Atomicity Checking in Linear Time using Vector Clocks

Umang Mathur* Mahesh Viswanathan†

Department of Computer Science, University of Illinois at Urbana Champaign, Urbana, IL, USA
Email: *umathur3@illinois.edu, †vmahesh@illinois.edu

Abstract—Multi-threaded programs are challenging to write and developers often need to reason about a prohibitively large number of thread interleavings to reason about the behavior of software. A non-interference property like atomicity can reduce this interleaving space by ensuring that any execution is equivalent to an execution where all atomic blocks are executed serially. We consider the well studied notion of conflict serializability for dynamically checking atomicity. Existing algorithms detect violations of conflict serializability by detecting cycles in a graph of transactions observed in a given execution. The size of such a graph can be as large as the size of the trace making the analysis not scalable. In this paper, we present AeroDrome, a novel single pass linear time algorithm that detects violations of conflict serializability using vector clocks. Experiments show that AeroDrome scales to traces with a large number of events with significant speedup.

Index Terms—dynamic analysis, concurrency, atomicity, conflict serializability, vector clocks

I. INTRODUCTION

Writing correct multi-threaded programs is extremely difficult and is the class of software that is most prone to errors. Reasoning about such multi-threaded programs is notoriously challenging due to the inherent nondeterminism that arises from thread scheduling in such systems. If the program satisfies certain fundamental properties then reasoning about them becomes easier, and if such properties are violated then it is often symptomatic of more serious bugs in the software. Atomicity is one such classical concurrency property, which guarantees that a programmer reasoning about a concurrent program can assume that atomic blocks of code can be executed sequentially without any context switches in between. Atomicity allows programmers to reason about atomic blocks without worrying about the effects of other threads. Unfortunately, violation of atomicity specs is quite common and is the root cause in a majority of real-world bugs [22].

Various approaches to identifying atomicity violations have been explored. Static analysis based approaches for atomicity checking are usually conservative, computationally expensive, and often rely on user annotations, like type annotations [1], [9], [10], [15], [32], [37]. The advantage of static analysis approaches is that they may successfully prove a program to satisfy all its atomicity requirements. Dynamic analysis for atomicity violations, on the other hand, have the advantage that they are fully automated and are computationally less expensive [3], [7], [8], [14], [21], [40]. Though they cannot prove that a program satisfies its atomicity specification, dynamic analysis can be used to check if an observed trace is witness to the violation of atomicity. Given their scalability, dynamic analysis for atomicity violation has proved to be very useful in practice.

In this paper, we will focus on sound and precise dynamic analyses; unsound dynamic analyses have the disadvantage that they report many false alarms. All sound and precise dynamic analyses [3], [7], [14] for atomicity violation are based on checking the conflict serializability of an observed program execution. An execution is conflict serializable if it can be transformed into an equivalent execution, where all statements in an atomic block are executed consecutively without context switches, by commuting adjacent, non-conflicting operations of different threads. Here conflicting operations are either two operations by the same thread, two accesses to a common global variable one of which is a write, or acquires and releases of common locks. Determining if an execution is conflict serializable can be reduced to checking for the existence of a cycle in a graph called the transaction graph. The transaction graph has atomic blocks (a.k.a. transactions) as vertices, and edges between blocks that contain non-commutable events. A path from atomic block A to B indicates that A must be executed before B in a serial execution, and so a cycle in such a graph indicates that the execution is not equivalent to a serial one. All current sound and precise dynamic analyses for conflict serializability [3], [14] rely on this idea and so have an asymptotic complexity of quadratic time — each new event of the trace requires updating the transaction graph, and checking for cycles, which gives a linear processing time per event.

The central question motivating this paper is the following: Is a quadratic running time necessary for checking conflict serializability? Or are there linear time algorithms for this problem? The main result of this paper is a new, linear time algorithm for checking conflict serializability.

For other concurrency specifications, like data race detection, that admit linear time sound and precise algorithms, the key to achieving an efficient algorithm is the use of vector clocks [25]. Such algorithms rely on computing vector timestamps for events in a streaming fashion as the trace is generated, and using these timestamps to recover the causal order between a pair of events. However, generalizing such an algorithmic principle to conflict serializability checking is far from straightforward. This is because checking conflict serializability requires identifying causal orders between
transactions (or atomic blocks) and not individual events. For this reason, Flanagan-Freund-Yi [14], in fact, dismiss the possibility of a vector clock based algorithm for conflict serializability checking.

“The traditional representation of clock vectors [25] is not applicable because our happens-before relation is over compound transactions and not individual operations.”

The challenge is to discover a way to associate a single timestamp with a transaction, even though new causal dependencies are discovered as each individual event in the trace is processed. This is further complicated by the following observation. Vector timestamps implicitly summarize the set of all events that must be ordered before. However, the set of transactions that must be executed before a transaction \( T \) might be known only well after all the events of \( T \) have been seen (see Example 2). These observations suggest that a scheme of assigning vector timestamps to transactions may only be computed if the algorithm makes multiple passes over the trace, which may result in an algorithm that is not linear time.

We address these challenges by assigning vector timestamps to individual events in a trace. The induced order on events is then used to discover the ordering relationship between transactions, and thereby determining if a trace is conflict serializable. For a trace containing a bounded number of variables, threads, and locks, our algorithm is a single pass, streaming algorithm that runs in linear time.

We have implemented our algorithm in our tool AIRTRACK. Our implementation first logs events as a program is running. We then run different algorithms for checking conflict serializability on the logged traces. The first phase of logging is not necessary; since our algorithm processes the trace in a streaming fashion, if our only goal was to build an atomicity checking tool, we would have implemented our algorithm so that it directly processes the trace as it is generated without logging it first. However, we log a trace so that we can run multiple conflict serializability checking algorithms on the same trace and thereby make a fair comparison between the running times of different algorithms.

We compared the performance of our algorithm against Velodrome [14] on various benchmark programs. Obtaining atomicity specifications (i.e., which blocks of code should be regarded as atomic) are difficult to come by. One naïve specification is to consider each method call to be atomic. Since often there is a main method for each thread, this means that the entire computation of each thread should be atomic. Programs are unlikely to satisfy such strong atomicity specifications, but running detection algorithms against these, gives us a baseline. We use such naïve specifications for some programs in our benchmark. On such examples, the small transaction graph coupled the overhead of maintaining vector clocks outweighs the benefits of a linear time algorithm, and Velodrome slightly outperforms our tool AIRTRACK. For other programs in our benchmark, we use the more realistic atomicity specifications given in [3]. Here transactions consist of smaller blocks of code, and the resulting transaction graph has many transactions. For such examples, our algorithm significantly outperforms Velodrome. This suggests that on realistic atomicity specifications, the benefits of having a linear time algorithm can be significant.

The rest of the paper is organized as follows. In Section II, we discuss preliminary notations such as that of concurrent program traces and the definition of conflict serializability. In Section III, we use motivating examples to illustrate the challenges involved in developing a linear time vector clock algorithm for dynamically checking conflict serializability. In Section IV, we discuss AeroDrome, a single pass linear time vector clock algorithm for checking conflict serializability, which is also the main contribution of the paper. Section IV also discusses the correctness and complexity guarantees of the algorithm and optimizations for improving the performance of AeroDrome. Our implementation of AeroDrome in our tool AIRTRACK and its performance evaluation on a suite of benchmark programs is discussed in Section V. We discuss closely related work in Section VI and present concluding remarks in Section VII.

II. Preliminaries

A trace of a concurrent program is a sequence of events performed by different threads. We will use \( \sigma, \sigma', \rho_1, \rho_2, \ldots \) to denote traces. Each event is a tuple \( e = \langle t, o \rangle \), where \( t \) denotes the thread that performs \( e \) and \( o \) is the operation performed by \( e \); we will use \( thr(e) \) to denote \( t \) and \( op(e) \) to denote \( o \). Operations can be one of \( r(x), w(x) \) (read from or write to variable/memory location \( x \)), \( acq(\ell), rel(\ell) \) (acquire or release of lock object \( \ell \)), \( fork(u), join(u) \) (fork or join of thread \( u \)), \( > \) or \( < \) (denoting the begin or end of an atomic block). Traces are assumed to be well-formed — all lock acquires and releases are well matched, a lock is not acquired by more than one thread at a time, all begin and end events are well matched, fork events occur before the first event of the child thread and join events occur after the last event of the child thread. A transaction \( T \) in thread \( t \) is a maximal subsequence \(^2\) of events of thread \( t \) that starts with \( \langle t, > \rangle \) and ends with the matching \( \langle t, < \rangle \), and we say \( e \in T \) if the event \( e \) belongs to this maximal subsequence; in this case, \( trans(e) \) denotes the transaction \( T \) to which \( e \) belongs. In a trace \( \sigma \), we will say that a transaction \( T \) is completed if the corresponding end transaction event \( \langle ., < \rangle \in \sigma \). If \( T \) is not completed in \( \sigma \), it is said to be active. In the observed full trace, we will assume (without loss of generality) that all transactions are completed, though in any prefix of the observed trace, some transactions may be active.

\(^1\)Vector clock based algorithms are linear time under the computational assumption that arithmetic operations take constant time. This is a reasonable assumption in practice because even for traces with billions of events, the numbers involved in vector clocks can be stored in a single word, and so addition and subtraction of such numbers can be reasoned to be constant time.

