demonstrate the efficiency and the effectiveness of our approach.

The rest of the paper is organized as follows: Section 2 describes the problem of trace simplification; Section 3 presents our algorithm; Section 4 reports our evaluation results; Section 5 discusses the related work and Section 6 concludes this paper.

2 Preliminaries

In this section, we first describe a simple but general concurrent program execution model. Based on this model, we then formally introduce the general trace simplification problem.

2.1 Concurrent program execution model

To formally present the trace simplification problem and to prove the trace equivalence theorem, we need to define a concurrent program execution model with precise execution semantics. Previous work [5, 4, 19] has described the concurrent program execution models in several different ways for different analysis purposes. To make our approach general, we define a model in a similar style to [5].

A concurrent program in our language consists of a set of concurrently executing threads $\mathbb{T} = \{t_1, t_2, ...\}$ that communicate through a global store σ. The global store consists of a set of variables $\mathbb{S} = \{s_1, s_2, ...\}$ that are shared among threads. Each thread has also its own local store π, consisting of the local variables and the program counter to the thread. We use $\sigma[s]$ to denote the value of the shared variable s on the global store. Each thread executes by performing a sequence of actions each of which operates on a single variable on the global store or the thread's own local store. Since the program counter is included in the local store, each thread is deterministic and the next action of t_i is determined by t_i's current local store π_i. The program state is defined as $\Sigma = (\sigma, \Pi)$, where σ is the global store and Π is a mapping from thread identifiers t_i to the local store π_i of each thread.

The program execution is modeled as a sequence of transitions defined over the program state Σ. Let α^k be the kth action in the global order of the program execution and Σ^{k-1} be the program state just before α^k is performed (Σ^0 is the initial state), the state transition sequence is:

$$\Sigma^0 \xrightarrow{\alpha^1} \Sigma^1 \xrightarrow{\alpha^2} \Sigma^2 \xrightarrow{\alpha^3} \ldots$$

Given a concurrent system described above, we next formally define the execution semantics of action α. Let $\Gamma(\alpha)$ denote the owner thread of action α and $var(\alpha)$ the variable accessed by α. If $var(\alpha)$ is a shared variable, we call α a *global action*, otherwise it is a *local action*. To give a precise definition, we also use some additional notations similar to [5]:

- $\sigma[s := v]$ is identical to σ except that it maps the variable s to the value v;
- $\Pi[t_i := \pi_i]$ is identical to Π except that it maps the thread identifier t_i to π_i;
- $\sigma \xrightarrow{\alpha} \sigma'$ models the effect of performing action α on the global store σ;
- $\pi \xrightarrow{\alpha} \pi'$ models the effect of performing action α on the local store π.

Local action For the case of local actions, the execution semantics of performing α is simply defined as:

$$\frac{var(\alpha) \notin \mathbb{S} \qquad \Gamma(\alpha) = t_i \qquad \pi_i \xrightarrow{\alpha} \pi_i'}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma, \Pi[t_i := \pi_i'])}$$

The program state transition above means that when a local action is performed by a thread, only the local store of that thread is changed to a new state determined by its current state. The global store and the local stores of the other threads remain the same.

Global action The colloquial of the semantics of all global actions is that when a global action is performed by a thread t_i on the shared variable s, only s and π_i are changed to new states. The states of all the other shared variables on the global store as well as the local stores of all the other threads remain the same. Let $\tau(\alpha)$ denote the computation type of the global action α. To make the execution model general to different programming languages, we consider the following types of global actions:

- READ - the thread t_i reads the value of a shared variable in the global store into its local store:

$$\frac{\tau(\alpha) = READ \quad var(\alpha) \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i'}{(\sigma, \Pi) \xrightarrow{\alpha} (\Pi[t_i := \pi_i'])}$$

- WRITE - the thread t_i assigns some value to a shared variable in the global store:

$$\frac{\tau(\alpha) = WRITE \quad var(\alpha) = s \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad \sigma[s] \xrightarrow{\alpha} \sigma'[s]}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[s := \sigma'[s]], \Pi[t_i := \pi_i'])}$$

- LOCK - the thread t_i acquires a lock l (which is also a shared variable on the global store); the pre-condition $l = 0$ means that the lock is available and the post-condition $l = i$ means that the lock l is now owned by the thread t_i:

$$\frac{\tau(\alpha) = LOCK \quad var(\alpha) = l \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad \sigma[l] = 0}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[l := i], \Pi[t_i := \pi_i'])}$$

