Atomicity Checking in Linear Time using Vector Clocks

Umang Mathur  Mahesh Viswanathan
University of Illinois at Urbana-Champaign

Abstract
Multi-threaded programs are challenging to write. 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 dynamic checking of 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.

1. Introduction
Writing correct multi-threaded programs is extremely difficult. It 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 [27].

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 [20, 14, 3, 13, 38, 45]. The advantage of static analysis approaches is that they may successfully prove that a program satisfies 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 [12, 50, 26, 11, 19, 5]. 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 1. All sound and precise dynamic analyses [11, 19, 5] 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 (one of which is a write) to a common memory location, 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 [19, 5] 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 sound and precise linear time algorithms, the key to achieving an efficient algorithm is the use of vector clocks [30]. 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

1 We use the term sound for a dynamic analysis technique if it does not report false alarms. This is consistent with the usage of “sound” in the context of dynamic analyses as explained in [40].
Atomicity specifications (i.e., which blocks of code should be processed) 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 streaming 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. As with standard vector clock algorithms, such as those used in data race detection, our algorithm summarizes information in vector clocks and thus does not need to store the timestamp of all the events in the trace to detect serializability violations.

We have implemented our algorithm in a prototype tool AIRTrack and have compared its performance against the Velodrome algorithm on various benchmark programs. Atomicity specifications (i.e., which blocks of code should be regarded as atomic) are hard to come by. One na"ıve 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 na"ıve specifications for some programs in our benchmark. For such benchmarks, the small transaction graph coupled with the overhead of maintaining vector clocks outweighs the benefits of a linear time algorithm, and Velodrome slightly outperforms our tool AIRTrack. For other\footnote{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.} programs in our benchmark, we use the more realistic atomicity specifications given in [5]. 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 2, we discuss preliminary notations such as that of concurrent program traces and the definition of conflict serializability. In Section 3, we use motivating examples to illustrate the challenges involved in developing a linear time vector clock algorithm for dynamically checking conflict serializability. In Section 4, 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 4 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 5. We discuss closely related work in Section 6 and present concluding remarks in Section 7. Some proofs and additional discussion can be found in the full version [2].

2. Preliminaries

An execution trace (or simply trace) of a concurrent program is a sequence of events, We will use \( \sigma, \rho_1, \rho_2, \ldots \) to denote traces. Each event in a trace is a pair \( e = (t, op) \), where \( t \) denotes the thread that performs \( op \) and \( op \) is the operation performed by \( e \); we will use \( \text{thr}(e) \) to denote \( t \) and \( \text{op}(e) \) to denote \( op \). Operations can be one of \( \text{r}(x), \text{w}(x) \) (read from or write to variable/memory location \( x \)), \( \text{acq}(l) \), \( \text{rel}(l) \) (acquire or release of lock object \( l \)), \( \text{fork}(u) \), \( \text{join}(u) \) (fork or join of thread \( u \)), \( \triangleright \) or \( \triangleleft \) (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 \footnote{We allow for nested blocks of begins and ends. In this case only the outermost begin and end constitute a transaction.} of events of thread \( t \) that starts with \( (t, \triangleright) \) and ends with the matching \( (t, \triangleleft) \), and we say \( e \in T \) if the event \( e \) belongs to this maximal subsequence; in this case, \( \text{txn}(e) \) denotes the transaction \( T \) to which \( e \) belongs. In a trace \( \sigma \), we will say that a transaction \( T \) is completed in \( \sigma \) if the corresponding end transaction event \( (\cdot, \triangleleft) \in \sigma \). If \( T \) is not completed in \( \sigma \), it is said to be active.

Given a trace \( \sigma \), we denote by \( \leq_T^\sigma \) the total order on events induced by \( \sigma \) — for events \( e, e' \) in \( \sigma \), we say \( e \leq_T^\sigma e' \) iff either \( e = e' \) or \( e \) occurs before \( e' \) in the sequence \( \sigma \). Two events
Definition 1 (Conflict Serializability [19]) A trace σ is conflict serializable if there is no sequence of \( k > 1 \) distinct transactions \( T_0, T_1, \ldots, T_{k-1} \) such that for every \( 0 \leq i \leq k-1 \), we have \( T_i \preceq_{Txn} T_{(i+1) \mod k} \). If σ is not conflict serializable, then such a sequence \( T_0, \ldots, T_{k-1} \) is said to be a witness to the violation.

Example 1 Consider the trace \( \rho_1 \) in Figure 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 \( T_1 = e_1 e_2 e_3 e_{10} \) 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 \( t_3 \). All pairs of events, both of which are performed by the same thread (such as \( (e_1, e_2) \) or \( (e_2, e_{10}) \) in \( \rho_1 \)) are conflicting. In addition, \( (e_2, e_4) \) and \( (e_7, e_8) \) are conflicting pairs of events in \( \rho_1 \) and we use an explicit arrow (\( \forearrows \)) to depict such inter-thread conflicting pairs. We have \( T_1 \preceq_{Txn} T_2 \) because \( e_2 \preceq_{CHB} e_4 \) and \( T_3 \preceq_{Txn} T_1 \) because \( e_3 \preceq_{CHB} e_9 \). Also note that \( \preceq_{CHB} \) is a transitive order and thus \( e_1 \preceq_{CHB} e_5 \) because \( e_1 \preceq_{CHB} e_2 \), \( e_2 \preceq_{CHB} e_4 \) and \( e_4 \preceq_{CHB} e_5 \). Finally, the trace \( \rho_1 \) is conflict serializable and the equivalent serial execution is the sequence \( \rho_1^{serial} = e_6 e_7 e_8 e_1 e_2 e_3 e_{10} e_4 e_5 e_9 \), in which the order of transaction is \( T_3 T_1 T_2 \). Observe that the relative order of conflicting events in \( \rho_1^{serial} \) is the same as in the original trace \( \rho_1 \).

Based on Definition 1, a cyclic dependency on transactions using \( \preceq_{Txn} \) suggests that \( \sigma \) does not have an equivalent serial execution and hence the program does not satisfy its atomicity specification. Previous techniques [19, 5] for checking conflict serializability dynamically, rely on constructing a directed graph. The vertices in such a graph are the different transactions in the observed trace, the edges correspond to the order imposed by \( \preceq_{Txn} \) and checking violations of conflict serializability reduces to searching for a cycle in this graph. These algorithms run in time that is quadratic in the length of the observed trace as they check for cycles each time a new edge is added in the graph.