\(^2\)We allow for nested blocks of begins and ends. In this case only the outermost begin and end constitute a transaction.
Given a trace $\sigma$, we denote by $\leq_{\sigma}$ the total order on events induced by $\sigma$ — for events $e, e'$ in $\sigma$, we say $e \leq_{\sigma} e'$ if either $e = e'$ or $e$ occurs before $e'$ in the sequence $\sigma$. Two events $e, e'$ are said to be conflicting if either (a) $\text{thr}(e) = \text{thr}(e')$, (b) $e = (t, \text{fork}(u))$ and $\text{thr}(e') = u$, (c) $\text{thr}(e) = u$ and $e' = (t, \text{join}(u))$, (d) there is a common memory location $x$ such that both $\text{op}(e), \text{op}(e')$ are one of $\{u(x), x(u)\}$ and not both are $x(u)$, or (e) there is a lock $\ell$ such that $\text{op}(e) = \text{rel}(\ell)$ and $\text{op}(e') = \text{acq}(\ell)$. Given a trace $\sigma$, conflict-happens-before $\leq_{\text{CHB}}$ is the smallest reflexive, transitive relation such that for every pair of conflicting events $e' \leq_{\sigma} e'$, we have $e \leq_{\text{CHB}} e'$. Atomicity is closely related to the property of conflict serializability. Informally, this property requires that an execution be serializable to a basic execution by commuting adjacent non-conflicting events; an execution is serial if for every transaction $T$ of thread $t$ in the execution, there are no events of any other threads between the begin and end events of $T$. In this context, if two events $e$ and $e'$ are ordered by $\leq_{\text{CHB}}$, then their order is the same in all equivalent executions. To capture conflict serializability, such a causal relationship needs to be lifted to transactions. Consider two transactions $T$ and $T'$ with events $e \in T$ and $e' \in T'$ such that $e \leq_{\text{CHB}} e'$. If the goal in a serial execution is to schedule all events of $T$ consecutively, given that $e$ is before $e'$ in all equivalent executions, it must be the case that all events of $T$ should happen before $T'$. Thus, transaction $T$ must happen before transaction $T'$ in trace $\sigma$ (denoted $T \leq_{\text{CHB}} T'$) if there are events $e \in T$ and $e' \in T'$ such that $e \leq_{\text{CHB}} e'$. We now present the definition of conflict serializability (which implies atomicity) from [14].

**Definition 1 (Conflict Serializability [14]):** A trace $\sigma$ is conflict serializable if there is no sequence of $k > 1$ distinct transactions $T_0, T_1 \ldots T_{k-1}$ such that $T_i \leq_{\text{CHB}} T_{(i+1)} \mod k$ for every $0 \leq i \leq k-1$.

**Example 1:** Consider the trace $\rho_1$ in Fig. 1. This trace is a sequence of 10 events, performed by three different threads $t_1, t_2$ and $t_3$. In all our examples, we will use $e_i$ to denote the $i$th event in the trace. This trace has three transactions — transaction $T_1 = e_1 e_2 e_5 e_10$ is performed in $t_1$, transaction $T_2 = e_3 e_4 e_5$ is performed in $t_2$ and transaction $T_3 = e_6 e_7 e_8$ is performed in $t_3$. Here, $(e_2, e_4)$ and $(e_7, e_9)$ are conflicting pairs of events; we depict this using an explicit arrow (→). In this trace, we have $T_1 \leq_{\text{CHB}} T_2$ because $e_2 \leq_{\text{CHB}} e_4$ and $T_3 \leq_{\text{CHB}} T_1$ because $e_7 \leq_{\text{CHB}} e_9$. Finally, this trace is conflict serializable and the equivalent serial execution is the sequence $\rho_{\text{serial}} = e_6 e_7 e_8 e_1 e_2 e_9 e_{10} e_3 e_4 e_5$, in which the order of transaction is $T_3 T_1 T_2$.

Based on Definition 1, a cyclic dependency on transactions using $\leq_{\text{CHB}}$ suggests that $\sigma$ does not have an equivalent serial execution and hence the program does not satisfy its atomicity specification. Previous techniques [3], [14] for checking conflict serializability dynamically, rely on constructing a directed graph, with transactions as vertices and $\leq_{\text{T}}$ as edges, and searching for a cycle in the graph. These algorithms run in time that quadratic in the length of the observed trace.

### III. Challenges in Designing a Vector Clock Algorithm

Vector clocks have been very useful in designing linear time algorithms for dynamic analysis of multi-threaded systems [11], [16], [19], [24], [30]. The broad principle behind these algorithms, is to assign vector timestamps to events as the trace is generated/observed in such a way that ordering between timestamps captures causal ordering. Since conflict serializability is defined in terms of the relation $\leq_{\text{T}}$ on transactions (Definition 1), the most straightforward vector clock algorithm would rely on assigning timestamps to transactions in such a way that the timestamp of $T_1$ is less than or equal to timestamp of $T_2$ if and only if $T_1 \leq_{\text{T}} T_2$. However, since a transaction is a sequence of events and not a single event, the first challenge is figuring out how to assign timestamps as individual events are processed; this is one of the reasons why such algorithms were deemed impossible for atomicity in [14]. However, there is a deeper and more fundamental challenge with assigning timestamps to transactions, as illustrated in the following example.

**Example 2:** Consider again the trace $\rho_1$ in Fig. 1. Notice that there is a “path” from $T_3$ to $T_2$ using $\leq_{\text{T}}$, even though $T_3$ starts after $T_2$ ends in the trace $\rho_1$. Further the discovery that $T_3$ has a path to $T_2$ can be made only after both $T_2$ and $T_3$ have ended. This poses serious challenges when designing a vector clock algorithm. A vector clock algorithm assigning a timestamp to transaction $T$ that is consistent with $\leq_{\text{T}}$ needs to know (explicitly or implicitly) the set of transactions that have a path to $T$; this is because the algorithm needs to ensure that the timestamp assigned to $T$ is after the timestamps assigned to all these “predecessor” transactions. However, as transaction $T_2$ in trace $\rho_1$ illustrates, this may require knowing future events and transactions.

Example 2 illustrates that transactions $T'$ that have a $\leq_{\text{T}}$-path to a transaction $T$ may only be determined by events that appear after $T$ itself. This suggests that one is unlikely to get a single-pass, streaming algorithm that assigns timestamps to transactions for detecting atomicity violations.

Therefore, we explore the possibility of an algorithm that assigns timestamps to events (not transactions), but which can nonetheless enable checking conflict serializability. The first key question to address is which relation among events should...
the timestamps try to capture implicitly. The relation $\prec_T$ is defined in terms of $\leq_{CHB}$, and therefore, a natural first step to explore, is to see if computing $\leq_{CHB}$ is sufficient to detect atomicity violations.

**Example 3:** Consider the trace $\rho_2$ in Fig. 2 with transactions $T_1$ and $T_2$ in threads $t_1$ and $t_2$ respectively. Here, we have, $e_1 \leq_{CHB} e_4$, and $e_5 \leq_{CHB} e_7$. If our algorithm assigns timestamps $C_e$ to each event $e$ in the trace and ensure that these timestamps respect these orderings, we could use the timestamps to identify that $C_{e_1}$ (timestamp of the begin of $T_1$) and $C_{e_5}$ (timestamp of $e_5$) are ordered, and also that $C_{e_5}$ and $C_{e_7}$ are ordered, thus giving us a $\leq_{CHB}$-path that starts and ends in the same transaction and passes through another transaction.

The atomicity violation in trace $\rho_2$ in Example 3 can be deduced based on the observation that there are 3 events $e, f, g$ ($e_1, e_5, e_7$ in $\rho_2$, specifically) such that $trans(e) = trans(g)$, $trans(e) \neq trans(f)$, and $e \leq_{CHB} f \leq_{CHB} g$. If we can prove that this equivalent to Definition 1, then all we need to do is to compute (implicitly using vector clocks) the $\leq_{CHB}$ ordering. Unfortunately, this is not true, i.e., violations of conflict serializability cannot be detected by simply using the $\leq_{CHB}$ ordering. This is illustrated by the next example.

**Example 4:** Consider trace $\rho_3$ in Fig. 3. Here, both $e_3 \leq_{CHB} e_6$ and $e_4 \leq_{CHB} e_5$. However, there is no $\leq_{CHB}$-path that starts and ends in the same transaction. If vector timestamps are used to compute $\leq_{CHB}$, then violations of conflict serializability cannot be detected by checking ordering of vector timestamps of events.

Example 4 demonstrates that $\leq_{CHB}$ is not the right relation on events to detect violations of conflict serializability. Then, what is the right relation to track? In order to identify that, we will first recast Definition 1 in terms of events.

We will say that there is a *path from event $e$ to $f$ through transactions* in trace $\sigma$ (denoted $e \rightarrow_{\sigma} f$), if there is a sequence of pairs $(e_1^1, e_2^1), (e_2^1, e_3^1), \ldots, (e_i^1, e_{i+1}^1)$ ($k > 1$) such that (a) $e = e_1^1$ and $f = e_k^1$, (b) $trans(e_1^1) = trans(e_2^1)$, while $trans(e_i^1) \neq trans(e_{i+1}^1)$, for every $i$, and (c) $e_i^1 \leq_{CHB} e_{i+1}^1$ for every $i < k$. Using the notion of path between events through transactions, we can recast the notion of conflict serializability as follows.

**Proposition 1:** A trace $\sigma$ is not conflict serializable if and only if there is a pair of events $e, f$ such that $e \rightarrow_{\sigma} f$ and $f \leq_{CHB} e$.