- UNLOCK - the thread t_i releases a lock l; the pre-condition $l = i$ means l is now owned by the thread t_i and the post-condition $l = 0$ means l is avaiable:

$$\frac{\tau(\alpha) = UNLOCK \quad var(\alpha) = l \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad \sigma[l] = i}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[l := 0], \Pi[t_i := \pi_i'])}$$

- FORK - the thread t_i forks a new thread t_j. Let the shared variable s_{t_j} denote the existence of the thread t_j in the program. The pre-conditions $s_{t_j} = NA$ and $\pi_j = NA$ mean that the thread t_j is unavailable and its local store is undefined, and the post-conditions $\sigma[s_{t_j} := 1]$ and $\Pi[t_j := \pi_j^0]$ mean the thread t_j is available now and its local store is initialized to π_j^0:

$$\frac{\tau(\alpha) = FORK \quad var(\alpha) = s_{t_j} \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad s_{t_j} = NA \quad \pi_j = NA}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[s_{t_j} := 1], \Pi[t_i := \pi_i', t_j := \pi_j^0])}$$

- JOIN - the thread t_i joins the termination of the thread t_j; the pre-condition $s_{t_j} = 0$ means that the thread t_j has already terminated:

$$\frac{\tau(\alpha) = FORK \quad var(\alpha) = s_{t_j} \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad s_{t_j} = 0}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma, \Pi[t_i := \pi_i')}$$

- START - the first action in the action sequence of the thread t_i. This is a *dummy* action indicating that the thread t_i is ready to run. This action does not change any program state and it immediately follows the FORK action that forked the thread t_i:

$$\frac{\tau(\alpha) = START \quad var(\alpha) = s_{t_i} \quad \Gamma(\alpha) = t_i}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma, \Pi)}$$

- EXIT - the last action in the action sequence of the thread t_i, indicating that t_i has terminated. The value of the shared variable s_{t_i} is set to 0 after this action:

$$\frac{\tau(\alpha) = EXIT \quad var(\alpha) = s_{t_i} \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad s_{t_i} = 1}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[s_{t_i} := 0], \Pi[t_i := NA])}$$

- SIGNAL - the thread t_i sets the value of a conditional variable c to 1:

$$\frac{\tau(\alpha) = SIGNAL \quad var(\alpha) = c \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i'}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[c := 1], \Pi[t_i := \pi_i'])}$$

- WAIT - the standard semantics of a $wait(c, l)$ action contains a sequence of three actions UNLOCK-WAIT-LOCK: the thread t_i first releases the lock l it is currently holding, then it waits for a conditional variable c to become 1 and resets it back to 0 after c becomes 1, and finally it re-acquires lock l. The following execution semantics model the second action:

$$\frac{\tau(\alpha) = WAIT \quad var(\alpha) = c \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi_i' \quad c = 1}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[c := 0], \Pi[t_i := \pi_i'])}$$

It is important notice that, for any action, either a global action or a local action, it only *operates on a single variable*. This is also true for synchronization actions, though they are only enabled when certain pre-conditions are met.

The execution semantics defined above conform to a general concurrent execution model with deterministic input. For concise presentation, dynamic shared variable creation and re-entrant locks are not explicitly supported by the semantics, but they can be modeled within the semantics in a straightforward way.

2.2 General trace simplification problem

Definition 1. *A* **trace** *is the action sequence $\langle \alpha^k \rangle$ of a program execution.*

Definition 2. *A* **context switch** *occurs when two consecutive actions in the trace are performed by different threads.*

Recall that $\Gamma(\alpha)$ denotes the owner thread of action α. Let δ denote a trace containing N actions and $\delta[k]$ the kth action in δ, and let $CS(\delta)$ denote the number of context switches in δ, we have $CS(\delta) = \Sigma_{k=1}^{N-1} u_k$ where u_k is a binary variable s.t. $u_k = 1$ if $\Gamma(\delta[k]) \neq \Gamma(\delta[k+1])$ and $u_k = 0$ otherwise.

Definition 3. *Two traces are* **equivalent** *if they drive the same initial program state to the same final program state.*

Given a trace as the input, the general trace simplification problem is to produce an output trace that is *equivalent* to the input trace and has minimum number of context switches among all the equivalent traces. To state more formally, suppose an input trace δ drives the program state to Σ^N, the general trace simplification problem is: *given δ, output a δ' s.t. $\Sigma^N = \Sigma'^N$ and $CS(\delta')$ is minimized.* Notice that the program state here is not limited to any local store or the global store but includes both the global store and the local stores of all the threads. In other words, the trace simplification problem defined above is general to all properties defined over the program state.