3. 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 [21, 35, 15, 24, 29]. The broad principle behind these algorithms, is to assign vector timestamps to events as the trace is generated/observed so that the ordering between these assigned timestamps captures causal ordering. Notice that, conflict serializability is defined in terms of the relation \( \preceq_{Txn} \) on transactions (Definition 1), and thus, the most straightforward vector clock algorithm would rely on assigning timestamps to transactions in such a way that the timestamp of transaction \( T_i \) is less than or equal to timestamp of transaction \( T_j \) if and only if \( T_i \preceq_{Txn} T_j \). However, since a transaction is a sequence of events (and not a single event), the first challenge is figuring out how to assign and update timestamps of transactions when individual events are being continuously generated by the execution; this is one of the reasons why such algorithms were deemed impossible for atomicity in [19]. 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 Figure 1. Notice that there is a “path” from \( T_3 \) to \( T_2 \) (via \( T_1 \)) using \( \preceq_{Txn} \), even though \( T_3 \) starts after \( T_2 \) is completed in the trace \( \rho_1 \). Further, the discovery that \( T_3 \) has a path to \( T_2 \) can be made only after the event \( e_9 \) is generated in the trace, and at that point, both \( T_2 \)
and $T_1$ have completed. This poses serious challenges when designing a vector clock algorithm. A vector clock algorithm assigns a timestamp to transaction $T$ that is consistent with $\leq_{\text{txn}}$; it 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 ordered after the timestamps assigned to all these "predecessor" transactions. However, as transaction $T_2$ in trace $p_1$ illustrates, this may require knowing future events and transactions.

Example 2 illustrates that transactions $T'$ that have a $\leq_{\text{txn}}$-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 linear time 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), 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? Recall that, the relation $\leq_{\text{txn}}$ (on transactions) is defined in terms of the relation $\leq_{\text{CHB}}$ (on events), and therefore, a natural first step to explore, is to see if computing $\leq_{\text{CHB}}$ is sufficient to detect atomicity violations.

**Example 3** Consider the trace $p_2$ in Figure 2 with two transactions $T_1$ and $T_2$ in threads $t_1$ and $t_2$ respectively. Here, we have, $T_1 \leq_{\text{txn}}^p T_2$ and $T_2 \leq_{\text{txn}}^p T_1$, thus giving us a violation of conflict serializability with the sequence $T_1, T_2$ witnessing the violation. Now consider the following $\leq_{\text{CHB}}$ path in the trace $-e_1 \leq_{\text{CHB}}^p e_4 \leq_{\text{CHB}}^p e_5 \leq_{\text{CHB}}^p e_7$. This path, in fact, is symptomatic of the atomicity violation because it starts and ends in the same transaction (transaction $T_1$) and passes through another transaction (transaction $T_2$).

The atomicity violation in trace $p_2$ 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 $p_2$, specifically) such that $\text{txn}(e) = \text{txn}(g)$, $\text{txn}(e) \neq \text{txn}(f)$, and $e \leq_{\text{CHB}} f \leq_{\text{CHB}} g$. If we can prove that this is equivalent to Definition 1, then all we need to do is to compute (implicitly using vector clocks) the $\leq_{\text{CHB}}$ ordering. Unfortunately, this is not true, i.e., violations of conflict serializability cannot be detected by simply using $\leq_{\text{CHB}}$ ordering and searching for the above kind of $\leq_{\text{CHB}}$ paths. We illustrate this in the next example.

**Example 4** Consider trace $p_3$ in Figure 3. As before, let $T_1$, $T_2$ be the two transactions by threads $t_1$ and $t_2$ respectively. Here, both $T_1 \leq_{\text{txn}}^p T_2$ (because $e_3 \leq_{\text{CHB}} g$) and $T_2 \leq_{\text{txn}}^p T_1$ (because $e_4 \leq_{\text{CHB}} e_5$), thus giving us a conflict serializability violation. However, there is no $\leq_{\text{CHB}}$-path that starts and ends in the same transaction. If vector timestamps are used to compute $\leq_{\text{CHB}}$, then violations of conflict serializability cannot be detected by checking ordering of vector timestamps of events.

Example 4 demonstrates that $\leq_{\text{CHB}}$ 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 $(e_1, f_1), (e_2, f_2), \ldots (e_k, f_k)$ ($k > 1$) such that (a) $e = e_1$ and $f = f_k$, (b) $\text{txn}(e_i) = \text{txn}(f_i)$, while $\text{txn}(f_i) \neq \text{txn}(e_{i+1})$, for every $i$, and (c) $f_i \leq_{\text{CHB}} e_{i+1}$ 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 $f \leq_{\text{CHB}} e$.

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 $e \lessdot_{\sigma} f$, if there is an event $g$ in $\sigma$ such that $e \leq_{\text{CHB}} g$ and either (a) $g = f$, or (b) $g \lessdot_{\sigma} f$ and $\text{txn}(g)$ is completed in $\sigma$.

The following theorem formalizes how we can check for conflict serializability violations using the new relation. The proof of this theorem is presented in [2].
For a transaction $T$, let $T_e$ denote the begin transaction event $\langle \cdot, \cdot \rangle$ of $T$. The following observations hold.

1. Any trace $\sigma$ with a transaction $T$, events $e$ and $f$ such that $f \in T_e$, $e \notin T_e$, $T_e \prec_E e$ and $e \prec_E f$, is not conflict serializable.

2. Let $\rho$ be a trace that is not conflict serializable with a witness $I_0, I_1, \ldots, I_{k-1}$ such that each $T_i$ except possibly one, is complete. Then there is a transaction $T$ and events $e, f$ in $\sigma$ such that $f \in T_e$, $e \notin T_e$, $T_e \prec_E e$ and $e \prec_E f$.

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

Example 5 Let us begin by looking at trace $\rho_3$ in Figure 3. Let $\sigma_0$ denote the prefix of $\rho_3$ up to (and including) event $e_1$. In trace $\sigma_0$, we have $e_3 \prec_E e_6, e_4 \prec_E e_5, \text{ and } e_1 \prec_E e_6$ because they are related by $\preceq_{\text{CHB}}$. Here, $e_1 \prec_{\text{CHB}} e_6$ because $\tau(x) = \tau(x_1)$, $e_3 \preceq_{\text{CHB}} e_6$ and $\tau(x) = \tau(x_4)$. However, it is not the case that $e_1 \prec_{\text{CHB}} e_4$. On the other hand, if we consider $\sigma_7$, then $e_1 \prec_{\text{CHB}} e_4$ as the transaction in $t_1$ is complete in $\sigma_7$. In $\sigma_0$ (and therefore also in the full trace $\rho_3$, conditions of Theorem 2 are satisfied — $e_1 \prec_{\text{CHB}} e_4$ and $e_4 \prec_{\text{CHB}} e_7$.