**Proof:** Follows from Definition 1 and the definition of $\rightarrow_{\sigma}$.

Though $\rightarrow_{\sigma}$ gives us a characterization of conflict serializability, it is not clear how to compute it algorithmically in a single pass over the trace. The reasons are technical and therefore, skipped. Instead, what we will compute is a slight restriction of the relation $\rightarrow_{\sigma}$, defined as follows.

**Definition 2:** For events $e, f$ in trace $\sigma$, we say $e \prec_{E} f$, if either (a) $e \leq_{CHB} f$, or (b) $e \rightarrow_{\sigma} f$ and $trans(e)$ is completed in $\sigma$.

Conflict serializability can be checked using this new relation on events as follows.

**Theorem 2:** A trace $\sigma$ is not conflict serializable if and only if there is a transaction $T$ and a pair of events $e, f$ such that $f \in T$, $T \sigma^* e$ and $e \prec_{E} f$, where $T \sigma^*$ is the begin transaction event $(\cdot, \cdot)$ of $T$.

**Proof:** ($\Rightarrow$) Observe that for any pair of events $e_1, e_2$, if $e_1 \prec_{E} e_2$ then $e_1 \rightarrow_{\sigma} e_2$. Thus, this direction just follows from Proposition 1.

($\Rightarrow$) Using Proposition 1, we can assume we have a sequence of pairs of events $(e_0^1, e_1^2), \ldots, (e_{k-1}^1, e_{k}^2)$ such that $trans(e_0^1) = trans(e_1^2)$ but $trans(e_i^1) \neq trans(e_{i+1}^1)$ mod $k$, and $e_i^2 \leq_{CHB} e_{i+1}^2$ mod $k$.

Let $T_i = trans(e_i^1) = trans(e_{i+1}^2)$. Without loss of generality, we may assume that each $T_i$ is distinct, and that all transactions, except possibly $T_0$, are completed in $\sigma$. Let $g$ be the begin transaction event of $T_0$, and take $e = e_1^1$ and $f = e_2^2$. Observe that $g \prec_{E} e$, and $e \prec_{E} f$, satisfying conditions of the theorem.

We conclude this section with examples illustrating both the definition $\prec_E$ and the use of Theorem 2.

**Example 5:** Let us begin by looking at trace $\rho_3$ in Fig. 3. Let $\sigma_6$ denote the prefix of $\rho_3$ upto (and including) event $e_i$.

In trace $\sigma_6$, we have $e_3 \prec_{E} e_6$, $e_4 \prec_{E} e_5$, and $e_6 \prec_{E} e_6$ because they are related by $\leq_{CHB}$. Here, $e_1 \rightarrow_{\sigma_6} e_3$ because $trans(e_1) = trans(e_3)$, $e_3 \leq_{CHB} e_6$ and $trans(e_6) = trans(e_3)$. However, it is not the case that $e_1 \prec_{E} e_4$. On the other hand, if we consider $\sigma_7$, then $e_1 \prec_{E} e_4$ as the transaction in $t_1$ is complete in $\sigma_7$. In $\sigma_7$ (and therefore also in the full trace $\rho_3$), conditions of Theorem 2 are satisfied $e_1 \prec_{E} e_4$ and $e_4 \prec_{E} e_7$.

**Example 6:** Consider trace $\rho_4$ in Fig. 4; this is a slight modification of trace $\rho_1$ from Fig. 1 that now has an atomicity
violation. Again $e_i$ denotes the $i^{th}$ event, and $\sigma_i$ denotes the prefix upto event $e_i$. Notice that in prefix $\sigma_{11}$, $e_1 \leq_{E} e_5$ (because $e_1 \leq_{E}^{CHB} e_5$) and $e_5 <_{E} e_{11}$ (because $e_5 \rightarrow_{\sigma_{11}} e_{11}$ and trans$(e_5)$ is complete in $\sigma_{11}$). Thus by Theorem 2, there is a violation of conflict serializability.

IV. VECTOR CLOCK ALGORITHM

Based on intuitions developed in Section III, we will now describe our vector clock based algorithm called AeroDrome, for checking violations of conflict serializability. Before presenting the algorithm itself, we recall some notation and concepts related to vector clocks that will be useful.

Let us fix the set of threads in the trace/program to be Thr. A vector time (or timestamp) is a vector of non-negative integers, whose size/dimension is $|\text{Thr}|$ (number of threads). For a thread $t \in \text{Thr}$, we denote the $i^{th}$ component of a vector time $V$ by $V(t)$. We say a vector time $V_1$ is less than another vector time $V_2$ (of the same dimension), denoted $V_1 \preceq V_2$ if $\forall t \in \text{Thr}, V_1(t) \preceq V_2(t)$. The minimum vector time on threads Thr is $\top_{\text{Thr}} = \lambda \cdot 0$, and we will often use $\bot$ when Thr is clear from context. Next, the join of two vector times $V_1$ and $V_2$ is the time $V_1 \sqcup V_2 = \lambda \cdot \max\{V_1(t), V_2(t)\}$. Finally, we use $V[e/t]$ to denote the timestamp $\lambda u$. if $u = t$ then $e$ else $V(u)$. Vector clocks are variables (or place holders) for vector timestamps. That is, vector clocks are variables that take values from the space of vector time, and will be used in our algorithm to compute the timestamps associated with various events in a trace. All the operations on vector times can be naturally thought of as applying to vector clocks as well.

A. The AeroDrome Algorithm

Similar to other vector clock algorithms, our algorithm will maintain a local vector clock for each thread. AeroDrome processes each event in the trace sequentially and (implicitly) assigns vector timestamps to each event in the trace $\sigma$. Broadly, the goal of the algorithm will be to assign vector timestamps that capture the relation $\leq_{E}$ (Definition 2) and use Theorem 2 to discover conflict serializability violations. The exact invariant maintained by the algorithm is technical and is postponed to later.

Pseudocode for AeroDrome is shown in Algorithm 1. It processes events in the trace based on their operation, calling the appropriate handler. It maintains a number of vector clocks, and we will depict these variables using the black-board font — C, L, W, R, etc. Let us assume for now that every event in the trace is part of some transaction, and that transactions are not nested; later in this section, we will describe how to handle nested transactions and unary transactions, i.e., events not enclosed within a begin and end atomic block.

1) Vector Clocks and Other Data in the State: As mentioned before, the algorithm will maintain a vector clock $C_t$ for each thread $t$, which stores the “local time” of thread $t$. The algorithm will check for violations of conflict serializability using Theorem 2, which requires checking a property of the begin event of a transaction. Therefore, in addition to the clock $C_t$, we will also maintain a clock $C^D_t$ which will store the local time of thread $t$ when the last begin event was performed by thread $t$.

The goal of these vector timestamps is to capture the relation $\leq_{E}$. Since $\leq_{E}$ is defined using $\leq_{\text{CHB}}$, we need to ensure that the vector timestamps are consistent with $\leq_{\text{CHB}}$. The fact that events of the same thread are ordered in all equivalent executions, is captured by ensuring that the vector clock $C_t$ of each thread $t$, monotonically increases. However, to capture other dependencies of the $\leq_{\text{CHB}}$ relation, we need auxiliary clocks. Consider an event $e$ of the form $(t, \text{acq}(\ell))$. All previously encountered events with operations on lock $\ell$ are $\leq_{\text{CHB}}$ before $e$. Hence the timestamp of $e$ must be after those assigned to such events. To do this, AeroDrome will maintain a vector clock $L_\ell$ for each lock $\ell$, that stores the timestamp of the last $\text{rel}(\ell)$ seen so far; this will be read to ensure that the timestamp of $e$ is appropriately larger. Similarly, we need to ensure that the timestamp of every write event is after the timestamp of all previous writes and reads to the same variable, and that of a read event is after the timestamp of previous writes. Therefore, for every variable $x$, AeroDrome has a clock $W_x$ that stores the timestamp of the last write $w(x)$-event and a clock $R_{x,\ell}$ that stores the time of the last $(t, x)$-event. Notice that when considering paths between events through transactions (\rightarrow), we need to make sure that successive transactions along the path are different. To be able to track this constraint, AeroDrome will also maintain scalar variables lastRelThr and lastWThr, which store the identifier of the thread that performed the last release on $\ell$ and write on $x$, respectively. Each of the clocks $C_t$ are initialized with the time $\bot_{[1/t]}$, all other clocks are initialized to $\bot$, and all the scalar variables are initialized to a default value of $\text{NIL}$.

2) Updates to the State: As new events are observed in the trace, the algorithm updates these vector clocks in manner that is consistent with tracking the $\leq_{E}$-relation.

