Atomicity Checking in Linear Time using Vector Clocks

Umang Mathur
umathur3@illinois.edu
University of Illinois at Urbana-Champaign

Mahesh Viswanathan
vmahesh@illinois.com
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 dynamically checking atomicity. Existing algorithms detect violations of conflict serializability by detecting cycles in a graph of transactions observed in a given execution. The number of edges in such a graph can grow quadratically with the length of the trace making the analysis not scalable. In this paper, we present AeroDrome, a novel single pass linear time algorithm that uses vector clocks to detect violations of conflict serializability in an online setting. Experiments show that AeroDrome scales to traces with a large number of events with significant speedup.

CCS Concepts. • **Software and its engineering → Dynamic analysis**; **Software testing and debugging**.

Keywords. Concurrency, Atomicity, Conflict Serializability, Vector Clocks, Dynamic Program Analysis

ACM Reference Format:
Umang Mathur and Mahesh Viswanathan. 2020. Atomicity Checking in Linear Time using Vector Clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3373376.3378475

1 Introduction

Writing correct multi-threaded programs is extremely difficult. It is the class of software that is most prone to errors. Reasoning about 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 specifications is quite common and is the root cause in a majority of real-world bugs [9, 13, 22, 29, 35, 37, 61].

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 [3, 18, 20, 21, 54, 62]. 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 [5, 12, 13, 19, 39, 67]. 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 techniques for detecting atomicity violations have 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]. Most sound and precise dynamic analyses [5, 12, 19] 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 (at least one of which is a write access) to a common memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378475

[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 the term "sound" in the context of dynamic analyses [56].

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 [5, 19] rely on this idea and thus have an asymptotic complexity of cubic time — each new event of the trace requires updating the transaction graph, and checking for cycles; the number of edges can be quadratic in the number of events, giving a quadratic processing time *per event*.

The central question motivating this paper is the following: Is a cubic running time necessary for checking conflict serializability? Or are there sub-cubic 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 [23, 27, 43, 45]. 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 [19], in fact, dismiss the possibility of a vector clock based algorithm for conflict serializability checking:

> "The traditional representation of clock vectors [45] 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, AeroDrome, is a single pass, streaming algorithm that runs in linear time [2]. As with standard vector clock algorithms, such as those used in data race detection [14, 50], 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 AeroDrome in our tool Rapid [41] and have compared its performance against Velodrome [19] on various benchmark programs. Atomicity specifications (i.e., which blocks of code should be regarded as atomic) are hard 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. For such benchmarks, conflict serializability is trivially violated in a small prefix of the observed trace. The resulting transaction graph is thus small, the overhead of maintaining vector clocks outweighs the benefits of a linear time algorithm, and Velodrome slightly outperforms AeroDrome. For other 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 Rapid and its performance evaluation on a suite of benchmark programs is discussed in Section 5. We discuss closely related work in Section 6 and

[2] Vector clock based algorithms are linear time under the computational assumption that arithmetic operations take constant time. This is a reasonable assumption 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 in constant time.

present concluding remarks in Section 7. Some proofs and additional discussion can be found in the full version [44].

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 $e = \langle t, op \rangle$, where t denotes the thread that performs e and op is the operation performed by e; we will use $\mathtt{thr}(e)$ to denote t and $\mathtt{op}(e)$ to denote op. Operations can be one of $\mathtt{r}(x), \mathtt{w}(x)$ (read from or write to variable/memory location x), $\mathtt{acq}(\ell)$, $\mathtt{rel}(\ell)$ (acquire or release of lock object ℓ), $\mathtt{fork}(u)$, $\mathtt{join}(u)$ (fork or join of thread u), $\rhd$ or $\lhd$ (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 [3] of events of thread t that starts with $\langle t, \rhd \rangle$ and ends with the matching $\langle t, \lhd \rangle$, and we say $e \in T$ if the event e belongs to this maximal subsequence; in this case, $\mathtt{txn}(e)$ denotes the transaction T to which e belongs. In a trace σ, we will say that a transaction T is *completed* in σ if the corresponding end transaction event $\langle \cdot, \lhd \rangle \in \sigma$. If T is not completed in σ, it is said to be *active*.

