Meta-analysis for Atomicity Violations under Nested Locking

Azadeh Farzan[1], P. Madhusudan[2], and Francesco Sorrentino[2]

[1] University of Toronto
[2] Univ. of Illinois at Urbana-Champaign

Abstract. We study the problem of determining, given a run of a concurrent program, whether there is any alternate execution of it that violates atomicity, where atomicity is defined using marked blocks of local runs. We show that if a concurrent program adopts *nested locking*, the problem of predicting atomicity violations is efficiently solvable, without exploring all interleavings. In particular, for the case of atomicity violations involving only two threads and a single variable, which covers many of the atomicity errors reported in bug databases, we exhibit efficient algorithms that work in time that is *linear* in the length of the runs, and quadratic in the number of threads. Moreover, we report on an implementation of this algorithm, and show experimentally that it scales well for benchmark concurrent programs and is effective in predicting a large number of atomicity violations even from a single run.

1 Introduction

The multicore revolution is transforming computer science. The fact that individual processors may not get any faster, and the only way software can gain speed is to exploit concurrent executions on multiple processor cores, has created a need to update all disciplines within computer science (algorithms, data structures, programming languages, software engineering, architecture, operating systems, testing, verification, etc.) to adapt themselves to concurrency.

The motivation of this paper is to study problems in testing concurrent programs. Testing, which is the primary technique used in the industry to assure correctness of software, is fundamentally challenged for concurrent programs because of the *interleaving explosion problem*. Given a concurrent program P and a *single* test input i to it, there are a multitude of interleaved executions on i. This grows exponentially with the number of cores, making a systematic exploration of all executions on the test infeasible.

One way to tackle this problem is to choose (wisely) a subset of interleaved executions to test. The CHESS project at Microsoft research is one such tool, which systematically explores all interleavings that involve only k context-switches (for a small fixed k), banking on the intuition that most errors manifest themselves within a few context switches. IBM's ConTest tool also explores schedules that are more likely to have data races, deadlocks, etc.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 248–262, 2009.

In the line of work we pursue, we have chosen the class of executions that *violate atomicity* as a candidate for selecting schedules. A programmer writing a procedure often wants uninterfered access to certain shared data that will enable him/her to reason about the procedure locally. The programmer often puts together concurrency control mechanisms to ensure atomicity, often by taking locks on the data accessed. This is however extremely error-prone: errors occur if not all the required locks for the data are acquired, non-uniform ordering of locking can cause deadlocks, and naive ways of locking can inhibit concurrency, which force programmers to invent intricate ways to achieve concurrency and correctness at the same time. Recent studies of concurrency errors [11] show that a majority of errors (69%) are atomicity violations. This motivates our choice in selecting executions that violate atomicity as the criterion for choosing interleavings to execute.

In this paper, we tackle a key problem towards this goal: given an execution of a concurrent program on a test input, say ρ, we pose the meta-analysis problem of efficiently checking whether there is an alternate scheduling of the events in ρ that violates atomicity.

Notice that this is much more complex than *monitoring* problem, which checks, given a particular execution ρ, whether ρ itself violates atomicity. We examined the monitoring problem in work reported in CAV last year [3], where we showed that it is efficiently solvable using *streaming* algorithms that take space independent of the length of the executions. The recent tool Velodrome [6] also considers only the simpler monitoring problem, and not the meta-analysis problem we consider in this paper.

In recent work [4], we have studied the meta-analysis problem for atomicity, and shown that when all synchronization actions in the execution are *ignored*, efficient algorithms that work in time *linear* in n (where n is the length of the execution) are feasible. However, we also showed that if the locking synchronization between threads is taken into account, an algorithm that is linear in n is *unlikely* to exist using complexity-theoretic arguments.

The main result of this paper is to show that when the program uses only *nested locking* (i.e. when threads release locks in the reverse order of how they acquired them), we can build algorithms that effectively analyze *all* interleavings for basic atomicity violations in time that is *linear* in the length of the execution.

Our goal in this work is also to find an efficient scalable practical algorithm for analyzing executions for atomicity violations. Consequently, we study atomicity violations caused by two threads accessing one variable only. More precisely, we define atomicity using *serializability*. We look for *minimal serializability* violations which are those caused by two threads and one variable only; i.e. we look for threads T and T', where there are two events e_1 and e_2 that access a variable v in a single transaction of T, and there is an action in thread T' that happens in between the events e_1 and e_2, and is conflicting with both e_1 and e_2.

In our experience in studying concurrency bugs, we have found that many atomicity violations are caused due to such patterns of interaction. The recent study of concurrency bugs by Lu et al. [11] in fact found that 96% of concurrency

errors involved only two threads, and 66% of (non-deadlock) concurrency errors involved only one variable. The restriction of atomicity errors involving only two threads and one variable makes our algorithm feasible in practice.

Our main result is to show that given a run ρ of length n where threads synchronize using nested locking, we can predict whether any of the many interleavings (which can be exponential in n) has a minimal serializability violation in time linear in n, linear in the number of global variables, and quadratic in the number of threads. Our algorithm is *compositional*: it works first locally on each individual thread, extracting information called *profiles* that depend only on the set of locks and variables, and is independent of n. Then, in a second phase, we combine the profiles obtained from the different threads to check whether there is an interleaved run that violates atomicity.