When processing a begin event $e = (t, \beta)$, the algorithm first increments the local component of $C_t$ (line 35 - $C_t := C_t[C_t(t) + 1]$). To understand why, let $T$ be the transaction started by $e$ and $T^\prime$ be the previous transaction (if any) by the same thread $t$. Further, let $T^\prime$ be some transaction such that $T^\prime$ has a $\sim_T$-path to $T^\prime$, but say $T$ does not have a $\sim_T$-path
Algorithm 1: AeroDrome: Vector Clock Algorithm for Checking Violation of Conflict Serializability

1: procedure Initialization
2:     for \( t \in \text{Threads} \) do
3:         \( C_t := \bot [1/t]; C_t^\triangledown := \bot; \)
4:     for \( \ell \in \text{Locks} \) do
5:         \( L_\ell := \bot; \text{lastRelThr}_\ell := \text{NIL}; \)
6:     for \( x \in \text{Vars} \) do
7:         \( \mathcal{W}_x := \bot; \text{lastWThr}_x := \text{NIL}; \)
8:     for \( t \in \text{Threads} \) do \( \mathcal{R}_{t,x} := \bot; \)
9:     procedure CHECKANDGET(clk, t)
10:         if \( C_t^\triangledown \subseteq \text{clk} \) and \( t \) has an active transaction then
11:             declare ‘conflict serializability violation’;
12:             \( C_t := C_t \cup \text{clk}; \)
13:     procedure ACQUIRE(t, \( \ell \))
14:         if \( \text{lastRelThr}_\ell \neq t \) then
15:             \( \text{CHECKANDGET}(L_\ell, t); \)
16:     procedure RELEASE(t, \( \ell \))
17:         \( L_\ell := C_t; \)
18:         \( \text{lastRelThr}_\ell := t; \)
19:     procedure FORK(t, u)
20:         \( C_u := C_t \cup C_t; \)
21:     procedure JOIN(t, u)
22:     \( \text{CHECKANDGET}(C_u, t); \)
23:     procedure READ(t, x)
24:         if \( \text{lastWThr}_x \neq t \) then
25:             \( \text{CHECKANDGET}(\mathcal{W}_x, t); \)
26:             \( \mathcal{R}_{t,x} := C_t; \)
27:     procedure WRITE(t, x)
28:         if \( \text{lastWThr}_x \neq t \) then
29:             \( \text{CHECKANDGET}(\mathcal{W}_x, t); \)
30:     procedure BEGIN(t)
31:         \( C_t(t) := C_t(t) + 1; \)
32:         \( C_t^\triangledown := C_t; \)
33:     procedure END(t)
34:     for \( u \in \text{Threads} \setminus \{t\} \) do
35:         if \( C_t^\triangledown \subseteq C_u \) then
36:             \( \text{CHECKANDGET}(C_t, u); \)
37:     for \( \ell \in \text{Locks} \) do
38:         \( L_\ell := C_t^\triangledown \subseteq L_\ell \cup C_t \cup L_\ell \cup L_\ell; \)
39:     for \( x \in \text{Vars} \) do
40:         \( \mathcal{W}_x := C_t^\triangledown \subseteq \mathcal{W}_x \cup \mathcal{W}_x; \)
41:     for \( u \in \text{Threads} \) do
42:         \( \mathcal{R}_{u,x} := C_t^\triangledown \subseteq \mathcal{R}_{u,x} \cup \mathcal{R}_{u,x} \cup \mathcal{R}_{u,x}; \)

to \( T' \). The increment of the local component ensures that this relationship between \( T, T_{prev} \) and \( T' \) can be accurately inferred from the timestamps of the events in these transactions — by ensuring that the local components of events in \( T' \) is different from the events in \( T_{prev} \). Finally, AeroDrome updates \( C_T^\triangledown \) with the timestamp of the current event \( e \) stored in \( C_t \).

When processing an acquire event \( e = \langle t, \text{acq}(\ell) \rangle \), the algorithm makes sure that the timestamp of \( e \) is ordered after the last release event \( e_\ell \) of lock \( \ell \). This is achieved by updating ‘\( C_t := C_t \cup L_\ell \)’ in the procedure CHECKANDGET when invoked at line 15; the procedure CHECKANDGET also checks for conflict serializability violation before updating \( C_t \), but more on that later. Of course, if \( e_\ell \) is performed by the same thread \( t \) (condition in line 14), then, this is already ensured and no explicit update is required.

At a write event \( e = \langle t, \text{w}(x) \rangle \), AeroDrome ensures that the timestamp of \( e \) is ordered after all the prior reads and writes on \( x \) by calling CHECKANDGET in lines 29 and 31. The algorithm then updates \( \mathcal{W}_x \) to be the timestamp of \( e \) (see line 32) and \( \text{lastWThr}_x \) to \( t \), thus preserving the semantics of the clock \( \mathcal{W}_x \) and the scalar variable \( \text{lastWThr}_x \). The updates performed at a read event are similar.

At a fork event \( e = \langle t, \text{fork}(u) \rangle \), the algorithm updates the clock of the child thread \( u \) (‘\( C_u := C_t \cup C_t^\triangledown \); line 20) so that all events of \( u \) are ordered after \( e \). At a join event \( e = \langle t, \text{join}(u) \rangle \), the algorithm updates \( C_t \) to \( C_t \cup C_u \) so that all events of thread \( u \) are ordered before \( e \).

Let us now consider the updates performed at an end-transaction event \( e = \langle t, \triangledown \rangle \). Let \( e^\triangledown \) denote the matching begin transaction event. Observe that if for an event \( f, e^\triangledown \preceq_E f \), then since \( \text{trans}(e) \) is completed in \( f, e^\triangledown \preceq f \). Thus, all future events that are \( e^\triangledown \)-after \( e^\triangledown \) must be assigned a timestamp after that of \( e \). This is ensured by updating \( C_u \) for all threads \( u \) with \( C_t^\triangledown \subseteq C_u \) (lines 38-40), and clocks \( L_\ell, \mathcal{W}_x \), and \( \mathcal{R}_{u,x} \) (lines 41-46).

3) Checking Violations of Atomicity: The algorithm detects violations of atomicity at various points by a call to the procedure CHECKANDGET. The checks can be broadly classified into two categories. First, the algorithm can report a violation at an event \( e = \langle t, \triangledown \rangle \) such that there is an earlier event \( e' \) (performed by a thread \( t' \neq t \)) that conflicts with \( e \). In this case, if \( e^\triangledown \preceq_E e' \) (where \( e^\triangledown \) is the begin event of \( \text{trans}(e) \)), then conditions in Theorem 2 are satisfied to demonstrate a violation. This check is performed at acquire events (line 15), at read events (line 25) and at write events (lines 29 and 31). Second, the algorithm reports atomicity violations when processing an end event \( e = \langle t, \triangledown \rangle \) (with a matching begin event \( e^\triangledown \)). The algorithm detects a violation when there is another thread \( u \neq t \) with an active transaction whose begin event is \( e_u^\triangledown \) and the last event \( e_u \), such that \( e_u^\triangledown \preceq_E e \) and \( e^\triangledown \preceq_E e_u \).
two threads and thus the size of each vector clock is if its value has changed after processing the begin event of the (active) transaction (line 10). It then updates the clock \( C_i \) to be \( C_i \cup \text{clk} \) (line 12).

4) Nested and Unary Transactions: Let us now consider the cases of nested and unary transactions that we postponed. In the case of nested transactions, it is enough to only consider the outermost transactions and ignore the inner transactions. This is because there if there is a cycle involving a transaction \( T \) that is nested inside another transaction \( T' \), then there is clearly also a cycle involving \( T' \). As a result, we simply ignore the begin and end events that have a non-zero nesting depth.

Events that are not enclosed by begin and end transaction events constitute a trivial atomic block, namely, one consisting of only that single event. These are called unary transactions. Notice that a unary transaction corresponding to a read, write, acquire or join event can only correspond to a cycle that involves another non-unity transactions. Our algorithm, in fact, does not detect a violation at these unary transactions (in the procedure \textsc{CheckAndGet}, line 10) as unary transactions are not active transactions.

We conclude this section with a theorem stating the correctness of Algorithm 1, whose proof sketch is presented in the Appendix.

**Theorem 3:** On any trace \( \sigma \), Algorithm 1 reports a violation of conflict serializability iff \( \sigma \) is not conflict serializable.

### B. AeroDrome on Example Traces

Let us illustrate AeroDrome’s workings on the traces from Section III. Eventhough these examples do not use any synchronization primitives like locking, they contain all the features needed to highlight the subtle aspects of AeroDrome.

Let us begin with the simplest trace \( \rho_2 \) from Fig. 2. We show the values of the relevant vector clocks in Fig. 5. In this figure, we only depict the value of a vector clock in row \( i \) if its value has changed after processing the \( i \)th event \( e_i \) in the trace. We do not show the values of the clocks \( \mathbb{R}_{t_1,x}, \mathbb{R}_{t_2,x}, \mathbb{R}_{t_1,y}, \mathbb{R}_{t_1,y} \) as they are not important here. There are two threads and thus the size of each vector clock is 2. The clocks \( C_{t_1} \) and \( C_{t_2} \) are initialized to the timestamps \( (1,0) \) and \( (0,1) \) respectively, and all other clocks are initialized to \( \bot=(0,0) \). The local clocks increment after a begin event (line 35 in Algorithm 1) and thus the clocks \( C_{t_1} \) and \( C_{t_2} \) become \( (2,0) \) and \( (2,0) \) after \( e_2 \). Further, these are also the values of the clocks \( C_{t_1}^p \) and \( C_{t_2}^p \) from this point onwards until the end of the execution. After processing \( e_3 = (t_1,w(x)) \), the value of the clock \( \mathbb{W}_x \) becomes \( (2,0) \) (line 32). At event \( e_4 \), the call to \textsc{CheckAndGet} (see line 25) with arguments \(( (2,0), t_2) \) updates the clock \( C_{t_2} \) to \((2,2) \) (line 12). The clock \( \mathbb{W}_y \) gets updated to \( C_{t_2}=(2,2) \) after processing \( e_5 \). Finally, at event \( e_6 \), the algorithm calls \textsc{CheckAndGet} with arguments \(( (2,2), t_1) \). In this procedure, the algorithm asserts that \( C_{t_1}^p \subseteq \mathbb{W}_y \) and declares an atomicity violation.