Given a trace σ, we denote by $\leq^{\sigma}_{\mathtt{tr}}$ the total order on events induced by σ — for events e, e' in σ, we say $e \leq^{\sigma}_{\mathtt{tr}} e'$ iff either $e = e'$ or e occurs before e' in the sequence σ. Two events e, e' are said to be *conflicting* if either (i) $\mathtt{thr}(e) = \mathtt{thr}(e')$, (ii) $e = \langle t, \mathtt{fork}(u) \rangle$ and $\mathtt{thr}(e') = u$, (iii) $\mathtt{thr}(e) = u$ and $e' = \langle t, \mathtt{join}(u) \rangle$, (iv) there is a common memory location x such that both $\mathtt{op}(e), \mathtt{op}(e')$ are one of $\{\mathtt{w}(x), \mathtt{r}(x)\}$ and not both are $\mathtt{r}(x)$, or (v) there is a lock ℓ such that $\mathtt{op}(e) = \mathtt{rel}(\ell)$ and $\mathtt{op}(e') = \mathtt{acq}(\ell)$. Given a trace σ, *conflict-happens-before* $\leq^{\sigma}_{\mathtt{CHB}}$ is the smallest reflexive, transitive relation such that for every pair of conflicting events $e \leq^{\sigma}_{\mathtt{tr}} e'$, we have $e \leq^{\sigma}_{\mathtt{CHB}} e'$.

Atomicity is closely related to the property of conflict serializability. Informally, this property requires that an execution be equivalent to a *serial* execution by commuting adjacent non-conflicting events; an execution is serial if for every thread t in the trace and for every transaction T of thread t, there are no events of any other thread between the begin and end events of T. In this context, if two events e and e' are ordered by $\leq_{\mathtt{CHB}}$, 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 $e \leq^{\sigma}_{\mathtt{CHB}} e'$. 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

[3] We allow for nested blocks of begins and ends. In this case only the outermost begin and end constitute a transaction.

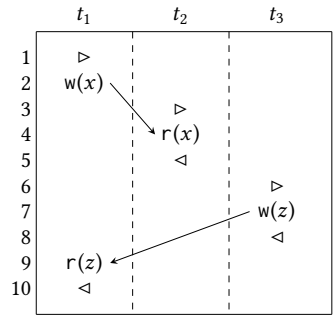

Figure 1. Trace ρ_1. Taking T_i to be the transaction of thread t_i, we have $T_3 \lessdot^{\rho_1}_{\mathtt{Txn}} T_1 \lessdot^{\rho_1}_{\mathtt{Txn}} T_2$.

that *every* event of T should happen before each event of T'. Thus, transaction T must *happen before* transaction T' in trace σ (denoted $T \lessdot^{\sigma}_{\mathtt{Txn}} T'$) if there are events $e \in T$ and $e' \in T'$ such that $e \leq^{\sigma}_{\mathtt{CHB}} e'$. We now present the definition of conflict serializability (which implies atomicity) from [19].

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 $T_i \lessdot^{\sigma}_{\mathtt{Txn}} T_{(i+1) \bmod k}$. 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 ρ_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 ith event in the trace. This trace has three transactions — transaction $T_1 = e_1e_2e_9e_{10}$ is performed in t_1, transaction $T_2 = e_3e_4e_5$ is performed in t_2 and transaction $T_3 = e_6e_7e_8$ 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 ρ_1) are conflicting. In addition, (e_2, e_4) and (e_7, e_9) are conflicting pairs of events in ρ_1 and we use an explicit arrow ($\longrightarrow$) to depict such inter-thread conflicting pairs. We have $T_1 \lessdot^{\rho_1}_{\mathtt{Txn}} T_2$ because $e_2 \leq^{\rho_1}_{\mathtt{CHB}} e_4$ and $T_3 \lessdot^{\rho_1}_{\mathtt{Txn}} T_1$ because $e_7 \leq^{\rho_1}_{\mathtt{CHB}} e_9$. Also note that $\leq_{\mathtt{CHB}}$ is a transitive order and thus $e_1 \leq^{\rho_1}_{\mathtt{CHB}} e_5$ because $e_1 \leq^{\rho_1}_{\mathtt{CHB}} e_2$, $e_2 \leq^{\rho_1}_{\mathtt{CHB}} e_4$ and $e_4 \leq^{\rho_1}_{\mathtt{CHB}} e_5$. Finally, the trace ρ_1 is conflict serializable and the equivalent serial execution is the sequence $\rho_1^{serial} = e_6e_7e_8e_1e_2e_9e_{10}e_3e_4e_5$, in which the order of transaction is $T_3T_1T_2$. Observe that the relative order of conflicting events in ρ_1^{serial} is the same as in the original trace ρ_1.*