The basic idea for reducing the context switches in a trace is to reschedule the actions in the trace such that more actions by the same thread are placed next to each other. A naive approach is to exhaustively generate all permutations of the events in the trace and pick an equivalent one with the smallest number of context switches. However, this naive approach requires checking N! permutations which is highly inefficient. A better approach is to repeatedly move the interleaving actions to some non-interleaving positions and then consolidate the neighboring actions by the same thread. However, there are two major challenges in this approach. First, how to ensure the rescheduled trace is feasible and also equivalent to the input trace? Second, how to make sure the output trace is optimal, i.e., has the minimum number of context switches among all the equivalent traces?

3 SimTrace: Efficient Static Trace Simplification

We address the trace simplification problem by leveraging the dependence relationship between actions in the trace. For the first challenge, we show that the trace equivalence can be guaranteed by respecting the dependence relation during the rescheduling process. For the second challenge, since Jalbert and Sen [9] have proved it is *NP-hard*, we present an efficient algorithm, SimTrace, that guarantees to generate a locally optimal solution. In this section, we first describe our modeling of the dependence relation. Based on the modeling, we describe a theorem of trace equivalence and offer a detailed proof. After that, we present the full SimTrace algorithm.

3.1 Modeling of the dependence relation

Previous work has proposed many causal models [16, 2, 10, 13, 21] that characterize the dependence relationship between actions in the trace. Among them, most models are developed for checking concurrency properties such as data race and atomicity violations, and they are tailored for the specific property. Different from these models, as we are dealing with all properties over the program state, we have to consider a general model that works for all properties.

Moreover, to support efficient trace simplification, the model should be as simple as possible. Although using a more precise dependence relation model, such as the maximal causal model [16] or the guarded independence model [21], may give us more freedom to simplify the trace (which can further reduce the context switches), such a model is often very expensive to construct in practice, because it requires to track all the value dependencies between events along the program branches and all the correlated control flow decisions in the program execution. We thus use a strict model defined as follows:

Definition 4. *The* **dependence relation** *($\rightarrow$) for a trace δ is the smallest transitive closure over the actions in δ, such that $a_i \rightarrow a_j$ holds whenever a_i occurs before a_j and one of the following holds:*

- **Local dependence relation**
  - *Program order* - a_i immediately precedes α_j in the same thread.
- **Remote dependence relation**
  - *Synchronization order* - a_i and a_j are consecutive synchronization actions by different threads on the same shared variable. There are four types of synchronization orders:
    - UNLOCK→LOCK: a_i is the UNLOCK action that releases the lock acquired by the LOCK action a_j;
    - FORK→START: a_i is the FORK action that forks the thread whose START action is a_j;
    - EXIT→JOIN: a_i is the EXIT action of a thread that the JOIN action a_j joins;
    - NOTIFY→WAIT: a_i is the NOTIFY action that sets the conditional variable the WAIT action a_j waits for;

- *Conflicting order* - a_i and a_j are consecutive conflicting actions by different threads on the shared variable. There are three types of conflicting orders:
  * WRITE→READ: a_i is a WRITE action and a_j is a READ action;
  * READ→WRITE: a_i is a READ action and a_j is a WRITE action;
  * WRITE→WRITE: both a_i and a_j are WRITE actions.

Given a dependence relation $a_i \rightarrow a_j$, If a_i and a_j are from different threads, we say a_i has a remote outgoing dependence to a_j, and similarly, a_j has a remote incoming dependence to a_i; Otherwise, we say a_i has a local outgoing dependence to a_j and a_j has a local incoming dependence to a_i.

It is important to notice that the remote dependence relations in our model are all between actions accessing the same shared variable. Therefore, context switches between threads accessing different variables in the trace are allowed to be reduced in our model. Nevertheless, also note that the remote dependence in our modeling includes both the synchronization order and the conflicting order, which may actually be unnecessary if we consider the value dependencies between events. For example, two writes to the same variable with the same value can be permuted without affecting the correctness of the trace. The main purpose for using this strict model is that we want to efficiently and safely guarantee the trace equivalence, towards which we must ensure that every action in the simplified trace reads or writes the same value as its corresponding action in the original trace.