Let us next consider the trace \( \rho_3 \) from Fig. 3. AeroDrome’s run on this trace is shown in Fig. 6. Updates corresponding to the first four events are straightforward. In event \( e_5 \), \( C_{t_1} \) gets updated to \( (2,2) \) because of the call to \textsc{CheckAndGet} in line 25. Notice that this call does not raise any violation of atomicity because at this point, \( C_{t_1}^p = (2,0) \) and the clock \( \mathbb{W}_y \) is \( (0,2) \) thus failing the check \( C_{t_1}^p \subseteq \mathbb{W}_y \) in line 10. The same explanation applies to the \( r(x) \) event \( e_6 \) in \( t_2 \) and thus no atomicity violation is reported here as well. Next, the algorithm processes the end event \( e_7 = (t_1, \bot) \). At this point, the algorithm checks if \( t_2 \) knows the clock \( C_{t_1}^p \) (corresponding to the check \( C_{t_1}^p \subseteq C_{t_2} \) in line 39 of Algorithm 1). This check succeeds since \( C_{t_2}=(2,0) \) and \( C_{t_1}=(2,2) \) at this point. The algorithm then checks if \( C_{t_2} \subseteq C_{t_1} \) in the procedure \textsc{CheckAndGet} and thus declares an atomicity violation. This illustrates the subtlety in how the algorithm reports atomicity violations at an end event.

We will now illustrate how Algorithm 1 detects the atomicity violation in the more involved trace \( \rho_1' \) from Fig. 4. This example illustrates how AeroDrome handles dependencies between transactions introduced by future events. The run of AeroDrome on \( \rho_1' \) is shown in Fig. 7. We omit the updates to the clocks \( \mathbb{R}_{t_1,u} \) \( (i \in \{1,2,3\}, u \in \{x,y,z\}) \) as they do not play a significant role in this example. All vector clocks have dimension 3 because there are three threads in \( \rho_1' \). As before, the clocks are initialized as follows: \( C_{t_2}=(1,0,0), C_{t_3}=(0,1,0), C_{t_4}=(0,0,1) \); all other clocks are initialized to \( (0,0,0) \). The begin events result in incrementing of local clocks and thus \( C_{t_1}=(2,0,0) \) after \( e_1 \). Further, the clock \( \mathbb{W}_z \) gets updated to the value of \( C_{t_1} \) at the end of \( e_2 \). The next two events \( e_3 \) and \( e_4 \) are processed in a similar fashion. At event \( e_5= (t_2, r(x)) \), the clock \( \mathbb{W}_y \) gets updated to \( (2,2,0) \) as in line 12. After this, the transaction in \( t_2 \) ends. The clocks of none of the threads is updated because of \( e_6 \).
thread $t_1$ nor $t_2$ have clock values larger than $C^\rho_{t_2}$ (line 38-40). However the write and read clocks are updated. Specifically, the clock $W_y$ maintaining the timestamp to the last write to $y$ is such that $C^\rho_{t_2} \sqsubseteq W_y$ and thus, the algorithm updates $W_y$ to $W_y \cup C_{t_2} = \langle 2, 2, 0 \rangle$ (line 44 in Algorithm 1). Event $e_7$ is a begin event and updates $C_{t_3}$ to $\langle 0, 0, 2 \rangle$. Now at the $r(y)$ event $e_8$, the clock $C_{t_3}$ gets updated with $W_y$ which at this point evaluates to $\langle 2, 2, 0 \rangle$, thus giving $C_{t_3} = \langle 2, 2, 2 \rangle$. The write clock $W_z$ then gets updated to $\langle 2, 2, 2 \rangle$ after $e_9$. Further, significant clock updates are performed at the end event $e_{10}$. Finally, the algorithm detects an atomicity violation at event $e_{11} = \langle t_1, r(x) \rangle$ — the algorithm checks if the clock $W_z$ know some event in $t_1$ ($C^\rho_{t_1} \sqsubseteq W_z$) and concludes that there is a violation of conflict serializability as this check passes.

C. Reducing the number of Read Clocks

Recall that Algorithm 1 maintains, a vector clock $R_{t,x}$ for every pair of thread $t$ and memory location $x$. Therefore, the number of such vector clocks that need to be tracked in the basic algorithm is $O(TV)$, where $T$ is the number of clocks and $V$ is the number of memory locations. Storing and updating these many clocks can be expensive, when the number of memory locations that need to be tracked is prohibitively large, as is the case for most real world software. We tackle this using our optimization to reduce the number of clocks from $O(TV)$ to $O(V)$. The role of the clocks $R_{t,x}$ is two-folds. First, these clocks help detect atomicity violation — at a write event $e = \langle t, w(x) \rangle$, the algorithm checks if there is a thread $u \neq t$ such that $C^\rho_u \sqsubseteq R_{u,x}$ (line 10 in Algorithm 1). Second, these clocks are used to update $C_t$ — at a write event $e = \langle t, w(x) \rangle$, we set $C_t := \bigcup_{u, x} C_t \cup R_{u,x}$ (line 12 called iteratively in the loop at line 30).

The reduction in clocks is achieved by instead maintaining a single clock (per memory location) for each of the above two purposes instead of maintaining $O(T)$ many clocks (per memory location). First, for updating clocks correctly at write events, we will maintain a single clock $R_x$ for each location $x$. This clock stores the value $\bigcup_u R_{u,x}$ at each point while processing the trace. Next, to perform checks for violations of conflict serializability, we will have another clock $\exists_1 R_x$ (check read). This clock will store the value $\bigcup_u R_{u,x}[0/u]$ at each point in the analysis. Based on the invariants maintained by the algorithm, one can show that checking $C^\rho_u \sqsubseteq \bigcup_{x \in T} R_{u,x}$ is equivalent to checking $C^\rho_u \sqsubseteq \exists_1 R_x$. In the Appendix, we present this optimization in greater detail and also outline other useful optimizations that improve the performance of AeroDrome.

We now state the time and space complexity for the optimized version discussed in this section. We will use $n_{\text{non-end}}$ and $n_{\text{end}}$ as the number of non-end events and end events in the trace (and therefore $n = n_{\text{non-end}} + n_{\text{end}}$ is the size of the trace). We will denote by $T$, $V$ and $L$ the number of threads, memory locations and locks in the input trace. Further, all arithmetic operations are assumed to take constant time.

Theorem 4: The algorithm takes $O(T(n_{\text{non-end}} + (T + L + V)n_{\text{end}}))$ time and $O(T(T + V + L))$ space.

The complexity observations easily follow from the description of the algorithm.

V. EXPERIMENTAL EVALUATION

In this section, we describe our implementation of AeroDrome, and the results of evaluating it on benchmark programs.

A. Implementation and Setup

We have implemented AeroDrome in our tool AIRTRACK. AIRTRACK is written in Java and analyzes traces generated by concurrent programs and dynamically checks for atomicity using violations of conflict serializability. We use RoadRunner [12] to log traces from our set of benchmark programs. RoadRunner logs read and write accesses to memory locations, acquire and release of synchronization objects (locks), forks and joins of threads and also events generated at the entry and exit of each method. We also implement the Velodrome algorithm [14] in AIRTRACK. Recall that the Velodrome analysis relies on building a directed graph, with transactions as nodes in the graph and where the edges correspond to $\lessdot_T$ relation between transactions. We use the Java graph library JGraphT [26] to implement Velodrome, including updating the graph and detecting cycles. In our implementation of Velodrome, we also incorporate garbage collection as an optimization suggested in [14] — transactions with no incoming edges do not participate in cycles and can be deleted from the
TABLE I
TRACE CHARACTERISTICS AND RUNNING TIMES FOR BENCHMARKS WITH NAIVE ATOMICITY SPECIFICATIONS.

<table>
<thead>
<tr>
<th>Program</th>
<th>Events</th>
<th>Threads</th>
<th>Locks</th>
<th>Vars.</th>
<th>Txns.</th>
<th>Atomic?</th>
<th>Velodrome (s)</th>
<th>AeroDrome (s)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>batik</td>
<td>186M</td>
<td>7</td>
<td>1916</td>
<td>4.9M</td>
<td>15M</td>
<td>X</td>
<td>52.7</td>
<td>65.5</td>
<td>0.81</td>
</tr>
<tr>
<td>crypt</td>
<td>126M</td>
<td>7</td>
<td>1</td>
<td>9M</td>
<td>50</td>
<td>X</td>
<td>92.1</td>
<td>104</td>
<td>0.88</td>
</tr>
<tr>
<td>fop</td>
<td>96M</td>
<td>1</td>
<td>115</td>
<td>5M</td>
<td>25M</td>
<td>✓</td>
<td>88.3</td>
<td>92.5</td>
<td>0.95</td>
</tr>
<tr>
<td>lafact</td>
<td>135M</td>
<td>4</td>
<td>1</td>
<td>252K</td>
<td>642M</td>
<td>X</td>
<td>2.4</td>
<td>2.9</td>
<td>0.82</td>
</tr>
<tr>
<td>series</td>
<td>40M</td>
<td>4</td>
<td>1</td>
<td>20K</td>
<td>20M</td>
<td>X</td>
<td>61.0</td>
<td>15.3</td>
<td>3.98</td>
</tr>
<tr>
<td>sparsematmult</td>
<td>726M</td>
<td>4</td>
<td>1</td>
<td>1.6M</td>
<td>25</td>
<td>X</td>
<td>1210</td>
<td>1197</td>
<td>1.01</td>
</tr>
<tr>
<td>tomcat</td>
<td>726M</td>
<td>4</td>
<td>1</td>
<td>1.6M</td>
<td>25</td>
<td>X</td>
<td>3.4</td>
<td>4.5</td>
<td>0.75</td>
</tr>
</tbody>
</table>

TABLE II
TRACE CHARACTERISTICS AND RUNNING TIMES FOR BENCHMARKS WITH ATOMICITY SPECIFICATIONS FROM DOUBLECHECKER.