Based on Definition 1, a cyclic dependency on transactions using $\lessdot^{\sigma}_{\mathtt{Txn}}$ suggests that σ does not have an equivalent serial execution and hence the program does not satisfy its atomicity specification. Previous techniques [5, 19] 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 $\lessdot_{\mathtt{Txn}}$ and checking violations of conflict serializability reduces to searching for a cycle in this graph. These algorithms run in time that is cubic in the length of

the observed trace as they check for cycles each time a new edge is added in the graph, whose size is quadratic in the size of the trace.

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 [14, 23, 24, 27, 43, 50, 52]. 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 $\lessdot_{\mathsf{Txn}}$ 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_1 is less than or equal to timestamp of transaction T_2 if and only if $T_1 \lessdot_{\mathsf{Txn}} T_2$. However, since a transaction is a *sequence of events* (and not a single event), the first challenge is figuring out how to assign 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 ρ_1 in Figure 1. Notice that there is a "path" from T_3 to T_2 (via T_1) using $\lessdot^{\rho_1}_{\mathsf{Txn}}$, even though T_3 starts after T_2 is completed in the trace ρ_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_3 have completed. This poses serious challenges when designing a vector clock algorithm. A vector clock algorithm assigning a timestamp to transaction T that is consistent with $\lessdot_{\mathsf{Txn}}$, 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 ρ_1 illustrates, this may require knowing future events and transactions.*

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

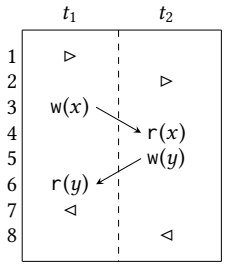

Figure 2. Trace ρ_2. There is a cycle in the transaction graph that can be realized by a path using $\leq_{\mathsf{CHB}}$-edges that begins and ends in the same transaction.

Example 3. *Consider the trace ρ_2 in Figure 2 with two transactions T_1 and T_2 in threads t_1 and t_2 respectively. Here, we have, $T_1 \lessdot^{\rho_2}_{\mathsf{Txn}} T_2$ and $T_2 \lessdot^{\rho_2}_{\mathsf{Txn}} T_1$, thus giving us a violation of conflict serializability with the sequence T_1, T_2 witnessing the violation. Now consider the following $\leq_{\mathsf{CHB}}$ path in the trace — $e_1 \leq^{\rho_2}_{\mathsf{CHB}} e_4 \leq^{\rho_2}_{\mathsf{CHB}} e_5 \leq^{\rho_2}_{\mathsf{CHB}} e_7$. 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 ρ_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 ρ_2, specifically) such that $\mathsf{txn}(e) = \mathsf{txn}(g)$, $\mathsf{txn}(e) \neq \mathsf{txn}(f)$, and $e \leq_{\mathsf{CHB}} f \leq_{\mathsf{CHB}} g$. 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 $\leq_{\mathsf{CHB}}$ ordering. Unfortunately, this is not true, i.e., violations of conflict serializability cannot be detected by simply using $\leq_{\mathsf{CHB}}$ ordering and searching for the above kind of $\leq_{\mathsf{CHB}}$ paths. We illustrate this in the next example.