Example 6 Consider trace $\rho_1^2$ in Figure 4: this is a slight modification of trace $\rho_1$ from Figure 1 that now has an atomicity violation. Again $e_1$ denotes the $i$th event, and $\sigma_i$ denotes the prefix up to event $e_i$. Notice that in prefix $\sigma_{i_1}$, $e_1 \preceq_{\text{CHB}} e_5$ (because $e_1 \preceq_{\text{CHB}} e_5$ and $e_5 \preceq_{\text{CHB}} e_1$ and $\tau(x_3) = \tau(x_4)$ is complete in $\sigma_{i_1}$). Thus by Theorem 2, there is a violation of conflict serializability.

4. Vector Clock Algorithm

Based on intuitions developed in Section 3, 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 $\text{Thr}$. 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 (or ordered before or simply before) another time $V_2$ (of the same dimension), denoted $V_1 \preceq V_2$ if $\forall t \in \text{Thr}, V_1(t) \leq V_2(t)$. In this case, we say that $V_2$ is greater than, ordered after or after $V_1$. The minimum vector time on threads $\text{Thr}$ is $\bot_{\text{Thr}} = \lambda t. 0$, and we will often use $\bot$ when $\text{Thr}$ is clear from context. Next, the join of two vector times $V_1$ and $V_2$ is the time $V_1 \sqcup V_2 = \lambda t. \max\{V_1(t), V_2(t)\}$. Finally, we use $V[\cdot/\cdot]$ to denote the timestamp $\lambda u. \text{ if } u = t \text{ then } c$ 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 times, 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.

4.1. The AeroDrome Algorithm

Our algorithm AeroDrome is a single pass linear time algorithm. It processes events in the trace as they are generated and (implicitly) assigns vector timestamps to each of these events. Broadly, the goal of the algorithm will be to assign vector timestamps that capture the relation $\preceq_E$ (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. Similar to other vector clock algorithms, such as those used in data race detection [35, 15], AeroDrome does not explicitly store the timestamps of each event in the trace; it instead maintains the timestamps of constantly many events using constantly many vector clocks. This small set of vector clocks is adequate for detecting conflict serializability violations.

Pseudocode for AeroDrome is shown in Algorithm 1. It processes events in the trace based on their operation, calling the appropriate handler. As mentioned before, the algorithm uses several vector clocks, which we will depict using the black-board font — $\mathbb{C}, \mathbb{L}, \mathbb{W}, \mathbb{R}$, 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 efficiently handle nested transactions and unary transactions, i.e., events not enclosed within a begin and end atomic block.

4.1.1. Vector Clocks and Other Data in the State

The most crucial set of clocks maintained by the algorithm are those of the form $C_t$, for each thread $t \in \text{Thr}$. The clock $C_t$, intuitively, stores the timestamp of the last event performed by the thread $t$ so far. That is, when performing an event $e = (t, o)$, the timestamp assigned to $e$ by AeroDrome is, in fact, determined by the value of the clock $C_t$ right after $e$ was processed by the algorithm. This is similar in spirit to vector clock algorithms for data race detection such as the standard DJIT+ [35] or its derivatives like FASTTrack [15].

The algorithm also checks for violations of conflict seri-
Algorithm 1: AeroDrome: Vector Clock Algorithm for Checking Violation of Conflict Serializability

1: procedure INITIALIZE
2:   for t ∈ Thr do
3:      $C_t := \bot \downarrow |1/t|; \ C_t^- := \bot;
4:   end for
5: for ℓ ∈ Locks do
6:      $L_{\ell} := \bot; \text{lastRelThr}_\ell := \text{NIL};
7:   end for
8: for x ∈ Vars do
9:      $W_x := \bot; \text{lastWThr}_x := \text{NIL};
10: end for
11: for t ∈ Thr do
12:      $R_{t,x} := \bot;
13: end for
14: procedure ACQUIRE(t, ℓ)
15:   if lastRelThr_ℓ ≠ t then
16:      CHECKANDGET($L_\ell, t$);
17:   end if
18: procedure RELEASE(t, ℓ)
19:   $L_\ell := C_t;
20:   lastRelThr_\ell := t;
21: procedure FORK(t, u)
22:   $C_u := C_t \cup {\ell};$
23: procedure JOIN(t, u)
24:   CHECKANDGET($C_u, t$);
25: procedure READ(t, x)
26:   if lastWThr_x ≠ t then
27:      CHECKANDGET($W_x, t$);
28:      $R_{t,x} := C_t;
29:   end if
30: procedure WRITE(t, x)
31:   if lastWThr_x ≠ t then
32:      CHECKANDGET($W_x, t$);
33: end if
34: procedure BEGIN(t)
35:   $C_t(t) := C_t(t) + 1$
36:   $C^-_t := C_t$
37: procedure END(t)
38:   for u ∈ Thr \ {t} do
39:      if $C^-_t \subseteq C^-_u$ then
40:         CHECKANDGET($C_t, u$);
41:   end if
42: end for
43: for ℓ ∈ Locks do
44:   $L_\ell := C^-_t \subseteq L_\ell? \ C_t \cup L_\ell : L_\ell$;
45: end for
46: for x ∈ Vars do
47:   $W_x := C^-_t \subseteq W_x? \ C_t \cup W_x : W_x$;
48: end for
49: for u ∈ Thr do
50:   $R_{u,x} := C^-_t \subseteq R_{u,x}? \ C_t \cup R_{u,x} : R_{u,x}$;

alizability using the characterization in Theorem 2, which relies on the timestamp of the begin event of a transaction. The algorithm, therefore, also maintains another clock $C^-_t$ which intuitively stores the timestamp of the last begin event 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 reflect the orderings induced by $\leq_{\text{CHB}}$. In order to capture the intra-thread dependencies imposed by $\leq_{\text{CHB}}$ and $\leq_E$, the algorithm ensures that for each thread $t$, the vector clock $C_t$ increases monotonically as new events arrive in the trace. 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 $\text{acq}(\ell)$ 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 $R_{t,x}$ that stores the time of the last $(t, x(x))$-event. Notice that when considering paths between events through transactions $(\xrightarrow{\rightarrow})$, we need to make sure that consecutive transactions along the path are different. To be able to track this constraint, AeroDrome will also maintain scalar variables lastRelThr_\ell and lastWThr_x, 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 $\bot \downarrow |1/t|$, all other clocks are initialized to $\bot$, and all the scalar variables are initialized to a default value of $\text{NIL}$.