3.2 A theorem of trace equivalence

Based on our model of the dependence relation in Section 3.1, we have the following theorem of trace equivalence:

Theorem 1. *Any rescheduling of the actions in a trace respecting the dependence relation defined in Definition 4 generates an equivalent trace.*

Proof. The main insight of the proof is that, by respecting the order defined by the dependence relation, every action in the rescheduled trace reads or writes the same value on the program state as its corresponding action in the input trace, and hence the rescheduled trace drives the program to the same final state as that of the input trace. Let δ denote the input trace with size N and δ' an arbitrary rescheduling of δ respecting the dependence relation, and suppose δ and δ' drive the program state from the same initial state Σ^0 and Σ'^0 to Σ^N and Σ'^N, respectively. Our goal is to prove $\Sigma'^N = \Sigma^N$.

Let us say two actions are *equal iff* they perform the same operation on the same variable and also read and write the same value. The core of the proof is to prove the following lemma:

Lemma 1. *For any action α' in δ', suppose it is the nth action of thread t_i, then α' is equal to the nth action of t_i in δ.*

If Lemma 1 holds, we can easily prove Theorem 1 by applying it to the last actions that write to each variable in both δ and δ'. To prove Lemma 1, we first define a notion of *version number* and show two lemmas related to it:

Definition 5. *Every variable is associated with a* **version number** *such that it is (1) initialized to be 0 and (2) incremented by 1 when the variable is written by an action.*

Lemma 2. *For any action α' in δ', suppose it is the kth action that writes to a variable s, then α' is also the kth action that writes to s in δ.*

Proof. To prove Lemma 2, we only need to make sure the order of write actions on each variable is unchanged during the rescheduling of the trace from δ to δ'. This is easy to prove as our modeling of the dependence relation includes all synchronization orders and the WRITE→WRITE orders on the same variable.

Lemma 3. *For any action α' in δ', suppose it reads the variable s with version number p, then α' also reads s with the same version number p in δ.*

Proof. Similar to the proof of Lemma 2, since our model of the dependence relation includes all the synchronization orders and the WRITE→READ and READ→WRITE orders on the same variable, we guarantee every READ action in the rescheduled trace reads the value written by the same WRITE action as that in the original trace.

Let $\sigma[s]^p$ denote the value of variable s with version number p, we next prove Lemma 1 by deduction on the version number of each variable:

Proof. Consider the jth actions performed by t_i, denoted by $\alpha_{i:j}$ and $\alpha'_{i:j}$ in δ and δ' respectively. To prove $\alpha'_{i:j}$ is equal to $\alpha_{i:j}$, we need to satisfy two conditions. First, their actions should be the same, i.e., they perform the same operation on the same variable. Second, suppose they both operate on the variable s (which should be true if the first condition holds), the values of s before $\alpha'_{i:j}$ is performed in δ' should be the same as that in δ before $\alpha_{i:j}$ is performed. Let $\pi_{i:j}$ and $\pi'_{i:j}$ denote the local store of t_i after $\alpha_{i:j}$ is performed in δ and after $\alpha'_{i:j}$ is performed in δ', respectively. For the first condition, since the execution semantics determine that the next action of any thread is determined by that thread's current local store, we need to ensure (I) $\pi'_{i:j-1} = \pi_{i:j-1}$. For the second condition, suppose $\alpha_{i:j}$ and $\alpha'_{i:j}$ operate on s with version number p and p', respectively, we need to ensure (II) $\sigma'[s]^{p'} = \sigma[s]^p$.

Let's first assume Condition I holds, we prove $p' = p$ in Condition II. If $\alpha'_{i:j}$ writes to s, i.e., $\alpha'_{i:j}$ is the p'^{th} action that writes to s, by Lemma 2, we can get that the corresponding action of $\alpha'_{i:j}$ in δ is also the p'^{th} action that writes to s. As Condition I holds, we know $\alpha_{i:j}$ is the corresponding action of $\alpha_{i:j}$ in δ'. Since $\alpha_{i:j}$ operates on s with version number p in our assumption, we get $p' = p$. Otherwise if $\alpha'_{i:j}$ reads on s, by Lemma 3, we can get that $\alpha'_{i:j}$'s corresponding action in δ also reads s with the same version number, and similarly, we get $p' = p$.