Our algorithm is derived by reducing the meta-analysis of minimal atomicity violations to the problem of pairwise reachability of two threads. We then use a beautiful result by Kahlon et al. [9] to solve the latter problem for threads with nested locking compositionally, using lock-sets and acquisition histories.

We have also implemented our meta-analysis algorithm for atomicity in a tool.[1] By transforming concurrent programs so that they can be monitored, we extract concurrent executions on test inputs, and use our algorithm to find atomicity violations. In these examples, the length of the executions are extremely long (some have about 11 million events), and any algorithm that runs even in time quadratic in n would not scale. We show, through experiments, that our linear algorithm scales well to these long executions, and accurately predicts a large number of atomicity violations even from a single run.

Related Work. Apart from the related work discussed above, *atomicity violations based on serializability* have been suggested to be effective in finding concurrency bugs in many works [5,7,17,18,19]. Lipton transactions have been used to find atomicity violations in programs [5,7,8,10]. In [2], we proposed a slightly different notion of atomicity called *causal atomicity*; the violations we find in this paper can also be seen as causal atomicity violations.

The run-time *monitoring* for atomicity violations is well-studied [3,6]. Note that here the problem is to simply observe a run and check whether that particular run (and only that run) is atomic. The work in [13] defines *access interleaving invariants*, which are certain patterns of access interactions on variables, learns the intended specifications using tests, and monitors runs to find errors. A variant of dynamic two-phase locking algorithm [12] for detection of serializability violations is used in the atomicity monitoring tool developed in [19].

Turning to predictive analysis, there are two main streams of work that are relevant. In papers [17,18], Wang and Stoller study the prediction of runs that violate serializability from a single run. Under the assumptions of deadlock-freedom and nested locking, they show precise algorithms that can handle serializability violations involving *at most two transactions* (not threads). They also give heuristic incomplete algorithms for checking arbitrary runs. In contrast, we focus on *minimal* serializability here, and check for violations involving

[1] All experimental data can be found at `http://www.cs.uiuc.edu/~madhu/cav09/`

two *threads* that could involve a large number of transactions. The approach in [17] uses a structure which grows quadratically with the size of the observed run, and therefore has limited scalability. Our algorithm uses space independent of the size of the observed run, and time linear in the observed run, and scales well. Predicting alternate executions from a single run are also studied in a series of papers by Rosu et al. [1,14]. While these tools can also predict runs that can violate atomicity, their prediction model is tuned towards *explicitly* generating alternate runs, which can then be subject to atomicity analysis. In sharp contrast, the results we present here search the exponential space of alternate interleavings efficiently, without enumerating them. However, the accuracy and feasibility of prediction in the above papers are better as the algorithm is aware of the static structure of the programs and other control dependencies.

2 Modeling Executions of Concurrent Programs

Notation. For any alphabet A, and any word $w \in A^\star$, let $w[i]$ $(1 \leq i \leq |w|)$ denote the letter in the i'th position of w, and $w[i,j]$ denote the substring $w[i]w[i+1]\dots w[j]$ of w. For $w \in A^\star$ and $B \subseteq A$, let $w|_B$ denote the word w projected to the letters in B.

Transactions and Schedules. A program consists of a set of threads that run concurrently. Each thread executes a series of *transactions*. A transaction is a sequence of *actions*; each action can be a read or a write to a (global) variable, or a synchronization action.

We assume an infinite set of thread identifiers $\mathcal{T} = \{T_1, T_2, \dots, \}$. We also assume an infinite set of entity names (or just entities) $\mathcal{X} = \{x_1, x_2, \dots, \dots\}$ that the threads can access. The set of actions that a thread T can perform on a set of entities $X \subseteq \mathcal{X}$ is defined as $\Sigma_{T,X} = \{T{:}\triangleright, T{:}\triangleleft\} \cup \{T{:}\mathrm{read}(x), T{:}\mathrm{write}(x) \mid x \in X\}$. Actions $T{:}\mathrm{read}(x)$ and $T{:}\mathrm{write}(x)$ correspond to thread T reading and writing to entity x, while $T{:}\triangleright$ and $T{:}\triangleleft$ correspond to the beginning and the end of transaction blocks in thread T.

Define $\Sigma_X = \bigcup_{T \in \mathcal{T}} \Sigma_{T,X}$ (actions on entities X by all threads), $\Sigma_T = \bigcup_{X \in \mathcal{X}} \Sigma_{T,X}$ (actions by thread T on all entities), and $\Sigma = \bigcup_{X \in \mathcal{X}, T \in \mathcal{T}} \Sigma_{T,X}$ (all actions).

For a word $w \subseteq \Sigma^\star$, let $w|_T$ be a shorthand notation for $w|_{\Sigma_T}$, which includes only the actions of thread T from w. The following defines the notion of observable behaviors on the global variables of a concurrent program, which we call a *schedule*.

Let $\mathit{Tran}_{T,\mathcal{X}} = (T{:}\triangleright) \cdot \{T{:}\mathrm{read}(x), T{:}\mathrm{write}(x) \mid x \in \mathcal{X}\}^\star \cdot (T{:}\triangleleft)$. A *transaction* tr of a thread T is a word in $\mathit{Tran}_{T,\mathcal{X}}$. Let $\mathit{Tran}_T = (\mathit{Tran}_{T,\mathcal{X}})^\star$ denote the set of all possible sequences of transactions for a thread T, and let Tran denote the set of all possible transaction sequences.