<table>
<thead>
<tr>
<th>Program</th>
<th>Events</th>
<th>Threads</th>
<th>Locks</th>
<th>Vars.</th>
<th>Txns.</th>
<th>Atomic?</th>
<th>Velodrome (s)</th>
<th>AeroDrome (s)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>2.4B</td>
<td>7</td>
<td>7</td>
<td>1079K</td>
<td>498M</td>
<td>X</td>
<td>TO</td>
<td>1.5</td>
<td>&gt;24000</td>
</tr>
<tr>
<td>luindex</td>
<td>570M</td>
<td>3</td>
<td>65</td>
<td>2.5M</td>
<td>86M</td>
<td>X</td>
<td>581</td>
<td>674</td>
<td>0.86</td>
</tr>
<tr>
<td>lusearch</td>
<td>2.0B</td>
<td>14</td>
<td>772</td>
<td>38M</td>
<td>306M</td>
<td>X</td>
<td>TO</td>
<td>5.5</td>
<td>&gt;650</td>
</tr>
<tr>
<td>moldyn</td>
<td>1.7B</td>
<td>4</td>
<td>1</td>
<td>121K</td>
<td>1.4M</td>
<td>X</td>
<td>TO</td>
<td>54.9</td>
<td>&gt;650</td>
</tr>
<tr>
<td>montecarlo</td>
<td>494M</td>
<td>4</td>
<td>1</td>
<td>30.5M</td>
<td>812K</td>
<td>X</td>
<td>TO</td>
<td>0.75</td>
<td>&gt;48000</td>
</tr>
<tr>
<td>pmd</td>
<td>367M</td>
<td>13</td>
<td>223</td>
<td>12.9M</td>
<td>81M</td>
<td>X</td>
<td>3.1</td>
<td>3.8</td>
<td>0.82</td>
</tr>
<tr>
<td>raytracer</td>
<td>2.8B</td>
<td>4</td>
<td>1</td>
<td>12.6M</td>
<td>277M</td>
<td>✓</td>
<td>TO</td>
<td>55m40s</td>
<td>&gt;10.7</td>
</tr>
<tr>
<td>sor</td>
<td>608M</td>
<td>4</td>
<td>2</td>
<td>1M</td>
<td>637K</td>
<td>X</td>
<td>6.9</td>
<td>9.6</td>
<td>0.72</td>
</tr>
<tr>
<td>sunflow</td>
<td>16.8M</td>
<td>16</td>
<td>9</td>
<td>1.2M</td>
<td>2.5M</td>
<td>X</td>
<td>67.9</td>
<td>0.65</td>
<td>104.5</td>
</tr>
<tr>
<td>xalan</td>
<td>1.0B</td>
<td>13</td>
<td>8624</td>
<td>31M</td>
<td>214M</td>
<td>X</td>
<td>1.6</td>
<td>2.0</td>
<td>0.8</td>
</tr>
</tbody>
</table>

We analyze AeroDrome and Velodrome on the same trace (generated by RoadRunner) to ensure a fair comparison. The tool DoubleChecker [3] also implements a graph based algorithm, similar to Velodrome. However, a comparison was not possible since the tool does not support analysis directly on concurrent program traces.

B. Benchmarks, Setup and Atomicity Specifications

In general, the logging mechanism in RoadRunner instruments and tracks all events corresponding to entering and exiting methods. A naïve atomicity specification would be to mark all method boundaries as atomic. However, as expected, not all methods are intended to be atomic. For example, default methods like run in Java, or the static main methods are often not intended to be atomic. Thus, atomicity specifications need to be specially identified by developers, by supplying manual annotations [15]. In the absence of such static annotations, we use atomicity specifications from prior work [3] whenever possible (Table II). For the benchmarks (Table I) for which no specifications were available, we declare all methods except the main and run methods to be atomic.

Our benchmark programs (Table I and Table II) are derived from the DaCaPo benchmark suite [4] and Java Grande Forum [34]. Our experiments were conducted on a 2.6GHz 64-bit Linux machine with Java 1.8 as the JVM and 30GB heap space. In each table, Column 1 depicts the name of the benchmark. Column 2 reports the number of events in the trace generated from the corresponding benchmark program and Column 1. Column 3, 4 and 5 report the number of distinct threads, locks and variables accessed in the trace generated. Column 6 reports the number of transactions in the trace. Column 7 reports ‘X’ if an atomicity violation was detected and reports ‘✓’ otherwise. Columns 8 and 9 report the time (in seconds) taken by respectively the Velodrome algorithm and AeroDrome introduced in this article to analyze the trace generated. A ‘TO’ in Column 8 represents timeout after 10 hours. Column 10 reports the speed-up of AeroDrome over Velodrome.

C. Evaluation Results

The first set of benchmarks (Table I) is evaluated against default atomicity specifications (all methods except main and run are assumed to be atomic), while in the second set of benchmarks (Table II) we use curated atomicity specifications from [3].

In the first set, we notice that the performance of our algorithm AeroDrome is comparable to that of Velodrome. This is expected because the atomicity specifications are inadequate and do not reflect realistic ones — typically most methods are non-atomic and developers have to identify a smaller set of
candidate code blocks that they think are atomic. As a result, on these benchmarks, violations are detected early on in the trace and thus, the size of the transaction graph in Velodrome’s analysis is small. A detailed analysis of the traces suggests that in all these benchmarks, the number of transactions did not grow more than 4, except for tomcat, for which the size of the graph grows to 21. In this case, the cost of maintaining vector clocks and updating them at every event overrides their potential benefits.

Let us now consider the second set of benchmarks from Table II. On most of these benchmarks, the violations of atomicity are discovered late in the trace. This is expected as the specifications are realistic and do not declare all methods to be atomic. The performance of AeroDrome is significantly better than that of Velodrome. Velodrome times out on most of these benchmarks (time limit was set to be 10 hours). This is because of the prohibitively large number of transactions that get accumulated in these traces. Consider, for example, the case of sunflow for which AeroDrome takes less than a second, while Velodrome spends about 68 seconds. Here, the number of nodes in the graph analyzed by Velodrome is about 9000. This coupled with the quadratic runtime complexity, results in the notable slowdown. Notice that, the slowdown is despite the garbage collection optimization implemented in Velodrome. Our algorithm, on the other hand has linear running time. Similarly, in the benchmark avrora, the number of transactions is more than 393K in the prefix of the trace in which AeroDrome reports an atomicity violation. Any super linear time analysis is unlikely to scale for so many transactions, and Velodrome, in fact, does not return an answer within 10 hours. AeroDrome, on the other hand, scales to traces with more than a billion events (avrora, lusearch, moldyn, raytracer,xalan) and demonstrates the effectiveness of a linear time vector clock algorithm. For the examples on which the performance of Velodrome is comparable, we discovered that the number of nodes in Velodrome’s graph analysis is fairly small — 13 nodes in pmd, 4 nodes in sor and 13 nodes in xalan.

VI. RELATED WORK

Multi-threaded programs are challenging to write and reason about. Atomicity is a principled concept that lets programmers reason about coarse behaviors of programmers, without being concerned about fine grained thread interleavings. Ensuring atomicity of concurrent program blocks is therefore an important question [22] and has been investigated thoroughly.

Static analysis techniques analyze source code to confirm the atomicity of code blocks marked atomic. Such techniques prominently rely on the design type of systems [13], [15]. These type systems rely on commutativity of operations and are inspired from Lipton’s theory of reduction [20] and the concept of purity [9]. Extensions to type inference [32] and to programs with non-blocking synchronization [37] have been developed. The work in [13] uses constraint based type system inference for inferring atomicity specifications.

There is a rich literature on dynamically checking atomicity. Lipton’s theory of reduction [20] has been a prominent theme in this space, most notably the analysis in Atomizer [8]. The notion of conflict serializability was introduced concurrently by Flanagan et al [14] and Farzan et al [7], with subtle differences on how synchronization events need to be ordered. In particular, Farzan et al [6] do not consider any lock operations in their traces and can lead to both false positives and false negatives. Further, their algorithm relies on maintaining sets of locks, threads and variables, similar to the use of locksets in Goldilocks [5] algorithm for employing HB for race detection, and similar to the case in race detection [11], [24], such an algorithm is expected to be orders of magnitude slower than a vector clock algorithm. We refer the reader to [3] for a more thorough analysis of the differences in the two approaches. Conflict serializability has been inspired from the theory of concurrency control in databases [27]. Recently, DoubleChecker [3] proposed a two-pass analysis for efficient detection of conflict serializability violations. Here, a coarse first pass detects potential cycles in the transaction graph. This is followed by a fine grained analysis that tracks more information and ensures the soundness of the overall analysis. Causal atomicity [6] is a slightly different serializability condition and asks for an equivalent trace where a particular transaction is serial.