4.1.2. Updates to the State As new events are observed in the trace, the algorithm updates these vector clocks in a manner that is consistent with tracking the $\leq_E$-relation.

When processing a begin event $e = (t, \xrightarrow{\rightarrow})$, the algorithm first increments the local component of $C_t$ (line 35 - \(C_t := C_t[C_t(t) + 1]\)). To understand why, let $e_{\text{prev}}$ be some event in the previous transaction (if any) by the same thread $t$. Further, let $e'$ be some event performed by a different thread $t' \neq t$ such that (a) $e_{\text{prev}} \leq_E e'$, and (b) $e \not<_E e'$. The increment of the local component ensures that this relationship between $e$, $e_{\text{prev}}$ and $e'$ can be accurately inferred from their timestamps by ensuring that the local component of the timestamp of $e$ is strictly greater than that of $e_{\text{prev}}$. Finally, AeroDrome updates
When processing an acquire event \( e = (t, \text{acq}(\ell)) \), the algorithm makes sure that the timestamp of \( e \) is ordered after the last release event \( e_r \) of lock \( \ell \). This is achieved by updating ‘\( C_t := C_t \cup L_\rho \)’ in the procedure \textsc{checkAndGet} when invoked at line 15; the procedure \textsc{checkAndGet} also checks for conflict serializability violation before updating \( C_t \), but more on that later. Of course, if \( e_i \) 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 \( e = (t, w(x)) \), \textsc{AeroDrome} ensures that the timestamp of \( e \) is ordered after all the prior reads and writes on \( x \) by calling \textsc{checkAndGet} in lines 29 and 31. The algorithm then updates \( W_x \) to be the timestamp of \( e \) (see line 32) and \textsc{lastWThr} to \( t \), thus preserving the semantics of the clock \( W_x \) and the scalar variable \textsc{lastWThr}. The updates performed at a read event are similar.

At a fork event \( e = (t, \text{fork}()\)\), the algorithm updates the clock of the child thread \( u \) (‘\( C_u := C_u \cup C_t \)’; line 20) so that all events of \( u \) are ordered after \( e \). At a join event \( e = (t, \text{join}(u)) \), 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 \( e = (t, <) \). Let \( e^o \) denote the matching begin transaction event. Observe that if for an event \( f \), \( e^o \prec_E f \), then since \( \text{txn}(e) \) is completed in \( \sigma \), \( e \prec_E f \). That is, all future events that are \( \prec_E \)-after \( e^o \) must be assigned a timestamp after that of \( e \). This is ensured by updating clocks \( C_u \) for all threads \( u \) that satisfy \( C^a_u \subseteq C_u \) (lines 38-40), and clocks \( \text{li}_t \), \( W_x \), and \( R_{u,i} \) (lines 41-46).

### 4.1.3. Checking Violations of Atomicity

The algorithm detects violations of atomicity at various points by a call to the procedure \textsc{checkAndGet}. The checks can be broadly classified into two categories. First, the algorithm can report a violation at an event \( e = (t, \text{op}) \) such that there is an earlier event \( e' \) (performed by a thread \( t' \neq t \)) that conflicts with \( e \). In this case, if \( e^o \prec_E e' \) (where \( e^o \) is the begin event of \( \text{txn}(e) \)), 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 \( e = (t, <) \) (with a matching begin event \( e^o \)). The algorithm detects a violation when there is another thread \( u \neq t \) with an active transaction whose begin event is \( e^o_u \), whose last event is \( e_u \) and \( e^o_u \prec_E e \) and \( e^o_u \prec_E e_u \) (line 40). These checks for violations of conflict serializability are performed in \textsc{checkAndGet}, which takes two arguments: \( \text{clk} \) (a vector timestamp) and \( t \) (a thread identifier), and declares a violation if (a) thread \( t \) has an active transaction, and (b) \( \text{clk} \) is ordered after \( C^a_t \), which is the timestamp of the begin event of the (active) transaction of \( t \) (line 10). It then updates the value of the clock \( C_t \) to \( C_t \cup \text{clk} \) (line 12).

### 4.1.4. 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 \textit{unary} transactions.

The pseudocode in Algorithm 1 works correctly even in the presence of 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-unary 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 (proof can be found in [2]).

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

### 4.2. AeroDrome on Example Traces

Let us illustrate \textsc{AeroDrome}’s workings on the traces from Section 3. Even though these examples do not use any synchronization primitives like locking, they contain all the features needed to highlight the subtle aspects of \textsc{AeroDrome}.

Let us begin with the simplest trace \( \rho_2 \) from Figure 2. We show the values of the relevant vector clocks in Figure 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 \( R_{t,i} \), \( R_{t,2,i} \), \( R_{t,1,y} \) or \( R_{t,1,y} \) as they are not important here. There are two threads and thus the size of each vector clock is 2. The clocks
$C_t$ and $C_r$ are initialized to the timestamps $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$ respectively, and all other clocks are initialized to $\bot = \langle 0, 0 \rangle$. The local clocks increment after a begin event (line 35 in Algorithm 1) and thus the clocks $C_t^{1}$ and $C_r^{2}$ become $\langle 2, 0 \rangle$ and $\langle 0, 2 \rangle$ after $e_2$. Further, these are also the values of the clocks $C_t^{1}$ and $C_r^{2}$ from this point onwards until the end of the execution. After processing $e_3 = (t_1, w(x))$, the value of the clock $W_x$ becomes $\langle 2, 0 \rangle$ (line 32). At event $e_4$, the call to CHECKANDGET (see line 25) with arguments $(\langle 2, 0 \rangle, t_2)$ updates the clock $C_t^{2}$ to $\langle 2, 2 \rangle$ (line 12). The clock $W_y$ gets updated to $\langle 0, 2 \rangle$ after processing $e_5$. Finally, at event $e_6$, the algorithm calls CHECKANDGET with arguments $(\langle 2, 2 \rangle, t_1)$. In this procedure, the algorithm asserts that $C_t^{1} \subseteq W_y$ and declares an atomicity violation.