Definition 1. *A* schedule *over a set of threads $\mathcal{T}$ and entities $\mathcal{X}$ is a word $\sigma \in (\Sigma_{\mathcal{T},\mathcal{X}})^\star$ such that for each $T \in \mathcal{T}$, $\sigma|_T$ belongs to Tran_T. Let $\mathit{Sched}_{\mathcal{T},\mathcal{X}}$ denote the set of all schedules over threads $\mathcal{T}$ and entities $\mathcal{X}$.*

In other words, a schedule is a sequence of actions such that its projection to any thread T is a word divided into a sequence of transactions, where each transaction begins with $T{:}\triangleright$, is followed by a set of reads and writes, and ends with $T{:}\triangleleft$.

When we refer to two particular actions $\sigma[i]$ and $\sigma[j]$ in σ, we say they *belong to the same transaction* if they are actions of the same thread T, and they are in the same transaction block in $\sigma|_T$: i.e. if there is some T such that $\sigma[i], \sigma[j] \in \mathcal{A}_T$, and there is no i', $i < i' < j$ such that $\sigma[i'] = T{:}\triangleleft$.

Concurrent executions with lock-synchronization. Let us now define *executions* of concurrent programs that synchronize using (nested) locks. An *execution* is more detailed than a *schedule* in that it also contains the synchronization actions a program performs. In this paper, we limit the synchronization actions to acquires and releases of global locks.

Let us fix a set of global locks $\mathcal{L}$. For a thread $T \in \mathcal{T}$ and a set of locks $L \subseteq \mathcal{L}$, define the set of lock-actions of T on L by $\Pi_{L,T} = \{T{:}\mathit{acquire}(l), T{:}\mathit{release}(l) |\ l \in L\}$. Let $\Pi_L = \bigcup_{T \in \mathcal{T}} \Pi_{L,T}$, and $\Pi_T = \Pi_{\mathcal{L},T}$, and finally $\Pi = \bigcup_{T \in \mathcal{T}} \Pi_T$.

A word $\gamma \in \Pi^\star$ is *lock-valid* if it respects the semantics of the locking mechanism; formally, for every $l \in \mathcal{L}$, $\gamma|_{\Pi_{\{l\}}}$ is a prefix of $\left[\bigcup_{T \in \mathcal{T}} (T{:}\mathit{acquire}(l)\ T{:}\mathit{release}(l))\right]^\star$.

A global execution over the set $\mathcal{L}$ is a *finite* word $\rho \in (\Sigma \cup \Pi_{\mathcal{L}})^\star$ such that (a) for any thread T, $\rho|_\Sigma$ is a schedule and (b) $\rho|_{\Pi_{\mathcal{L}}}$ is lock-valid.

In other words, a global execution is a finite sequence of actions, involving a finite set of threads accessing a finite set of variables, along with acquisitions and releases of locks, such that the sequence projected to any thread forms a sequence of transactions, and the sequence respects the locking mechanism.

We will often handle *local executions* as well, which are executions of individual threads. Formally, **a local set of executions over the set** $\mathcal{L}$ is a set $\{\alpha_t\}_{t \in \mathcal{T}}$, where for each thread T, $\alpha_T \in \left(\Sigma_{\{T,\mathcal{X}\}} \cup \Pi_{\mathcal{L}}\right)^*$. Note that a global execution ρ naturally defines a set of local executions $\{\rho_T\}_{T \in \mathcal{T}}$.

An event in a set of local executions $\{\rho_T\}_{T \in \mathcal{T}}$ is a pair (T, i) where $T \in \mathcal{T}$ and $1 \leq i \leq |\rho_T|$. In other words, an event is a particular action that one of the threads executes, indexed by its local timestamp.

Let ρ be a global execution, and $e = (T, i)$ be an event in $\{\rho_T\}_{T \in \mathcal{T}}$. Then we say that the j'th action ($1 \leq j \leq |\rho|$) in ρ *is the event* e (or, $\mathit{Event}(\rho[j]) = e = (T, i)$), if $\rho[j] = T{:}a$ (for some action a) and $\rho_T[1, i] = \rho[1, j]|_T$. In other words, the event $e = (T, i)$ appears at the position j in ρ in the particular interleaving of the threads that constitutes ρ. Conversely, for any event e in $\{\rho_T\}_{T \in \mathcal{T}}$, let $\mathit{Occur}(e, \rho)$ denote the (unique) j ($1 \leq j \leq |\rho|$) such that the j'th action in ρ is the event e, i.e $\mathit{Event}(\rho[j]) = e$. Therefore, we have $\mathit{Event}(\rho[\mathit{Occur}(e, \rho)]) = e$, and $\mathit{Occur}(\mathit{Event}(\rho[j])) = j$.

A (global) execution ρ **over** $\mathcal{L}$ **is said to respect *nested-locking*** if there is no thread T and two locks l and l' such that $\rho|_{\Pi_{\{l,l'\},\{T\}}}$ has a contiguous subsequence $T{:}\mathit{acquire}(l)T{:}\mathit{acquire}(l')T{:}\mathit{release}(l)$. In other words, an execution respects nested-locking if each thread releases locks strictly in the reverse order in which they were acquired.

Finally, for any execution ρ, the schedule defined by ρ is the word obtained by removing the locking-events from it: $Sched(\rho) = \rho|_{\Sigma}$.