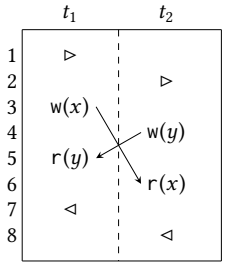

Figure 3. Trace ρ_3. There is no $\leq_{\mathsf{CHB}}$ path that starts and ends in the same transaction.

Example 4. *Consider trace ρ_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 $T_1 \lessdot^{\rho_3}_{\mathsf{Txn}} T_2$ (because $e_3 \leq^{\rho_3}_{\mathsf{CHB}} e_6$) and $T_2 \lessdot^{\rho_3}_{\mathsf{Txn}} T_1$ (because $e_4 \leq^{\rho_3}_{\mathsf{CHB}} e_5$), thus giving us a conflict serializability violation.*

However, there is no $\leq_{\mathsf{CHB}}$-path that starts and ends in the same transaction. If vector timestamps are used to compute $\leq_{\mathsf{CHB}}$, then violations of conflict serializability cannot be detected by checking ordering of vector timestamps of events.

Example 4 demonstrates that $\leq_{\mathsf{CHB}}$ 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 σ (denoted $e \xrightarrow{*}_{\sigma} f$), 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) $\mathsf{txn}(e_i) = \mathsf{txn}(f_i)$, while $\mathsf{txn}(f_i) \neq \mathsf{txn}(e_{i+1})$, for every i, and (c) $f_i \leq^{\sigma}_{\mathsf{CHB}} e_{i+1}$ 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 σ is not conflict serializable if and only if there is a pair of events e, f such that $e \xrightarrow{*}_{\sigma} f$ and $f \leq^{\sigma}_{\mathsf{CHB}} e$.*

Though $\xrightarrow{*}_{\sigma}$ 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 $\xrightarrow{*}_{\sigma}$, defined as follows.

Definition 2. *For events e, f in trace σ, we say $e \lessdot^{\sigma}_{\mathsf{E}} f$, if there is an event g in σ such that $e \leq^{\sigma}_{\mathsf{CHB}} g$ and either (a) $g = f$, or (b) $g \xrightarrow{*}_{\sigma} f$ and $\mathsf{txn}(g)$ is completed in σ.*

The following theorem formalizes how we can check for conflict serializability violations using the new relation. The proof of this theorem is presented in [44].

Theorem 2. *For a transaction T, let $T_{\rhd}$ denote the begin transaction event $\langle \cdot, \rhd \rangle$ of T. The following observations hold.*

1. *Any trace σ with a transaction T, events e and f such that $f \in T$, $e \notin T$, $T_{\rhd} \lessdot^{\sigma}_{\mathsf{E}} e$ and $e \lessdot^{\sigma}_{\mathsf{E}} f$, is not conflict serializable.*
2. *Let σ be a trace that is not conflict serializable with a witness $T_0, \ldots T_{k-1}$ such that each T_i, except possibly one, is complete in σ. Then there is a transaction T and events e, f in σ such that $f \in T$, $e \notin T$, $T_{\rhd} \lessdot^{\sigma}_{\mathsf{E}} e$ and $e \lessdot^{\sigma}_{\mathsf{E}} f$.*

We conclude this section with examples illustrating both the definition $\lessdot_{\mathsf{E}}$ and the use of Theorem 2.