Let us next consider the trace $\rho_3$ from Figure 3. AeroDrome’s run on this trace is shown in Figure 6. Updates corresponding to the first four events are straightforward. In event $e_5$, $C_t^{1}$ gets updated to $\langle 2, 2 \rangle$ because of the call to CHECKANDGET in line 25. Notice that this call does not raise any violation of atomicity because at this point, $C_t^{1} = \langle 2, 0 \rangle$ and the clock $W_y$ is $\langle 0, 2 \rangle$ thus failing the check $C_t^{1} \subseteq W_y$ 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 $e_7 = (t_1, \langle 0 \rangle)$. At this point, the algorithm checks if any event in the currently active transaction of $t_2$ is ordered after $e_1$ (condition $C_t^{1} \subseteq C_t^{2}$ in line 39 of Algorithm 1). This check succeeds since $C_t^{1} = \langle 2, 0 \rangle$ and $C_t^{2} = \langle 2, 2 \rangle$ at this point. The algorithm then checks if $C_t^{2} \subseteq C_t^{1}$ in the procedure 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 Figure 4. This example illustrates how AeroDrome handles dependencies between transactions introduced by future events. The run of AeroDrome on $\rho_1'$ is shown in Figure 7. We omit the updates to the clocks $R_{t,x}$ ($i \in \{1, 2, 3\}$, $u \in \{x, y, z\}$) 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 — $C_l^{1} = \langle 1, 0, 0 \rangle$, $C_r^{2} = \langle 0, 1, 0 \rangle$ and $C_t^{3} = \langle 0, 0, 1 \rangle$; all other clocks are initialized to $\langle 0, 0, 0 \rangle$. The begin events result in incrementing of local clocks and thus $C_t^{1} = \langle 2, 0, 0 \rangle$ after $e_1$. Further, the clock $W_x$ 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 $e_5 = (t_2, w(x))$, the clock $C_t^{2}$ gets updated to $\langle 2, 2, 0 \rangle$ as in line 12. After this, the transaction in $t_2$ ends. The clocks of none of the threads is updated because of $e_6$ as neither thread $t_1$ nor $t_2$ have clock values larger than $C_t^{1} = \langle 38-40 \rangle$. However the write and read clocks are updated. Specifically, the clock $W_y$ maintaining the timestamp to the last write is such that $C_t^{1} \subseteq W_y$ and thus, the algorithm updates $W_y$ to $W_y \cup C_t^{1} = \langle 2, 2, 0 \rangle$ (line 44 in Algorithm 1). Event $e_7$ is a begin event and updates $C_t^{1}$ to $\langle 0, 0, 2 \rangle$. Now at the $r(x)$ event $e_8$, the clock $C_t^{1}$ gets updated with $W_y$ which at this point evaluates to $\langle 2, 2, 0 \rangle$, thus giving $C_t^{1} = \langle 2, 2, 2 \rangle$. 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 $e_{11} = (t_1, r(z))$ — the algorithm checks if the clock $W_z$ knows some event in $t_1$ ($C_t^{1} \subseteq W_z$) and concludes that there is a violation of conflict serializability as this check passes.

4.3. 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 $O(|\text{Thr}|V)$, where $|\text{Thr}|$ is the number of threads 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 $O(|\text{Thr}|V)$ 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 $e = (t, w(x))$, the algorithm checks if there is a thread $u \neq t$ such that $C_t^{1} \subseteq R_{u,x}$ (line 10 in Algorithm 1). Second, these clocks are used to update $C_t^{1}$ — at a write event $e = (t, w(x))$, we set $C_t^{1} := \bigcup_{i \neq t} C_i \cup R_{u,x}$ (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 $O(|\text{Thr}|)$ 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 $\bigcup_{u} \bigcup_{x} R_{x}$, at each point while processing the trace. Next, to perform checks for violations of conflict serializability, we will have another clock $\mathfrak{e}rt_{x}$. (check read). This clock will store the value $\bigcup_{u} \bigcup_{x} R_{x} (0/|u|)$ at each point in the analysis. Based on the invariants maintained by the algorithm, one can show that checking $\mathfrak{e}rt_{y} \subseteq \bigcup_{u} \bigcup_{x} R_{x}$ is equivalent to checking $\mathfrak{e}rt_{y} \subseteq \mathfrak{e}rt_{x}$. This optimization and other useful optimizations that improve the performance of AeroDrome, are outlined in greater detail in [2].

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 $|\text{Thr}|$, $V$ 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 $O(|\text{Thr}|(n_{\text{non-end}} + (|\text{Thr}| + L + V)n_{\text{end}}))$ time and $O(|\text{Thr}|(|\text{Thr}| + V + L))$ space.

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

5. **Experimental Evaluation**

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

5.1. **Implementation**

We have implemented AeroDrome in a prototype tool AIRTRACK. AIRTRACK is available publicly; we omit the URL to preserve anonymity of the authors. AIRTRACK is written in Java and analyzes traces generated by concurrent programs and detects violations of conflict serializability. The primary goal of the evaluation is to assess if the theoretical bound (linear time) of the algorithm also translates to effective performance in practice, or in other words, does our vector clock algorithm perform better than existing approaches such as the classical graph based algorithm (Velodrome) proposed in [19]? We emphasize that the primary purpose of the evaluation is to compare different algorithms for checking atomicity instead of comparing different tools that implement these algorithms.

**Logging.** In order to evaluate our algorithm against the above objective and to ensure a fair comparison with other approaches, we must ensure that all competing candidate algorithms analyze the same trace. However, the dynamic behavior of a concurrent program can vary significantly across different runs, even when starting with the same input. In order to ensure fairness, we compare the performance of the different algorithms on the same dynamic execution. Our tool AIRTRACK therefore first extracts an execution trace from a concurrent program and then analyzes the same trace against all candidate algorithms. We use RoadRunner [16] to log traces from our set of benchmark programs. RoadRunner uses load time program instrumentation and can be extended to log various events — read and write accesses to memory locations, acquire and release of synchronization objects (locks), forks and joins of threads, and events generated at the entry and exit of each method, which we respectively mark as transaction begin ($\triangleright$) and end ($\triangleleft$) events.

**Velodrome.** The Velodrome algorithm [19] runs in (worst case) quadratic time and analyzes traces by building a directed graph, with transactions as nodes in the graph and where the edges correspond to $\triangleleft_{\text{Txn}}$ relation between transactions. There was no publicly available implementation of Velodrome that analyzes logged executions. Thus, we also implement this algorithm in AIRTRACK. We use the Java graph library JGraphT [31] to implement various graph operations (adding nodes and edges, cycle detection, etc.) in Velodrome algorithm. In our implementation of Velodrome, we also incorporate garbage collection as an optimization suggested in [19] — transactions with no incoming edges do not participate in cycles and can be deleted from the graph. In line with the objective of our evaluation, we analyze AeroDrome and Velodrome on the same trace (generated by RoadRunner) to ensure a fair comparison.