The prediction model. Given an execution ρ over a set of locks $\mathcal{L}$, we would like to *infer* other executions ρ' from ρ. This prediction model we consider is defined as follows. An execution ρ' belongs to the inferred set of ρ iff ρ'_T is a prefix of ρ_T, for every thread T. (Of course, ρ' is lock-valid by the merit of being an execution.)

In other words, we infer executions from ρ by projecting ρ to each thread to obtain local executions, and combining these local executions into a global execution ρ' in any interleaved fashion that respects the synchronization mechanism. Let $Infer(\rho)$ denote the set of executions inferred from ρ.

Notice that our prediction model is an *abstraction* of the problem of finding alternate runs that violate atomicity in the concrete program, and is quite optimistic: it recombines executions in any manner that respects the locking constraints. Of course, these executions may not be valid/feasible in the original program (this could happen if the threads communicate using other mechanisms; for example, if a thread writes a particular value to a global variable based on which another thread chooses an execution path, an execution that switches these events may not be feasible). The choice of a simple abstract prediction model is *deliberate*: while we could build more accurate models, we believe that having a simple prediction model can yield faster algorithms. Since we can in any case try to execute a predicted interleaving that violates atomicity and check whether it is feasible, this will not contribute to the final false positives in a testing scenario.

Deadlock freedom. We say that an execution ρ is deadlock-free if no run inferred from ρ deadlocks. Formally, ρ is *deadlock-free* if for every $\rho' \in Infer(\rho)$, there is a $\rho'' \in Infer(\rho)$ such that ρ' is a prefix of ρ'' and $|\rho| = |\rho''|$ (i.e. any partial execution inferred from ρ can be completed to another that executes all the actions of ρ). Note that deadlock freedom is defined abstractly by combining events of the execution, and not on the concrete program. See the end of Section 3 for a discussion on deadlock freedom.

Defining atomicity through serializability. We now define atomicity as the notion of *conflict serializability*. Define the *dependency* relation D as a symmetric relation defined over the events in Σ, which captures the dependency between (a) two events accessing the same entity, where one of them is a write, and (b) any two events of the same thread, i.e.,

$$\begin{aligned} D = \{(T_1{:}a_1, T_2{:}a_2) \mid\ & T_1 = T_2 \text{ and } a_1, a_2 \in A \cup \{\triangleright, \triangleleft\} \text{ or} \\ & \exists x \in \mathcal{X} \text{ such that } (a_1 = \text{read}(x) \text{ and } a_2 = \text{write}(x)) \text{ or} \\ & (a_1 = \text{write}(x) \text{ and } a_2 = \text{read}(x)) \text{ or } (a_1 = \text{write}(x) \text{ and } a_2 = \text{write}(x))\} \end{aligned}$$

Definition 2 (Equivalence of schedules). *The equivalence of schedules is defined as the* smallest *equivalence relation* $\sim \subseteq Sched \times Sched$ *such that: if* $\sigma = \rho ab \rho', \sigma' = \rho ba \rho' \in Sched$ *with* $(a, b) \notin D$, *then* $\sigma \sim \sigma'$.

It is easy to see that the above notion is well-defined. Two schedules are considered equivalent if we can derive one schedule from the other by iteratively swapping consecutive independent actions in the schedule.

We call a schedule σ *serial* if all the transactions in it occur sequentially: formally, for every i, if $\sigma[i] = T{:}a$ where $T \in \mathcal{T}$ and $a \in A$, then there is some $j < i$ such that $T[i] = T{:}\triangleright$ and every $j < j' < i$ is such that $\sigma[j'] \in A_T$. In other words, the schedule is made up of a sequence of complete transactions from different threads, interleaved at boundaries only.

Definition 3. *A schedule is* serializable *(or atomic) if it has an equivalent serial schedule. That is, σ is a serializable schedule if there is a* serial *schedule σ' such that $\sigma \sim \sigma'$.*

2.1 Serializability Violations Involving Two Threads and one Variable

While the above defines the general notion of serializability, in this paper we confine ourselves to checking a more restricted notion called *minimally serializable*; a schedule is minimally serializable if there are no serializability violations that involve two threads and a single variable only. More precisely,

Definition 4. *A schedule σ is* minimally serializable *(or minimally atomic) if for every pair of threads (T, T') and every entity $x \in \mathcal{X}$, the schedule $\sigma|_{\Sigma_{\{T,T'\},\{x\}}}$ is serializable. An execution ρ is minimally serializable if the schedule corresponding to it, $Sched(\rho)$, is minimally serializable.*

We can now define the precise problem we consider in this paper:

[Problem of meta-analysis of executions for minimal serializability:]

Given: A finite deadlock-free execution ρ over a set of threads, entities and locks.

Problem: Is there any execution $\rho' \in \mathit{Infer}(\rho)$ that is not minimally serializable?

First, note that an execution ρ' is not minimally serializable iff there exist two threads T and T' such that ρ' projected to T and T' is not serializable. Even for a fixed T and T', there are a large number of interleavings possible (in fact, exponential in $|\rho|$), making an explicit search over interleavings infeasible.

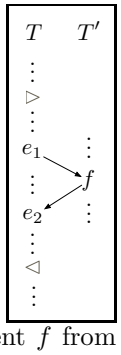
A better way of solving the above problem is to build an automaton that generates all possible interleavings of two threads T and T', and *intersect* it with another automaton that detects atomicity violations [3]. The state of this automaton is represented by a triple (x, y, π) where x tracks the position in the first thread, y tracks the position of the second thread, and $\pi : \mathcal{L} \to \{\bot, T, T'\}$ tracks the thread that holds each lock ($\bot$ denoting the lock is free). Alternatively, we can view this as a *dynamic programming* solution, where we track, for each pair (x, y), the state of the monitor on them.