As with most concurrency bugs, detecting atomicity violations is a challenging problem and is subject to interleaving explosion problem. Techniques such as that in CTrigger [29] and AVIO [21] resort to directed exploration of thread interleavings to expose subtle atomicity violations. Penelope [35] detects 2 thread atomicity violations using directed interleaving exploration. The work in [2], [23], [38], [39] is also based on exercising specific thread schedules. SMT solving based predictive analysis techniques [36] have been developed, but tend to not scale. The work of Samak et. al. [31] synthesizes directed unit tests for catching atomicity violations. The work in [6], [33] develop techniques for model checking concurrent programs for exposing atomicity violations. The use of random sampling and thread scheduling have also been proposed previously in the literature [18], [28]. Jin et. al. [17] study the problem of synthesizing bug patches for fixing atomicity bugs.

VII. CONCLUSIONS

In this paper, we considered the problem of checking atomicity in concurrent programs. Conflict serializability of traces is a popular notion for checking atomicity dynamically. We present, in this paper, the first linear time, vector clock based algorithm that we have developed, for checking violations of conflict serializability on traces of concurrent programs. Our experimental evaluation demonstrates the power of a linear time algorithm, in that, it scales well to large executions and is often faster than existing graph based algorithms. Interesting avenues for future work include extending the insights developed in our paper to design efficient algorithms for other notions of atomicity, including causal atomicity [6], view serializability [38] or reduction based atomicity characterizations as in [8], [39].


APPENDIX A
CORRECTNESS OF AERO DROME

We now prove that Algorithm 1 reports a violation on a trace \( \sigma \) if and only if \( \sigma \) is not conflict serializable (as per Definition 1). The key is to identify the invariant being maintained by the algorithm. Intuitively, the vector clocks track the \( <_E \) dependencies, but the precise invariant is technical. We need to introduce some notation to state it precisely.

Consider a complete observed trace \( \sigma \). For an event \( e \in \sigma \), \( \text{prefix}^\sigma(e) \) is the shortest prefix of \( \sigma \) that contains \( e \). For an arbitrary prefix \( \pi \) of \( \sigma \), we will find it useful to introduce notation for identifying some specific events in \( \pi \). For a pair \( p = \langle t, \text{op} \rangle \), \( \text{ev}_p^\pi \) denotes the last event of the form \( p \) in \( \pi \); note that for some pairs \( p \), this maybe undefined as there might be no event of this form in \( \pi \). Thus, for example, \( \text{ev}_{(t, \text{op})}^\pi \) denotes the last transaction begin event performed by thread \( t \) in \( \pi \). Sometimes, it will be convenient to leave one of the two arguments in the pair \( p = \langle t, \text{op} \rangle \) unspecified, and in this case \( \text{ev}_p^\pi \) will denote the last event of type identified by the specified argument. Thus, for example, \( \text{ev}_x^\pi \) is the last \( \text{x} \)-event in \( \pi \) (regardless of the thread performing it), and \( \text{ev}_{(t, \text{op})}^\pi \) is the last event of thread \( t \) in \( \pi \) (regardless of the operation). For an event \( e \), let us define \( B(e) \) to be the number of \( \langle \text{thr}(e), \text{op} \rangle \) events in \( \text{prefix}^\pi(e) \), i.e., \( B(e) \) is the number of begin transaction events performed by \( \text{thr}(e) \) before \( e \) (including \( e \)). Finally, to state the invariant, we identify the timestamp of an event in the prefix. This timestamp changes as we process more of the trace. The vector timestamp \( C(e, \pi) \) of event \( e \) in prefix \( \pi \) is given by

\[
C(e, \pi)(u) = \begin{cases} 
B(e) + 1, & \text{if } u = \text{thr}(e) \\
\max\{B(f) + 1 \mid f = \langle u, \text{op} \rangle \text{ and } f \leq e\}, & \text{otherwise}
\end{cases}
\]

We can now state the invariant that identifies the values of all the vector clocks maintained by the algorithm.

**Lemma 5**: After any prefix \( \pi \) of \( \sigma \), Algorithm 1 stores the following values.

\[
C_t = C(\text{prefix}^\pi_{\langle t, \text{op} \rangle}(\pi), \pi) \quad C_{t,x} = C(\text{prefix}^\pi_{\langle t, \text{x} \rangle}(\pi), \pi) \quad C_{t,x} = C(\text{prefix}^\pi_{\langle t, \text{x} \rangle}(\pi), \pi) \quad C_{t,x} = C(\text{prefix}^\pi_{\langle t, \text{x} \rangle}(\pi), \pi)
\]

The lemma is proved by an induction on the length of the trace processed by the algorithm. The proof is straightforward, and skipped. The invariant allows us to establish the correctness of the algorithm. The proof of Theorem 3 follows from Theorem 2 and Lemma 5.

APPENDIX B
OPTIMIZATIONS FOR AERO DROME

A. Read Clocks

The algorithm maintains the invariant that for two events \( e_1 \) and \( e_2 \) with \( \text{thr}(e_1) = t_1 \), we have \( C_{e_1} \subseteq C_{e_2} \) iff \( C_{e_1}(t_1) \leq C_{e_2}(t_1) \). In other words, in order to compare the timestamps of two events, it is enough to compare the local time corresponding to the thread of the smaller timestamp.

Now, observe that, at a write event \( e = \langle t, w(x) \rangle \), the algorithm detects an atomicity violation by either comparing with the clock of the last write event \( (w_x) \), or by comparing with the clocks of the last read events of each thread, except the thread \( t \). Let us consider the second check. Observe that in this case, a violation is raised if there is a thread \( u \neq t \) such that \( C_u^\pi \subseteq \mathbb{R}_{u,x} \). Based on our earlier observation about local times, this check is equivalent to the check \( \exists u \neq t \cdot C_u^\pi(t) \subseteq \mathbb{R}_{u,x}(t) \).

Now observe that

\[
\exists u \neq t \cdot C_u^\pi(t) \subseteq \mathbb{R}_{u,x}(t) \quad \text{iff} \quad C_u^\pi(t) \subseteq \bigcup_{u \neq t} \mathbb{R}_{u,x}(t)
\]

Based on the above observation, we can perform the check for atomicity if we have a single clock that maintains the timestamp \( \bigcup_{t \in T(x)} C_{t,x}(t) \), where \( C_{t,x}(t) \) is the last event of the form \( \langle u, x \rangle \) seen in the trace so far. For this, will use a new single clock \( \mathbb{C}_{t,x} \) to store this value and inductively maintain this in the algorithm.

Next, observe that the algorithm ensures that the timestamp of the last event in a given thread is larger than the timestamp of any earlier event in the same thread. This means that, at any point, \( R_{t,x} \subseteq C_t \) and thus we have \( R_{x} = C_{x} = C_{t} \) at any point in the algorithm. Now, let us consider how the algorithm updates \( C_t \) with the various \( R_{u,x} \) clocks at a write event. Precisely, if an atomicity violation is not detected when comparing with the read clocks, the value of \( C_t \) becomes \( C_{t}^\text{old} \cup \bigcup_{u \neq t} \mathbb{R}_{u,x}(t) \), where \( C_{t}^\text{old} \) is the value of \( C_{t} \) before the updates. Coupled with our previous observation, this new value is the same as the value \( C_{t}^\text{old} \cup \bigcup_{u \neq t} \mathbb{R}_{u,x}(t) \) which, in turn, can be re-written as \( C_{t}^\text{old} \cup \bigcup_{u \neq t} \mathbb{R}_{u,x}(t) \). Thus, we can maintain the timestamp \( \bigcup_{t \in T(x)} C_{t,x}(t) \) in a single clock \( \mathbb{C}_{t,x}(t) \) as before, is the last event of the form \( \langle u, x \rangle \) seen in the trace so far. We use a new clock \( \mathbb{R}_{x} \) to maintain this value.

We present the read clock optimization in Algorithm 2.

B. Other Optimizations

We now discuss some additional optimizations that help improve the runtime performance and memory overhead of AeroDrome. These are presented in Algorithm 3.

**Lazy Clock Updates**: This optimization is based on the following observation. Many times, a given memory location \( x \) is repeatedly read from in by a single thread, before being written to. This means that the algorithm updates the clocks \( R_x \) and \( \mathbb{C}_{t,x} \) (or the clocks \( R_{t,x} \) in line 26 of Algorithm 1) repeatedly, a lot of times, without being used to compute other clocks (lines 31 and 46 in Algorithm 1) or to detect atomicity violation. When the length of such contiguous subsequence of reads is large, these updates to \( R_x \) and \( \mathbb{C}_{t,x} \) are often redundant. To cater for this, we update the \( R_x \) clocks in a lazy fashion as follows. For every memory location, we maintain a set \( \text{Stale}_x^t \). \( \text{Stale}_x^t \) is maintained to be the set of threads \( t \) that have performed a read on \( x \) after the last write to \( x \) in
Algorithm 2 AeroDrome: Reducing the number of read clocks. Only procedures that differ from Algorithm 1 have been presented.