**Other techniques.** The tool DoubleChecker [5] is a state-of-the-art tool for checking conflict serializability in a sound and complete manner. DoubleChecker implements a two-phase analysis — the first phase performs a fast but imprecise analysis and reports an over-approximation of the actual set of cycles in the transaction graph. The second phase then filters out the false positives from this set with a more fine grained analysis. DoubleChecker’s performance crucially relies on the first phase being carried out while the program executes. Therefore, one cannot get performance data for DoubleChecker on a logged trace. As a result, there can be no fair comparison between our algorithm and DoubleChecker as one cannot guarantee that the two analyses run on the same trace. In order to gauge if DoubleChecker will significantly outperform our implementation of AeroDrome, we ran DoubleChecker’s publicly available implementation [1] on a subset of our benchmarks. On these benchmarks, DoubleChecker’s performance was slower by an order of magnitude. While these experiments do not indicate that DoubleChecker performs worse than our algorithm, they do suggest that our algorithm will be competitive against DoubleChecker. We choose not to present these numbers in this paper, because they are not an apples-to-apples comparison.

5.2. **Atomicity Specifications, Benchmarks and Setup**

**Atomicity Specifications.** In general, the logging mechanism in RoadRunner instruments and tracks all events corresponding to entering and exiting methods. A naïve atomicity spec-
Table 1: 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>Variants</th>
<th>Transactions</th>
<th>Atomic?</th>
<th>Velodrome (s)</th>
<th>AeroDrome (s)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>avora</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>elevator</td>
<td>280K</td>
<td>5</td>
<td>50</td>
<td>725</td>
<td>22.6K</td>
<td></td>
<td>162</td>
<td>1.7</td>
<td>97</td>
</tr>
<tr>
<td>hedy</td>
<td>9.8</td>
<td>7</td>
<td>13</td>
<td>1694</td>
<td>84</td>
<td></td>
<td>0.07</td>
<td>0.06</td>
<td>1.16</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>TO</td>
<td>581</td>
<td>674</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; 6545</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>philo</td>
<td>613</td>
<td>6</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>✓</td>
<td></td>
<td>0.02</td>
<td>0.02</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></td>
<td>55m40s</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>tsp</td>
<td>312M</td>
<td>9</td>
<td>2</td>
<td>181M</td>
<td>9</td>
<td>X</td>
<td>4.2</td>
<td>5.7</td>
<td>0.73</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></td>
<td>1.6</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**5.3. Evaluation Results**

For the first set of benchmarks (Table 1), we use the atomicity specification obtained from prior work [5]. For the second set of benchmarks (Table 2), we use default atomicity specifications (all methods except `main` and `run` are assumed to be atomic). The specifications from [5] are carefully crafted to ensure that spurious atomicity violations are not reported. In the absence of careful specifications, we can expect that the violations will be reported early on in executions.

Let us first consider the first set of benchmarks from Table 1. 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 a linear running time. Similarly, in the benchmark avora, 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 (avora, lusearch, moldyn, raytracer, xalan) and demonstrates the effectiveness of a linear time vector clock algorithm. For the examples on which AeroDrome does not give a huge speedup over Velodrome, we discovered that the number of nodes in Velodrome’s graph analysis is fairly small; for example, there were 13 nodes in the graph for pmd, 4 nodes in sor and 13 nodes in xalan.

In the second set of benchmarks, we notice that the per-
formance 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 tomatc, 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, and as a result, the graph based algorithm runs faster.

6. 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 [27] 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 of type systems [18, 20]. These type systems rely on commutativity of operations and are inspired from Lipton’s theory of reduction [25] and the concept of purity [13]. Extensions to type inference [38] and to programs with non-blocking synchronization [45] have been developed. The work in [18] uses constraint based type system inference for inferring atomicity specifications.

Dynamic analysis algorithms for checking atomicity inspect individual program executions instead of the program source code. Lipton’s theory of reduction [25] has been a prominent theme in this space, most notably the analysis employed by Atomizer [12]. This approach however leads to false alarms. The notion of conflict serializability was introduced concurrently by Flanagan et al [19] and Farzan et al [11], with subtle differences on how synchronization events need to be ordered. In particular, Farzan et al [10] 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 lock-sets in Goldilocks [9] algorithm for employing HB for race detection, and similar to the case in race detection [15, 29], such an algorithm is expected to be orders of magnitude slower than a vector clock algorithm. We refer the reader to [5] 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 [32]. Recently, DoubleChecker [5] 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. The notion of causal atomicity [10] 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 [34] and AVIO [26] resort to directed exploration of thread interleavings to expose subtle atomicity violations. Penelope [42] detects 2 thread atomicity violations using directed interleaving exploration. The work in [47, 46, 28, 4] is also based on exercising specific thread schedules. SMT solving based predictive analysis techniques [44] have been developed, but tend to not scale. The work of Samak et. al. [37] synthesizes directed unit tests for catching atomicity violations. The work in [10, 39] 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 [23, 33]. Jin et. al. [22] study the problem of synthesizing bug patches for fixing atomicity bugs.

7. 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 the first linear time, vector clock algorithm 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
atomicty [10], view serializability [46] or reduction based atomicty characterizations as in [12, 47]. Other promising lines of work include improving the efficiency of the proposed dynamic analysis for atomicty by incorporating ideas from data race detection. This includes the classic epoch optimizations [15], static analysis for redundancy elimination [17] and optimal check placement [36], and other advances concerning instrumentation [7, 48, 8, 49].

References


8. Proof of for Theorem 2

Proof: (⇐) Observe that for any pair of events \( e_1, e_2 \) if \( \text{txn}(e_1) \neq \text{txn}(e_2) \) and \( e_1 <_{\mathcal{E}} e_2 \) then \( e_1 \rightarrow_{\sigma} e_2 \). This means \( T_\sigma \rightarrow_{\sigma} e \rightarrow_{\sigma} f \). Since \( \text{txn}(f) = \text{txn}(T_\sigma) \), we can rewrite this as \( T_\sigma \rightarrow_{\sigma} e \rightarrow_{\sigma} f \). The rest of the proof follows from Proposition 1, the observation that \( T_\sigma \leq_{CHB} T_\sigma \).