Though the above algorithm does not explicitly enumerate all interleavings, it works in time $O(n^2)$ (where $n = |\rho|$). Since n, the length of the given run, can

be extremely large (millions of events in our benchmarks), an algorithm that runs in time quadratic in n simply will not scale in practice.

The goal of this paper is to find an algorithm that solves the above problem in time *linear* in n (and linear in the number of entities, and quadratic in the number of threads). Note that this means that we do not explore all interleavings explicitly, and yet predict atomicity violations. Our scheme is in fact *compositional*; we extract a finite amount of information from each local execution in linear time, and combine this information to predict minimal atomicity violations.

3 Meta-analysis of Runs for Minimal Serializability



In this section, we present the main result of this paper: the basis for an algorithm that solves the meta-analysis problem for minimal serializability in time linear in the length of the given run. We will show how meta-analysis for minimal serializability can be reduced to the *global reachability problem* for two threads, which in turn can be compositionally and efficiently solved for nested-locking programs. The results obtained in this section will be used to formulate our algorithm in Section 4.

The first observation is that only three events are relevant in finding a violation of minimal serializability; we need to observe two events e_1 and e_2 from a *single transaction* of a thread T and an event f from another thread T' such that e_1 and f are dependent and e_2 and f are dependent. Moreover, and crucially, there should exist an execution in which f occurs after e_1, and e_2 occurs after f. The figure on the right describes this pattern, and the following lemma captures this property:

Lemma 1. *Let ρ be a global execution, and let $\{\rho_T\}_{T\in\mathcal{T}}$ be the set of local executions corresponding to it. Infer(ρ) contains a minimally non-serializable run iff there exists two different threads T and T', an entity $x \in \mathcal{X}$, and $\rho' \in$ Infer$(\rho|_{\Sigma_{\{T,T'\},\{x\}}})$ such that there are (read or write) events e_1, e_2, f of $\{\rho_T\}_{T\in\mathcal{T}}$ where*

- *Occur$(e_1, \rho') <$ Occur$(f, \rho') <$ Occur(e_2, ρ')*
- *e_1 and e_2 are events of thread T, and f is an event of thread T'*
- *e_1 and e_2 belong to the same transaction,*
- *e_1 D f D e_2.*

While we can find candidate events e_1 and e_2 from thread T and a candidate event f from T' by *individually* examining the local runs of T and T', the main problem is in ensuring the condition that we can find an inferred run where e_1 occurs before f and f occurs before e_2. This is hard as the threads synchronize using (nested) locks which needs to be respected by the inferred run. In fact, for threads communicating using general locking, our results in [3] show that

it is highly unlikely to avoid considering the two thread runs in tandem, which involves $O(n^2)$ time.

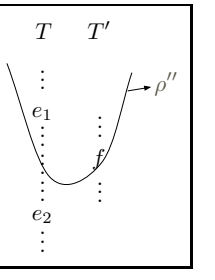
Let us first show the following lemma that reduces checking whether three events e_1, f, and e_2 are executable in that order, to global reachability of two threads (this lemma has nothing to do with atomicity violations).

Lemma 2. *Let ρ be a deadlock-free execution, and let T, T' be two threads with $T \neq T'$. Let e_1, e_2, f be (read or write) events in $\{\rho_T\}_{T \in \mathcal{T}}$ such that $e_1 = (T, i_1)$ and $e_2 = (T, i_2)$ are events of thread T with $i_1 < i_2$, and f is an event of thread T'. Then, there is an execution $\rho' \in Infer(\rho)$ such that $Occur(e_1, \rho') < Occur(f, \rho') < Occur(e_2, \rho')$*

if, and only if,

there is an execution $\rho'' \in Infer(\rho)$ such that

- *e_1 occurs in ρ'' and e_2 does not occur in ρ'', and*
- *f occurs in ρ'', and in fact f is the last event of T' that occurs in ρ''.*



Intuitively, the above lemma says the following: fix an execution ρ, and three events e_1, e_2, f in it such that events e_1 and e_2 belong to the same transaction (and thread) and event f belongs to a different thread. Then, we can find a run inferred from ρ that executes event e_1 followed by event f followed by event e_2 if, and only if, we can find an (incomplete) inferred run that executes events e_1 of thread T (and possibly later events), but does not execute e_2, and executes precisely up to event f in thread T'. This is depicted in the figure on the right.

The above lemma is useful as it reduces finding a series of three events to the simpler global reachability question of a set of pairs of positions in the two threads.

Pairwise reachability. Our final hammer in solving the problem relies on a beautiful result by Kahlon et al. [9] that argues that global reachability of two threads communicating via nested locks is effectively and *compositionally* solvable by extracting locking information from the two threads in terms of *acquisition histories*.

Let ρ be an execution and let $\{\rho_T\}_{T \in \mathcal{T}}$ be its set of local executions. Consider ρ_T (for any T). The *lock-set held after* ρ_T is the set of all locks T holds: $LockSet(\rho_T) = \{l \in \mathcal{L} \mid \exists i. \rho_T[i] = T{:}acquire(l)$ *and there is no* $j > i$ *and* $\rho_T[j] = T{:}release(l)\}$.