We next prove both Condition I and Condition II hold. For condition I, suppose $\alpha_{i:j-1}$ and $\alpha'_{i:j-1}$ operate on the variable $s1$ with version number $p1$. To satisfy condition I, we need again to make sure (Ia) $\pi'_{i:j-2} = \pi_{i:j-2}$ and (Ib) $\sigma'[s1]^{p1} = \sigma[s1]^{p1}$. For condition II, let $\alpha_{i1:j1}$ and $\alpha'_{i1':j1'}$ denote the actions that write $\sigma[s]^p$ and $\sigma'[s]^p$, respectively. Since the current value of a variable is determined by the action that last writes to it, to satisfy condition II, we need to make sure $\alpha'_{i1':j1'}$ is equal to $\alpha_{i1:j1}$, which again requires (IIa) $\pi'_{i1':j1'-1} = \pi_{i1:j1-1}$ and (IIb) $\sigma'[s]^{p-1} = \sigma[s]^{p-1}$. If we apply this reasoning logic deductively for all threads, we will finally reach the base condition (i) $\forall t_i \in \mathbb{T}, \pi'_{i:0} = \pi_{i:0}$ and (ii) $\forall s \in \mathbb{S}, \sigma'[s]^0 = \sigma[s]^0$, which are satisfied by the equivalence of the initial program states $\Sigma'^0 = \Sigma^0$. Hence, Lemma 2 is proved.

Theorem 1 forms the basis of static trace simplification as it guarantees every rescheduling of the actions in the trace that respects the dependence relation produces a valid simplification result, without the need of any runtime verification. In other words, as long as we do not violate the order defined by the dependence relation, we can safely reschedule the events in the trace without worrying about correctness of the final result.

3.3 SimTrace algorithm

Our algorithm starts by constructing from the input trace a dependence graph (see Definition 6), which encodes all the actions in the trace as well as the dependence relations between the actions. We then simplify the dependence graph by ordinally performing a "merging" operation on two consecutive nodes by the same thread in the graph. When the dependence graph cannot be further simplified, our approach applies a simple topological sort on the graph to produce the final simplified trace.

Definition 6. *A* **dependence graph***, built upon a trace, is a directed acyclic graph in which each node in the graph corresponds to a sequence of consecutive actions by the same thread started by a unique action that has remote incoming dependence. There are two types of edges in the dependence graph: the local edges (denoted by dashed arrows ⇢) and the remote edges (denoted by solid arrows →). The local edges connect neighboring nodes by the same thread, while the remote edges connect nodes by different threads meaning that there are dependence relations from some actions in one node to some actions in the other node.*

Note that the dependence graph is directed acyclic graph. Otherwise it indicates there are cyclic dependences between events in the trace, which is impossible according to dependence relation model. We next describe our algorithms for constructing and simplifying the dependence graph in detail.

Dependence Graph Construction Algorithm 1 shows our algorithm for constructing the dependence graph. Given an input trace, we first conduct a linear scan

Algorithm 1 ConstructDependenceGraph(δ)

```
1: input: δ (a trace)
2: output: graph (the dependence graph built from δ)
3: map_t2n ← empty map from a thread identifier to its current graph node
4: t_old ← null
5: for i ← 0 to |δ|-1 do
6:    t_cur ← the thread identifier of the action δ[i]
7:    node_cur ← map_t2n(t_cur)
8:    if node_cur is null then
9:       node_cur ← new node(δ[i])
10:      map_t2n(t_cur) ← node_cur
11:      add node node_cur to graph
12:   else
13:      if δ[i] has remote incoming dependence and t_cur ≠ t_old then
14:         node_old ← node_cur
15:         node_cur ← new node(δ[i])
16:         add node node_cur to graph
17:         add local edge node_old ⇢ node_cur to graph
18:         for each action a with remote outgoing dependence to δ[i] do
19:            node_a ← the node to which a belongs
20:            add remote edge node_a → node_cur to graph
21:         end for
22:      else
23:         add action δ[i] to node_cur
24:      end if
25:   end if
26:   t_old ← t_cur
27: end for
```

of all the actions in the trace to build the smallest dependence relation between actions, according to our model in Section 3.1. We then visit each action in their appearing order in the trace once to construct the dependence graph according to Definition 6. Our construction of the dependence graph leverages the observation that most of the dependence relations in the trace are local dependencies within the same thread, while the number of remote dependence relations are comparatively much smaller. We can hence greatly reduce the size of the initial dependence graph by shrinking consecutive actions with only local dependence between them into a single node. The running time of Algorithm 1 is linear to the trace size.

Note that, in our dependence graph construction process, each node in the initial dependence graph has exactly two incoming edges except the root node: a local incoming edge and a remote incoming edge. The number of edges in the graph is thus less than twice the number of nodes in the graph. Moreover, since each node in the dependence graph may represent a sequence of actions in the trace, the number of nodes in the graph is much smaller than the original trace

size. As a result, performing a topological sort on the dependence graph is much more efficient than that on the original trace.