(⇒) Let \( T_0, \ldots, T_{k-1} \) be a witness sequence for the conflict serializability violation of \( (k > 1) \) and \( T_i \neq T_j \) for every \( i \neq j \). Then, we must have a sequence of pairs of events \((e_0, f_0), \ldots, (e_{k-1}, f_{k-1})\) such that \( \text{txn}(e_i) = \text{txn}(f_i) = T_i \), and \( f_i \leq_{CHB} e_{(i+1) \mod k} \). Observe that for every \( i \neq j \), we have \( e_i \rightarrow_{\sigma} f_j \). Let \( m \) be the index of the only active transaction in \( \sigma \) amongst \( \{T_i\}_{i=0}^{k-1} \); if all transactions are completed, pick \( m = 0 \). Now let \( T = T_m, e = e_{(m+1) \mod k} \notin T \) and \( f = f_m \in T \). Now, \( T_\sigma \leq_{CHB} T_m \leq_{CHB} e_{(m+1) \mod k} \). Also, because of the choice of \( m \), the transaction \( \text{txn}(e) \) is completed in \( \sigma \) and \( e \rightarrow_{\sigma} f \).

9. Correctness of AeroDrome

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 \( <_{\mathcal{E}} \) 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 \( e \in \sigma \), 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 \( p = (t, op) \), \( ev_p \) 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, \( ev_{\{t, \cdot \}} \) 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 \( p = (t, op) \) unspecified, and in this case \( ev_p \) will denote the last event of type identified by the specified argument. Thus, for example, \( ev_{\{t, \cdot \}}(x) \) is the last \( w(x) \)-event in \( \pi \) (regardless of the thread performing it), and \( ev_{\{t, \cdot \}}(y) \) 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 \( \{\text{thr}(e), t\} \) events in prefix\( \sigma (e) \), 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

\[
C(e, \pi)(u) = \begin{cases} B(e) + 1, & \text{if } u = \text{thr}(e) \\ \max\{B(f) + 1 | f = (u, op) \text{ and } f <_{\mathcal{E}} e\}, & \text{otherwise} \end{cases}
\]

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.

\[
\begin{align*}
C_t &= C(ev_{\{t, \cdot \}}, \pi) \\
C_{\pi'} &= C(ev_{\{\cdot, \cdot \}}, \text{prefix}(ev_{\{t, \cdot \} })) \\
R_{x,t} &= C(ev_{\{\cdot, x(t) \}}, \pi) \\
W_x &= C(ev_{\{\cdot, w(x) \}}, \pi) \\
L_{\pi} &= C(ev_{\{\cdot, \text{rel}(t) \}}, \pi)
\end{align*}
\]

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.

10. Optimizations for AeroDrome

10.1. 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 \( C_{e_1}(t_1) \subseteq C_{e_2}(t_1) \). 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 \( e = (t, w(x)) \), 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 \( C_{\pi'}(t) \subseteq R_{u,x}(t) \). Based on our earlier observation about local times, this check is equivalent to the check \( \exists u \neq t: C_{\pi'}(t) \subseteq R_{u,x}(t) \). Now observe that

\[
\exists u \neq t: C_{\pi'}(t) \subseteq R_{u,x}(t) \quad \text{iff} \quad C_{\pi'}(t) \subseteq \bigcup_{u \neq t} R_{u,x}(t) \quad \text{iff} \quad C_{\pi'}(t) \subseteq \bigcup_{u} R_{u,x}(0/u)(t)
\]

Based on the above observation, we can perform the check for atomicity if we have a single clock that maintains the timestamp \( \bigcup_{u} R_{u,x}(0/u)(t) \), where \( e_{u,x}(t) \) is the last event of the form \( (u, x(t)) \) seen in the trace so far. For this, will use a new single clock \( e_{\text{ch}}R_x \) 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, \( R_{x,t} \subseteq C_t \) and thus we have \( R_x \cup C_t = C_t \) at any point in the algorithm. Now, let us consider how the algorithm updates \( C_t \) with the various \( R_{x,t} \) 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 \( C_{old} \cup \bigcup_{u \neq t} R_{u,x} \), where \( C_{old} \) is the value of \( C_t \) before the updates. Coupled with our previous observation, this new value is the same as the value \( C_{old} \cup R_{x,t} \cup \bigcup_{u \neq t} R_{u,x} \), which, in turn, can
be re-written as $C^\text{old}_u \cup \bigsqcup_u R_u$. Thus, we can maintain the timestamp $\bigcup_{x(u)} C_{t,x(u)}$ in a single clock $(e_{u,x(x)})$ as before, is the last event of the form $(u, x(x))$ seen in the trace so far. We use a new clock $R_x$ to maintain this value.

We present the read clock optimization in Algorithm 2.

### 10.2. 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 $c\eta_x$ (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 $c\eta_x$ 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 $\text{Stale}_x$, which is the set of threads $t$ that have performed a read on $x$ after the last write to $x$ in the current transaction of $t$. And then, at a write event $e = (t, w(x))$, we use the values of the clocks $\{C_u | u \in \text{Stale}_x\}$ to update $C_t$, $R_x$ and $c\eta_x$. 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 $\text{Stale}_x$). An analogous optimization also applies for the $W_x$ 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 [19]. 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 transac-
Algorithm 2 AeroDrome: Reducing the number of read clocks. Only procedures that differ from Algorithm 1 have been presented.

1: procedure INITIALIZATION
2: for \( t \in \text{Thr} \) do
3: \( C_t := \bot[1/t]; C^r_t := \bot; \)
4: for \( \ell \in \text{Locks} \) do
5: \( L_\ell := \bot; \text{lastRelThr}_\ell := \text{NIL}; \)
6: for \( x \in \text{Vars} \) do
7: \( W_x := \bot; \text{lastWThr}_x := \text{NIL}; \)
8: \( R_x := \bot; \text{ch}R_x := \bot; \)
9: \( \text{procedure} \ READ(t, x) \)
10: if lastWThr\(_x\) \( \neq t \) then
11: \( \text{CHECKANDGET}(W_x, W_x, t); \)
12: \( R_x := C_t; \)
13: \( \text{ch}R_x := C_t[0/t]; \)
14: \( \text{procedure} \ WRITE(t, x) \)
15: if lastWThr\(_x\) \( \neq t \) then
16: \( \text{CHECKANDGET}(W_x, W_x, t); \)
17: \( \text{CHECKANDGET}(\text{ch}R_x, R_x, t); \)
18: \( W_x := C_t; \)
19: \( \text{lastWThr}_x := t; \)
20: \( \text{procedure} \ END(t) \)
21: for \( u \in \text{Thr} \setminus \{t\} \) do
22: if \( C^u \subseteq C_t \) then
23: \( \text{CHECKANDGET}(C_t, C_t, u); \)
24: for \( \ell \in \text{Locks} \) do
25: \( L_\ell := C^u \subseteq L_\ell \wedge C_t \cup L_\ell : L_\ell; \)
26: for \( x \in \text{Vars} \) do
27: \( W_x := C^u \subseteq W_x \wedge C_t \cup W_x : W_x; \)
28: if \( C^u \subseteq R_x \) then
29: \( R_x := C_t \cup R_x; \)
30: \( \text{ch}R_x := C_t[0/t] \cup \text{ch}R_x; \)
Algorithm 3 Optimized version of AeroDrome