The *acquisition history* of ρ_T records, for each lock l held by T at the end of ρ_T, the set of locks that T acquired (and possibly released) after the last acquisition of the lock l. Formally, the acquisition history of ρ_T, $AH(\rho_T) : LockSet(\rho_T) \to 2^{\mathcal{L}}$, where $AH(l)$ is the set of all locks $l' \in \mathcal{L}$ such that $\exists i. \rho_T[i] = T{:}acquire(l)$ and there is no $j > i$ such that $\rho_T[j] = T{:}release(l)$ and $\exists k > i. \rho_T[k] = T{:}acquire(l')$.

Two acquisition histories AH and AH' are said to be *compatible* if there do not exist two locks l and l' such that $l' \in AH(l)$ and $l \in AH(l')$. The following is a direct consequence of a result by Kahlon et al. [9], which says that there is an

execution that ends with event e in one thread and event f in the other thread, if, and only if, the acquisition histories at e and f are compatible.

Lemma 3 (Kahlon et al. [9]). *Let ρ be an execution, let $\{\rho_T\}_{T\in\mathcal{T}}$ be its set of local executions, and let T and T' be two different threads. Let $e = (T, i)$ be an event of thread T and $f = (T', j)$ be an event of thread T' of these local executions.*

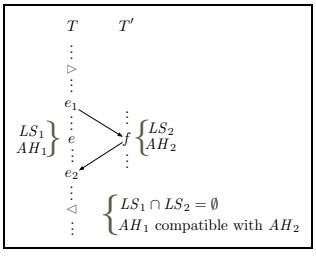
There is a run $\rho' \in Infer(\rho)$ such that e and f occur in ρ', and further, $\rho'_T = \rho_T[1, i]$ and $\rho'_{T'} = \rho_{T'}[1, j]$

if, and only if,

$Lockset(\rho_T[1, i]) \cap Lockset(\rho_{T'}[1, j]) = \emptyset$, and the acquisition history of $\rho_T[1, i]$ and the acquisition history of $\rho_{T'}[1, j]$ are compatible. □

We have reduced checking of serializability to pairwise reachability, which is solvable compositionally by computing the acquisition histories from each thread, and checking them for compatibility. We summarize this in the following:

Theorem 1. *Let ρ be a deadlock-free global execution. A minimally non-serializable execution can be inferred from ρ iff there exists two different threads T and T' and an entity $x \in \mathcal{X}$, and there are events $e_1 = (T, i), e_2 = (T, i'), f = (T', j)$ of $\{\rho_T\}_{T\in\mathcal{T}}$ such that*



- *e_1 and e_2 belong to the same transaction,*
- *e_1 D f D e_2,*
- *There is an event $e = (T, i'')$ of $\{\rho_T\}_{T\in\mathcal{T}}$ such that $i \leq i'' < i'$ and the acquisition histories of $\rho_T[1, i'']$ and $\rho_{T'}[1, j]$ are compatible.*

On the assumption of deadlock freedom. Deadlock freedom is a strong assumption. Note that a deadlocking inferred run may not be feasible in the concrete program. We use the deadlock-free assumption to ensure that the partial run (containing e_1 and f) can be completed to a full run where e_2 occurs. We can remove the assumption of deadlock freedom and build a more sophisticated algorithm (using locksets and *backward* acquisition histories) that ensures that e_2 is also executed; however, this complicates the algorithm considerably, and we have chosen not to implement it for practical reasons. In the benchmarks that we experiment with, all partial runs could be completed to full runs.

Actually, under the assumption of deadlock freedom, checking compatibility of acquisition histories is redundant, as it suffices to check if locksets are disjoint. However, we check compatibility of acquisition histories in order not to rely on the assumption of deadlock freedom to generate the events e_1 and f.

4 The Meta-analysis Algorithm for Minimal Serializability

Given a set of local executions $\{\rho_T\}_{T\in\mathcal{T}}$ with nested locking, Theorem 1 allows us to engineer an efficient algorithm to predict an interleaving of them that violates minimal serializability.

The aim is to find three events e_1, e_2, and f, where e_1 and e_2 occur in the same transaction in a thread T, f occurs in a different thread T', with $e_1 D f D e_2$, and further, find an event e between e_1 and e_2 (e is allowed to be e_1 but not to be e_2) such that the locksets of e and f are disjoint, and their acquisition histories are compatible.

The algorithm is divided into two phases. In the first phase, it gathers the lockset and acquisition histories of all possible witnesses e and all possible witnesses f; this is done by examining the events of each thread *individually*. In the second phase, we test the compatibility of the locksets and acquisition histories of every pair of witnesses e and f in different threads.

Let us fix an entity $x \in \mathcal{X}$. We divide our work into finding two patterns: one where e_1 and e_2 are writes to x and f is a read of x, and the other where e_1 and e_2 are accesses (read/write) to x and f is a write to x. This clearly covers all cases of minimal serializability violations— the former covers violations $e_1 - f - e_2$ of the form $Write$–$Read$–$Write$, while the latter covers those of the form $Read$–$Write$–$Read$, $Read$–$Write$–$Write$, $Write$–$Write$–$Read$ and $Write$–$Write$–$Write$.

Phase I. In the first phase, for each thread T and each entity x, the algorithm gathers witnesses in four lists: $R[T,x]$, $W[T,x]$, $WW[T,x]$ and $AA[T,x]$. Intuitively, the sets $R[T,x]$ and $W[T,x]$ gather witnesses of events of thread T that read and write to x, respectively, for each lockset and acquisition history pair, in order to witness the event f in our pattern.