Simplifying Dependence Graph Following Theorem 1, it is easy to see that any topological sort of the initial dependence graph produces a correct answer to our problem, i.e., generates an equivalent trace to the input trace. However, to make the resultant trace as simple as possible, i.e., to minimize the context switches, we have to wisely choose the next node in each sorting step during the topological sort, which is a difficult problem with no existing solution or even good approximation algorithm, to our best knowledge.

We formulate this problem as an optimization problem on the number of nodes in the dependence graph and use a graph merging algorithm to compute a locally optimal solution to it. Before describing the formulation, let us first introduce a dual notion of context switch:

Definition 7. *A* **context continuation** *occurs when two consecutive actions in the trace are performed by the same thread.*

Let $CC(\delta)$ denote the number of context continuations in a trace δ, we have the following lemma:

Lemma 4. *Minimizing* $CS(\delta)$ *is equivalent to maximizing* $CC(\delta)$.

Proof. Traversing the trace once, it is easy to see that for each action, either $CS(\delta)$ or $CC(\delta)$ is incremented. Thus, $CS(\delta) + CC(\delta) = N - 1$. Hence, $CS(\delta)$ is minimized when $CC(\delta)$ is maximized.

Therefore, our goal becomes to maximize the number of context continuations in the simplified trace. Now let us consider the action sequence represented by each node in the dependence graph. Since all actions in the same action sequence are performed by the same thread, their number of context continuations are already optimized. The remaining possible context continuations can only come from actions that are in different action sequences. Mapping this back to the dependence graph and because nodes representing action sequences by the same thread are connected by local edges, it is easy to show the following lemma:

Lemma 5. *Minimizing* $CS(\delta)$ *is equivalent to maximizing the number of context continuations contributed by local edges in the dependence graph.*

Consider a local edge in the graph, if the action sequences represented by the two nodes connected by this local edge are consolidated together, it will contribute one context continuation. Let us call a *merging* operation as the consolidating of two nodes connected by a local edge in the dependence graph. As each merging operation eliminates a local edge and correspondingly reduces one node in the dependence graph, it is easy for us to get the following theorem:

Theorem 2. *Minimizing* $CS(\delta)$ *is equivalent to minimizing the number of nodes in the dependence graph.*

Following Theorem 2, to produce an optimal solution, our objective is performing as many merging operation as possible to minimize the number of nodes in the dependence graph. However, recall that the dependence relation between actions in the trace must be respected. Therefore, we cannot arbitrarily perform the merging operation without satistifying a certain pre-condition: the merging condition is that the to-be-merged two nodes are connected by the local edge only. Otherwise, the resultant graph after the merging operation would become cyclic that violates the definition of dependence graph. Mapping this back to the semantics of the dependence relation, the merging condition simply requires that there should not exist another dependent action in the trace that interleaves the two action sequences represented by the to-be-merged two nodes in the dependence graph. Checking the merging condition is simple because it only requires testing the reachability relation between the two merged nodes, which costs a linear running time in the number of nodes in the dependence graph.

Therefore, our dependence graph simplification algorithm (Algorithm 2) traverses each local edge in the dependence graph, and performs the merging operation if the merging condition is satisfied. This algorithm evaluates each local edge in the initial dependence graph once and each evaluation computes the reachability relation between two nodes once. The worst case time complexity is thus quadratic in the number of nodes in the initial dependence graph.

Algorithm 2 SimplifyDependenceGraph($graph$)

1: **input**: $graph$ (the dependence graph)
2: **output**: $graph'$ (the simplified dependence graph)
3: $graph' \leftarrow graph$
4: **for** each local edge $node_a \rightarrow node_b$ **do**
5:    **if** $node_b$ is not reachable from $node_a$ except from the local edge **then**
6:       $merge(node_a, node_b, graph')$
7:    **end if**
8: **end for**