1: procedure INITIALIZATION
2: for \( t \in \text{Threads} \) do
3: \( \mathcal{C}_t := \bot[1/t] \); \( \mathcal{C}_t^\triangledown := \bot \);
4: for \( \ell \in \text{Locks} \) do
5: \( L_\ell := \bot \); lastRelThr\( _\ell := \text{NIL} \);
6: for \( x \in \text{Vars} \) do
7: \( \mathcal{W}_x := \bot \); lastWThr\( _x := \text{NIL} \);
8: \( \mathcal{R}_x := \bot \); ch\( _x := \bot \);
9: procedure READ\( (t, x) \)
10: if lastWThr\( _x \neq t \) then
11: CHECKANDGET\( (\mathcal{W}_x, \mathcal{W}_x, t) \);
12: \( \mathcal{R}_x := \mathcal{C}_t \);
13: \( \text{ch} \mathcal{R}_x := \mathcal{C}_t[0/t] \);
14: procedure WRITE\( (t, x) \)
15: if lastWThr\( _x \neq t \) then
16: CHECKANDGET\( (\mathcal{W}_x, \mathcal{W}_x, t) \);
17: CHECKANDGET\( (\text{ch} \mathcal{R}_x, \mathcal{R}_x, t) \);
18: \( \mathcal{W}_x := \mathcal{C}_t \);
19: lastWThr\( _x := t \);
20: procedure END\( (t) \)
21: for \( u \in \text{Threads} \setminus \{ t \} \) do
22: if \( \mathcal{C}_t^\triangledown \subseteq \mathcal{C}_u \) then
23: CHECKANDGET\( (\mathcal{C}_t, \mathcal{C}_t, u) \);
24: for \( \ell \in \text{Locks} \) do
25: \( L_\ell := \mathcal{C}_t^\triangledown \subseteq L_\ell \triangleq L_\ell \cup L_\ell : L_\ell \);
26: for \( x \in \text{Vars} \) do
27: \( \mathcal{W}_x := \mathcal{C}_t^\triangledown \subseteq \mathcal{W}_x \triangleq \mathcal{C}_t \cup \mathcal{W}_x : \mathcal{W}_x \);
28: if \( \mathcal{C}_t^\triangledown \subseteq \mathcal{R}_x \) then
29: \( \mathcal{R}_x := \mathcal{C}_t \cup \mathcal{R}_x \);
30: \( \text{ch} \mathcal{R}_x := \mathcal{C}_t[0/t] \cup \text{ch} \mathcal{R}_x \);

the current transaction of \( t \). And then, at a write event \( e = (t, w(x)) \), we use the values of the clocks \( \{ \mathcal{C}_u \mid u \in \text{Stale}_x^\triangledown \} \) to update \( \mathcal{C}_t, \mathcal{R}_x \) and \( \text{ch} \mathcal{R}_x \). This optimization therefore allows us to avoid expensive vector clock operations (at the majority of) read events in lieu of cheaper set operations (adding a thread to \( \text{Stale}_x^\triangledown \)). An analogous optimization also applies for the \( \mathcal{W}_x \) clocks.

Maintaining Sets of Memory Locations to be updated: Notice that at an end event (line 43 in Algorithm 1), we check, for every memory location \( x \), whether two clocks are ordered, and if so, perform clock updates accordingly. The set of memory locations in the entire trace can however be prohibitively large, and comparing vector clocks can be expensive (when performed for every location at every end event). We observed that most of the times, memory locations are often local to a small set of threads, and thus often, clock comparisons in line 43 are often redundant. We optimize the number of comparisons by maintaining, for every thread \( t \), the set of memory locations that have a read or write event ordered after some event in the (unique) active transaction of \( t \). Then, at an end event, we only need to iterate over this potentially smaller subset of memory locations.

Garbage Collection: This optimization is inspired from the garbage collection mechanism described in [14]. The basic idea there is the following. If a transaction \( T \) is such that there is no event \( e \) in the transaction that is ordered (using \( \leq_{\text{CHB}} \)) after some event of another transaction, then \( T \) cannot participate in any cycle and the analysis can essentially ignore such a transaction. This optimization can easily be implemented using vector clocks as follows. In order to check if a transaction of thread \( t \) has an incoming edge, we check if either the transaction that forked \( t \) is active or if there is a \( u \neq t \) such that \( \mathcal{C}_t^\triangledown (u) \neq \mathcal{C}_t (u) \) at the end of the transaction.
Algorithm 3 Optimized version of AeroDrome

1: procedure INITIALIZE
2: for \( t \in \text{Threads} \) do
3: \( C_t := \perp[t/t]; \quad C_t^u := \perp \)
4: UpdateSet\(_t^u := \emptyset; \quad \text{UpdateSet}^u_t := \emptyset; \)
5: for \( \ell \in \text{Locks} \) do
6: \( L_\ell := \perp; \quad \text{lastRelThr}_\ell := \text{NIL}; \)
7: for \( x \in \text{Vars} \) do
8: \( W_x := \perp; \quad \text{lastWThr}_x := \text{NIL}; \)
9: \( R_x := \perp; \quad \text{ehR}_x := \perp \)
10: \( \text{Stale}_x^c := \emptyset; \quad \text{Stale}_x^w := \text{NIL}; \)

11: procedure HASINCOMINGEDGE\((t, \ell)\)
12: return \((\text{parentTr}_t \text{ is alive}) \lor (C_t^0[t/t] \neq C_t[0/t])\);

13: procedure CHECKANDGET\((\text{clk}_1, \text{clk}_2, t)\)
14: if \( C_t^0 \subseteq \text{clk}_1 \) \&\& \( t \) has an active transaction then
15: declare ‘conflict serializability violation’;
16: \( C_t := C_t \cup \text{clk}_2; \)

17: procedure ACQUIRE\((t, \ell)\)
18: if \( \text{lastRelThr}_\ell \neq t \) then
19: \( \text{CHECKANDGET}(L_\ell, L_\ell, t); \)

20: procedure RELEASE\((t, \ell)\)
21: \( L_\ell := C_t; \)
22: \( \text{lastRelThr}_\ell := t; \)

23: procedure FORK\((t, u)\)
24: \( C_u := C_t \cup C_t; \)
25: procedure JOIN\((t, u)\)
26: \( \text{CHECKANDGET}(C_u, C_u, t); \)

27: procedure READ\((t, x)\)
28: if \( \text{lastWThr}_x \neq t \) then
29: if \( \text{Stale}_x^c = \top \) then
30: \( \text{CHECKANDGET}(C_{\text{lastWThr}_x}, C_{\text{lastWThr}_x}, t) \)
31: else
32: \( \text{CHECKANDGET}(W_x, W_x, t); \)
33: \( \text{Stale}_x^c := \text{Stale}_x^c \cup \{t\}; \)
34: for \( u \in \text{Threads} \) do
35: if \( u \) has an active transaction and \( C_u^0 \subseteq C_t \) then
36: \( \text{UpdateSet}_u^c := \text{UpdateSet}_u^c \cup \{x\}; \)

37: procedure WRITE\((t, x)\)
38: if \( \text{lastWThr}_x \neq t \) then
39: if \( \text{Stale}_x^w = \top \) then
40: \( \text{CHECKANDGET}(C_{\text{lastWThr}_x}, C_{\text{lastWThr}_x}, t); \)
41: else
42: \( \text{CHECKANDGET}(W_x, W_x, t); \)

43: for \( u \in \text{Stale}_x^c \) do
44: \( R_x := R_x \cup C_u; \)
45: \( \text{ehR}_x := \text{ehR}_x \cup C_u[0/u]; \)
46: \( \text{Stale}_x^w := \emptyset \)
47: \( \text{CHECKANDGET}(\text{ehR}_x, R_x, t); \)
48: \( \text{Stale}_x^w := \top; \)
49: \( \text{lastWThr}_x := t; \)
50: for \( u \in \text{Threads} \) do
51: if \( u \) has an active transaction and \( C_u^0 \subseteq C_t \) then
52: \( \text{UpdateSet}_u^w := \text{UpdateSet}_u^w \cup \{x\}; \)

53: procedure BEGIN\((t)\)
54: \( C_t(t) := C_t(t) + 1; \)
55: \( C_t^0 := C_t; \)

56: procedure END\((t)\)
57: if \( \text{HASINCOMINGEDGE}(t, \ell) \) then
58: for \( u \in \text{Threads} \cup \{t\} \) do
59: if \( C_u^0 \subseteq C_t \) then
60: \( \text{CHECKANDGET}(C_t, C_t, u); \)
61: for \( \ell \in \text{Locks} \) do
62: \( L_\ell := C_t^0 \subseteq L_\ell ? C_t \cup \cup \ell : L_\ell; \)
63: for \( x \in \text{UpdateSet}_t^w \) do
64: if \( \text{Stale}_x^w = \top \lor \text{lastWThr}_x = t \) then
65: \( W_x := C_t \cup W_x; \)
66: if \( \text{lastWThr}_x = t \) then
67: \( \text{Stale}_x^w := \top; \)
68: \( \text{UpdateSet}_t^w := \emptyset; \)
69: for \( x \in \text{UpdateSet}_t^r \) do
70: \( R_x := C_t \cup R_x; \)
71: \( \text{ehR}_x := \text{ehR}_x \cup C_t[0/t]; \)
72: \( \text{Stale}_x^r := \text{Stale}_x^r \setminus \{t\}; \)
73: \( \text{UpdateSet}_t^r := \emptyset; \)
74: else
75: for \( x \in \text{UpdateSet}_t^r \) do
76: \( \text{Stale}_x^r := \text{Stale}_x^r \setminus \{t\}; \)
77: \( \text{UpdateSet}_t^r := \emptyset; \)
78: for \( x \in \text{UpdateSet}_t^w \) do
79: if \( \text{lastWThr}_x = t \) then
80: \( \text{lastWThr}_x := \text{NIL}; \)
81: \( \text{UpdateSet}_t^w := \emptyset; \)
82: for \( \ell \in \text{Locks} \) do
83: if \( \text{lastRelThr}_\ell = t \) then
84: \( \text{lastRelThr}_\ell := \text{NIL}; \)