The set $WW[T,x]$ gathers all witnesses e that are sandwiched between two write-events to x that occur in the same transaction of thread T, keeping only one representative witness for each lockset and acquisition history pair. Similarly $AA[T,x]$ gathers witnesses e sandwiched between any two accesses of thread T to x that occur in the same transaction.

The algorithm gathers the witnesses by processing the execution in a single pass. It continually updates the lockset and acquisition history, adding events to the various witness sets, making sure that no set has multiple events with the same lockset and acquisition history. Note that the computation of $WW[T,x]$ and $AA[T,x]$ sets need care due to the fact that events e recorded must be validated by a later occurrence of the relevant event e_2.

Note that phase I considers every event at most once, in one pass, in a streaming fashion, and hence runs in time linear in the length of the execution.

Phase II. In the second phase, the algorithm checks whether there are pairs of compatible witnesses that were collected in the first phase. More precisely, we check whether, for any entity x, and for any pair of threads T and T', there is an event $e \in WW[T,x]$ and an event $f \in R[T',x]$ that have disjoint locksets and compatible acquisition histories. Similarly, we also check whether there is an event $e \in AA[T,x]$ and an event $f \in W[T',x]$ that have disjoint locksets and compatible acquisition histories. The existence of any such pair of events would mean (by Theorem 1) that there is a minimal serializability violation.

```
1 for each entity x ∈ 𝒳 do
2   for each T, T' in 𝒯 such that T ≠ T' do
3     for each (e, LS, AH) in WW[T, X] do
4       for each (f, LS', AH') in R[t, x] do
5         if (LS ∩ LS' = ∅  ∧  AH and AH' are compatible) then
6           Report minimal serializability violation found;
7 end
```

Fig. 1. Phase II for R-WW patterns

For example, the algorithm runs the procedure in Figure 1 for finding the violations using the R and WW sets (the procedure using the W and AA sets is similar):

Note that phase II runs in time $O(t^2.v.ah^2)$ where t is the number of threads, v is the number of entities accessed, and ah is the total number of disjoint acquisition histories of events in the thread. Note that this is *independent* of the length of the execution (Phase I summarized the events in the execution, and Phase II does not consider the run again). The compatibility check for acquisition histories can be done in time linear in the size of the acquisition histories (however, our current implementation is quadratic.).

The quadratic dependence on the number of threads is understandable as we consider serializability violation between all pairs of threads. The linear dependence on x is very important for scalability as the number of entities accessed can be very large on typical runs. The number of different acquisition histories, in theory, can be large ($O(2^{l^2})$, where the execution uses l locks)— however, in practice, there tend to be very few distinct acquisition histories that get manifested, and hence is not a bottleneck (see the next section for details).

Though we record violations only in terms of the two witnesses e and f, we can actually recover a precise execution that shows the atomicity violation. This run can be obtained using the locksets and acquisition histories of e and f (using the method prescribed by Kahlon et al. [9]), and may in fact involve several context switches ($O(l)$ of them, if there are l locks) to execute the atomicity violation. However, this replay of an exact run that violates atomicity is involved, and has not been implemented.

5 Implementation and Experiments

We have implemented the meta-analysis algorithm to look for minimal serializability violations in the set of executions inferred from a single execution of a concurrent program. Note that though the real problem is to find alternate executions of a concurrent program that violate minimal serializability, we have, in this paper, adopted a simple prediction model, according to which we find alternate runs. The algorithm that we implement (as detailed in the previous sections) is sound and complete with respect to our prediction model, but clearly not with respect to the concrete model. In particular, the alternate schedules that we report may not be feasible in the concrete model. We could, of course, try to schedule them and see if they are feasible; however, this is a complex engineering task that is out of the scope of this paper. Preliminary analysis of the runs found

in our experiments, however, suggest that most executions are indeed feasible in the concrete program (see below).

Note that our algorithm computes only one representative violation for each pair of threads, each entity, each pair of events with a compatible set of locksets and acquisition histories, and each pattern of violation (R-W-R and A-R-A). The current implementation does not find all multiplicities of these serializability violations. Note that our algorithm guarantees to report at least one minimal serializability violation, if violations at all exist. The tool can also be easily modified to enumerate all possible violations. More specifically, after the second phase of the algorithm, all the *interesting acquisition histories* are known, so one can use this information and the original execution to generate all violations without significant performance cost.

We evaluated the algorithms on a benchmark suite of six concurrent Java programs that use `synchronized` blocks and methods as means of synchronization (note that using synchronized blocks automatically ensures nested locking, and is one of the reasons why nested locking programs are common). They include `raytracer` from the Java Grande multithreaded benchmarks [15], `elevator`, `tsp`, and `hedc` from [16], and `Vector` and `HashTable` from Java libraries. `elevator` simulates multiple lifts in a building, `tsp` solves the traveling salesman problem in parallel for a given input map, `raytracer` renders a frame of an arrangement of spheres from a given view point, `hedc` is a web-crawler, and `Vector` and `HashTable` are Java libraries that respectively implement the concurrent vector and the concurrent hashtable data structures.

Since *recurrent* locks (multiple acquisitions of the same lock by the same thread) are typical in Java, the tool is tuned to handle them by ignoring all the subsequent acquisitions of the same lock by the same thread. One can easily verify that this approach maintains well-nestedness of locks and (hence) the correctness of the main theorem of the paper.