Notice that in our merging algorithm, the evaluation order of the local edges may affect the simplification result. To illustrate this problem, let us the (incomplete) dependence graph in Figure 1 as an example. The graph contains 6 nodes, 3 local edges, and 4 remote edges: $a_1 \dashrightarrow a_2$, $b_1 \dashrightarrow b_2$, $c_1 \dashrightarrow c_2$, $a_1 \rightarrow b_2$, $c_1 \rightarrow b_2$, $b_1 \rightarrow a_2$ and $b_1 \rightarrow c_2$. If b_1 and b_2 are merged first, as shown in Figure 1 (a), it would produce the trace <a_1-c_1-b_1-b_2-c_2-a_2> that contains 4 context switches. However, the optimal solution is to merge a_1 and a_2, and c_1 and c_2, which produces the trace <b_1-a_1-a_2-c_1-c_2-b_2> that contains only 3 context switches. In fact, this problem is NP-hard (proved by Jalbert and Sen [9]), and there does not seem to exist an efficient algorithm for generating an optimal solution. Our algorithm thus picks an arbitrary order or a random order for evaluating the lo-

cal edges. Though it does not guarantee to produce a global optimum, it always produces a local optimum specific to the chosen evaluation order.

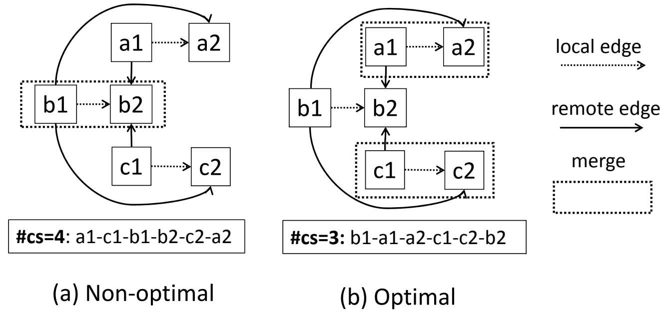

Fig. 1. A greedy merge may produce non-optimal result (a); while the problem of producing optimal result (b) is NP-hard.

4 Implementation and Experiments

We have implemented SimTrace as a prototype tool on top of our LEAP [8] record and replay framework for multithreaded Java programs. From the user's perspective, our tool consists of three phases. It first obtains a trace of a buggy concurrent Java program execution, which contains all the shared memory reads and writes as well as synchronization operations performed by each thread in the program. Then our tool applies the SimTrace algorithm on the trace and produces a simplified trace. In the third phase, it uses a replay engine to re-execute the program according to the scheduling decisions in the simplified trace. Our replayer is transparent to the programmers such that they can deterministically investigate the simplified buggy trace in a normal debugging environment.

Our evaluation subjects include eight widely used multithreaded Java benchmarks. Each subject has one or more known concurrency bugs. *Philosopher* is a simulation of the classical dinning philosophers problem, *Bubble* is a buggy multithreaded version of the bubble sort algorithm, *TSP* is a multithreaded implementation of a parallel branch and bound algorithm for the travelling salesman problem, *StringBuffer* is an open library from Suns JDK 1.4.2, *Cache4j* is a thread-safe implementation of cache for Java objects with an atomicity violation bug, *Weblech* is a multi-threaded web site download and mirror tool, *SpecJMS* is SPEC's benchmark for evaluating the performance of enterprise message-oriented middleware servers based on JMS, and *Jigsaw* is W3C's leading-edge web server platform. Similar to Tinertia [9], we use the random testing approach to generate the initial buggy trace for each subject. To remove the non-determinism related to random numbers, we fix the seed of random numbers to a constant in all the subjects. All experiments were conducted on a

HP EliteBook running Windows 7 with 2.53GHz Intel Core 2 Duo processor and 4GB memory. For others to verify our experimental results, we put our implementation and all the evaluation subjects publicly available at `http://www.cse.ust.hk/prism/simtrace`.

Program	LOC	Thread	SV	Trace Size	Time	Old Ctxt	New Ctxt	Reduction
Philosopher	81	6	1	131	6ms	51	18	65%
Bubble	417	26	25	1,493	23ms	454	163	71%
Elevator	514	4	13	2104	8ms	80	14	83%
TSP	709	5	234	636,499	149s	9272	1337	86%
Cache4j	3,897	4	5	1,225,167	592s	417	33	92%
Weblench	35,175	3	26	11,630	57ms	156	24	85%
OpenJMS	154,563	32	365	376,187	38s	96,643	11,402	88%
Jigsaw	381,348	10	126	19,074	130ms	2396	65	97%