1: procedure INITIALIZATION
2: for \( t \in \text{Thr} \) do
3: \( C_t := \perp[1/t]; C^*_t := \perp; \)
4: UpdateSet\(_t^\prime := \emptyset; \) UpdateSet\(_t^\prime := \emptyset; \)
5: for \( \ell \in \text{Locks} \) do
6: \( L_\ell := \perp; \text{lastRelThr}_\ell := \text{NIL}; \)
7: for \( x \in \text{Vars} \) do
8: \( W_x := \perp; \text{lastWThr}_x := \text{NIL}; \)
9: \( R_x := \perp; \text{chR}_x := \perp; \)
10: \( \text{Stale}_x^\prime := \emptyset; \text{Stale}_x^\prime := \text{NIL}; \)
11: procedure HASINCOMINGEDGE\( (t) \)
12: return (parentTr\(_t \) is alive) \( \lor (C^*_t[0/t] \neq C_t[0/t]) \);
13: procedure CHECKANDGET\( (\text{clk}_1, \text{clk}_2, t) \)
14: if \( C^*_t \subseteq \text{clk}_1 \) and \( t \) 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: CHECKANDGET\( (\text{L}_\ell, \text{L}_\ell, t) \);
20: procedure RELEASE\( (t, \ell) \)
21: \( \text{L}_\ell := C_t; \)
22: \( \text{lastRelThr}_\ell := t; \)
23: procedure FORK\( (t, u) \)
24: \( C_u := C_u \cup C_t; \)
25: procedure JOIN\( (t, u) \)
26: CHECKANDGET\( (C_u, C_u, t) \);
27: procedure READ\( (t, x) \)
28: if \( \text{lastWThr}_x \neq t \) then
29: if \( \text{Stale}_x^\prime = \text{T} \) then
30: CHECKANDGET\( (\text{C}_{\text{lastWThr}_x}, \text{C}_{\text{lastWThr}_x}, t) \);
31: else
32: CHECKANDGET\( (\text{W}_x, \text{W}_x, t) \);
33: \( \text{Stale}_x^\prime := \text{Stale}_x^\prime \cup \{t\}; \)
34: for \( u \in \text{Thr} \) do
35: if \( u \) has an active transaction and \( C^*_u \subseteq C_t \) then
36: UpdateSet\( u^\prime := \text{UpdateSet}_u^\prime \cup \{t\}; \)
37: procedure WRITE\( (t, x) \)
38: if \( \text{lastWThr}_x \neq t \) then
39: if \( \text{Stale}_x^\prime = \text{T} \) then
40: CHECKANDGET\( (\text{C}_{\text{lastWThr}_x}, \text{C}_{\text{lastWThr}_x}, t) \);
41: else
42: CHECKANDGET\( (\text{W}_x, \text{W}_x, t) \);
43: for \( u \in \text{Stale}_x^\prime \) do
44: \( R_x := R_x \cup C_u; \)
45: \( \text{chR}_x := \text{chR}_x \cup C_u[0/u]; \)
46: \( \text{Stale}_x^\prime := \emptyset; \)
47: CHECKANDGET\( (\text{chR}_x, R_x, t) \);
48: \( \text{Stale}_x^\prime := \text{T}; \)
49: \( \text{lastWThr}_x = t; \)
50: for \( u \in \text{Thr} \) do
51: if \( u \) has an active transaction and \( C^*_u \subseteq C_t \) then
52: UpdateSet\( u^\prime := \text{UpdateSet}_u^\prime \cup \{t\}; \)
53: procedure BEGIN\( (t) \)
54: \( C_t(t) := C_t(t) + 1; \)
55: \( C^*_t := C_t; \)
56: procedure END\( (t) \)
57: if \( \text{HASINCOMINGEDGE}(t) \) then
58: for \( u \in \text{Thr} \{t\} \) do
59: if \( C^*_u \subseteq C_u \) then
60: CHECKANDGET\( (\text{C}_t, C_t, u) \);
61: for \( \ell \in \text{Locks} \) do
62: \( L_\ell := C^*_t \subseteq \text{L}_\ell \ ? C_t \cup L_\ell \ : L_\ell; \)
63: for \( x \in \text{UpdateSet}_u^\prime \) do
64: if \( \text{Stale}_x^\prime = \text{T} \) and \( \text{lastWThr}_x = t \) then
65: \( \text{W}_x := C_t \cup \text{W}_x; \)
66: if \( \text{lastWThr}_x = t \) then
67: \( \text{Stale}_x^\prime := \text{T}; \)
68: UpdateSet\( u^\prime := \emptyset; \)
69: for \( x \in \text{UpdateSet}_u^\prime \) do
70: \( R_x := C_t \cup R_x; \)
71: \( \text{chR}_x := \text{chR}_x \cup C_u[0/t]; \)
72: \( \text{Stale}_x^\prime := \text{Stale}_x^\prime \setminus \{t\}; \)
73: UpdateSet\( u^\prime := \emptyset; \)
74: for \( x \in \text{UpdateSet}_u^\prime \) do
75: \( \text{Stale}_x^\prime := \text{Stale}_x^\prime \setminus \{t\}; \)
76: UpdateSet\( u^\prime := \emptyset; \)
77: for \( x \in \text{UpdateSet}_u^\prime \) do
78: if \( \text{lastWThr}_x = t \) then
79: \( \text{Stale}_x^\prime := \text{T}; \)
80: \( \text{lastWThr}_x := \text{NIL}; \)
81: UpdateSet\( u^\prime := \emptyset; \)
82: for \( \ell \in \text{Locks} \) do
83: if \( \text{lastRelThr}_\ell = t \) then
84: \( \text{lastRelThr}_\ell := \text{NIL}; \)