We investigated the other concurrent benchmarks in the Java Grande suite, but they either had no synchronization, or used *barriers* as their synchronization mechanism, and hence were not good candidates for testing our analysis method.

Executions were extracted by (manually) instrumenting programs to output accesses to entities and synchronization events at runtime. We have a simple *escape analysis* unit that excludes from the execution all accesses to thread-local entities. We then run the meta-analysis algorithm on these output files off-line. The algorithm can be implemented as an online algorithm as well (as Phase I can be implemented online), but the current implementation works off-line.

Table 1 presents the results of our evaluation. We ran each benchmark with different input parameters, such as number of threads and input files. For each program, we report in the table the number of lines of code (LOC) (appears below the program names), number of threads used in the execution, the number of *truly shared* entities between threads, the number of locks, the number of transactions, and the length of the global execution (number of events). The table presents the results of the meta-analysis of the generated executions; in particular, we report how many inferred executions with minimal serializability violations were found

Table 1. Running Results (K=1000; M=1000000)

Application (LOC)	Threads	Entities	Locks	Events	Trans	Time (s)	Violations WRW	Violations RWR /RWW WWR /WWW
elevator (566)	3	32	8	9K	140	0.09s	0	4
	5	32	8	29K	423	0.27s	0	4
	5	200	50	78K	273	26.23s	8	12
raytracer (1537)	10	1	10	86	10	0.03s	0	90
	20	1	20	160	20	0.02s	0	380
	40	1	40	320	40	0.16s	0	1560
hedc (2165)	10	6	2	176	10	0.03s	20	4
	10	6	2	132	10	0.03s	12	5
tsp (794)	3	30	2	97	5	0.03s	0	0
	8	50	2	18K	10	0.53s	0	0
	5	140	2	1.4M	17	1.55s	16	248
	10	140	2	2.5M	31	2.51s	36	339
	20	1510	2	11M	106	12.58s	171	1609
stack (1400)	2	4	2	105	2	0.07s	0	194
vector (1281)	2	4	2	107	2	0.08s	0	144

for the two classes of patterns (these correspond to possible alternate executions we could schedule in a testing scenario). Note that the cause for the somewhat large number of violations reported for some benchmarks is that a single problem in the program could manifest itself in several inferred violations. For example, one wrongly synchronized method for the class `Vector` causes many violations, one for each time it is executed in parallel with (almost) every other method in the class.

The executions were obtained using various machines (2-core and 4-core); the analysis of the executions were performed on a 2.16 Ghz Intel Core2 Duo laptop with 2 GB of memory running MacOS 10.5.

Our results clearly illustrate the tremendous impact of using an algorithm that runs in time linear in the length of the execution. There are examples of long executions for the `tsp` benchmark for which the algorithm finds the violations very quickly. For example, in the setting with 20 threads and more then 11 million events, the algorithm finds 1780 violations in less than 13 seconds. The only exception is `elevator` with 50 locks. However, we noticed that for this example, the time is almost entirely spent in the compatibility check between two acquisition histories. Unfortunately, the compatibility check is not implemented optimally in the current version of the tool (as we need indexed dictionaries, etc. to do so); we believe that an optimized version of this procedure will considerably speed up our algorithm.

In the case of Java libraries `Vector` and `Stack`, the experiments were set to run each pair of the library methods concurrently as two threads. These experiments include many small executions (a few hundred, involving two threads, four entities, and two locks), in which many atomicity violations are found which are actually related to subtle concurrency errors in quite a few of these library methods. The numbers reported in Table 1 represent average values over these executions, while the violations are the *total* number of violations found.

Though checking whether alternate schedules found by the tool are actually feasible in the concrete program is beyond the scope of this paper, preliminary

results suggest that many are feasible (e.g. `raytracer` has all schedules feasible, `hedc` has 16/24 and 12/17 schedules feasible).

Acknowledgements. This work was partially funded by NSF CAREER Award CCF 0747041 and by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign (a center sponsored by Intel Corporation and Microsoft Corporation).

References

1. Chen, F., Serbanuta, T.F., Rosu, G.: Jpredictor: a predictive runtime analysis tool for java. In: ICSE, pp. 221–230 (2008)
2. Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
3. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)
4. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 155–169. Springer, Heidelberg (2009)
5. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL, pp. 256–267 (2004)
6. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI, pp. 293–303 (2008)
7. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349 (2003)
8. Hatcliff, J., Robby, Dwyer, M.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 175–190. Springer, Heidelberg (2004)
9. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
10. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
11. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proc. ASPLOS (2008)
12. Papadimitriou, C.: The theory of database concurrency control. Computer Science Press, Inc., New York (1986)
13. Tucek, S.J., Qin, F., Zhou, Y.: Avio: detecting atomicity violations via access interleaving invariants. In: ASPLOS, pp. 37–48 (2006)
14. Sen, K., Rosu, G., Agha, G.: Online efficient predictive safety analysis of multithreaded programs. STTT 8(3), 248–260 (2006)
15. Java Grande Benchmark Suite, `http://www.javagrande.org/`
16. von Praun, C., Gross, T.R.: Object race detection. SIGPLAN Not. 36(11), 70–82 (2001)
17. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP, pp. 137–146 (2006)
18. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multi-threaded programs. IEEE Transactions on Software Engineering 32, 93–110 (2006)
19. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. 40(6), 1–14 (2005)