Table 1. Experimental results

Table 4 shows the experimental results. The first five columns show the statistics of the test cases, including the program name, the size of the program in lines of source code, the number of threads, the number of real shared memory locations that contain both read and write accesses from different threads in the given trace, and the length of the trace. The next four columns shows the statistics of our trace simplification algorithm, including the running time of our analysis, the number of context switches in the original trace, the number of context switches in the simplified trace and the percentage of reduction due to our simplification. The results show that our approach is promising in terms of both the trace simplification efficiency and the effectiveness. For the eight subjects, our approach is able to reduce the number of context switches in the trace by 65% to 97%. This reduction percentage is close to that of Tinertia, the reduction percentage of which ranges from to 32.1% to 97.0% in their experiments. More importantly, our approach is able to scale to much larger traces compared to that of Tinertia. For a trace with only 1505 events (which is the largest trace reported by Tinertia in their experiments), Tinertia requires a total of 769.3s to finish the simplification, while our approach can analyze a trace (the *Cache4j* subject) with more than 1M events within 600s. For a trace (the *Bubble* subject) with 1,493 events, our approach requires only 23ms to simplify it. Although a direct comparison between Tinertia and our approach is not applicable as we two approaches are implemented for different program languages (Tinertia is implemented for C/C++ programs) and we have different evaluation subjects, we believe the statistical data provides some evidence demonstrating the value of our approach compared to the state of the art.

5 Related Work

From the perspective of the trace theory, our dependence graph modeling of traces is an instantiation of Mazurkiewicz traces [1], which models a class of equivalent traces with respect to an alphabet and a dependence relation over the alphabet. Different from the equivalence axiom of Mazurkiewicz traces [1], our trace equivalence theorem is built on top of the computation equivalence of program state transitions and relies on a concrete model of the program execution semantics.

Our model of the dependence relation is closely related to, but different from the classical happens-before model [10] and various other causal models [2, 13, 16, 21], which do not enforce the conflicting orders used in our model, but rely on the value dependencies between the events to obtain a more precise causal model than ours. For example, Wang et al. [21] introduced a notion of guarded independence to enable precisely checking of atomicity violations, Chen et al. [2] proposed a notion of sliced causality that significantly reduces the size of the computed causality relation by slicing the causal dependences between the events, and Şerbănuţă et al. [16] proposed a maximal causal model that comprises in theory all consistent executions that can be derived from a given trace. Although using a more precise dependence relation model may further reduce the context switches in the trace, as explained in Section 2.1, our strict modeling enables us to more efficiently perform the trace simplification.

Besides the problem of reducing the context switches in the trace, another important problem for the trace simplification is to reduce the size of trace. Along this direction, Tallam et al. [18] proposed an execution reduction (ER) system for removing the events that are irrelevant to the bug property in the trace. Similar to SimTrace, it also builds a dependence graph based on the dynamic dependencies between events in the trace. The main difference is that the ER system in [18] lacks a formal modeling of the trace elements. Without formalizing the semantics of each event in the trace, it is not obvious to understand the characteristics and the correctness of the resultant dependence graph.

To help with automatic concurrent program verification, Sinha and Wang [17] recently proposed a novel approach that separates intra- and inter-thread reasoning by symbolically encoding the shared memory accesses and composing the intra-thread summaries based on a sequential consistent memory model. Since this approach avoids redundant bi-modal reasoning, it has been shown to greatly improve the efficiency of trace analysis over previous approaches [20]. A promising way to go with this approach is to investigate how to apply it for helping programmers understand concurrency bugs.

6 Conclusion

We present an efficient static trace simplification technique for reducing the context switches in a concurrent program execution trace. By constructing a dependence graph model of events, our algorithm scales linearly to the trace size and quadratic to the number of nodes in the dependence graph. Moreover,

underpinned by a trace equivalence theorem, our approach guarantees to generate an equivalent simplified trace without any dynamic program re-execution, making it attractive to practical use.

References

1. Mazurkiewicz A. Trace theory. *Advances in Petri Nets*, 1987.
2. Feng Chen and Grigore Roşu. Parametric and sliced causality. In *CAV*, 2007.
3. Ronald Curtis and Larry D. Wittie. Bugnet: A debugging system for parallel programming environments. In *ICDCS*, 1982.
4. Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, 2009.
5. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
6. Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. *DSN*, 2010.
7. Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.
8. Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multiprocessor replay of concurrent Java programs. In *FSE*, 2010.
9. Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *FSE*, 2010.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.
11. Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently. In *ISCA*, 2008.
12. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
13. Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.
14. Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.
15. Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.
16. Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. Maximal causal models for sequentially consistent multithreaded systems. Technical report, University of Illinois, 2010.
17. Nishant Sinha and Chao Wang. Staged concurrent program analysis. In *FSE*, 2010.
18. Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA*, 2007.
19. Kahlon Vineet and Wang Chao. Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.
20. Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *ESEC/SIGSOFT FSE*, 2009.

21. Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.