Example 5. *Let us begin by looking at trace ρ_3 in Figure 3. Let σ_i denote the prefix of ρ_3 upto (and including) event e_i. In trace σ_6, we have $e_3 \lessdot^{\sigma_6}_{\mathsf{E}} e_6$, $e_4 \lessdot^{\sigma_6}_{\mathsf{E}} e_5$, and $e_1 \lessdot^{\sigma_6}_{\mathsf{E}} e_6$ because they are related by $\leq_{\mathsf{CHB}}$. Here, $e_1 \xrightarrow{*}_{\sigma_6} e_4$ because $\mathsf{txn}(e_1) = \mathsf{txn}(e_3)$, $e_3 \leq^{\sigma_6}_{\mathsf{CHB}} e_6$ and $\mathsf{txn}(e_6) = \mathsf{txn}(e_4)$. However, it is not the case that $e_1 \lessdot^{\sigma_6}_{\mathsf{E}} e_4$. On the other hand, if we consider σ_7, then $e_1 \lessdot^{\sigma_7}_{\mathsf{E}} e_4$ as the transaction in t_1 is complete in σ_7. In σ_7 (and therefore also in the full trace ρ_3), conditions of Theorem 2 are satisfied — $e_1 \lessdot^{\sigma_7}_{\mathsf{E}} e_4$ and $e_4 \lessdot^{\sigma_7}_{\mathsf{E}} e_7$.*

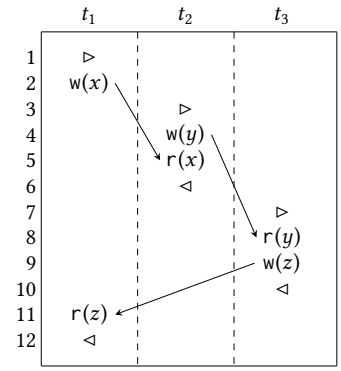

Figure 4. Trace ρ_4. Each transaction is a $\lessdot_{\mathsf{Txn}}$ predecessor of the other.

Example 6. *Consider trace ρ_4 in Figure 4; this is a slight modification of trace ρ_1 from Figure 1 that now has an atomicity violation. Again e_i denotes the ith event, and σ_i denotes the prefix upto event e_i. Notice that in prefix σ_{11}, $e_1 \lessdot^{\sigma_{11}}_{\mathsf{E}} e_5$ (because $e_1 \leq^{\sigma_{11}}_{\mathsf{CHB}} e_5$) and $e_5 \lessdot^{\sigma_{11}}_{\mathsf{E}} e_{11}$ (because $e_5 \xrightarrow{*}_{\sigma_{11}} e_{11}$ and $\mathsf{txn}(e_5)$ is complete in σ_{11}). Thus by Theorem 2, there is a violation of conflict serializability.*

4 Vector Clock Algorithm

Based on the 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 Thr. A vector time (or timestamp) is a vector of non-negative integers, whose size/dimension is |Thr| (number of threads). For a thread $t \in \mathsf{Thr}$, we denote the tth 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 $V_1 \sqsubseteq V_2$ if $\forall t \in \mathsf{Thr}.\ V_1(t) \leq V_2(t)$. In this case, we say that V_2 is *greater than, ordered after* or *after* V_1. The minimum vector time on threads Thr is $\bot_{\mathsf{Thr}} = \lambda t.\ 0$, 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 $V_1 \sqcup V_2 = \lambda t \cdot \max\{V_1(t), V_2(t)\}$. Finally, we use $V[c/t]$ to denote the timestamp $\lambda u.$ if $u = t$ 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 $\lessdot_{\mathsf{E}}$ (Definition 2) and use Theorem 2 to discover conflict serializability violations. The exact invariant maintained by the algorithm is technical and is presented in [44]. Similar to vector clock algorithms used in data race detection algorithms [14, 27, 50], 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 $\mathbb{C}_t$, for each thread $t \in \mathsf{Thr}$. The clock $\mathbb{C}_t$, intuitively, stores the timestamp of the last event performed by the thread t so far. That is, when performing an event $e = \langle t, op \rangle$, the timestamp assigned to e by AeroDrome is, in fact, determined by the value of the clock $\mathbb{C}_t$ 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+ [50] or its derivatives like FASTTRACK [14]. The precise definition of ''the timestamp associated with an event' is technical and is deferred to [44].

The algorithm also checks for violations of conflict serializability 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 $\mathbb{C}_t^{\rhd}$ 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 $\lessdot_{\mathsf{E}}$. Since $\lessdot_{\mathsf{E}}$ is defined using $\leq_{\mathsf{CHB}}$, we need to ensure that the vector timestamps reflect the orderings induced by $\leq_{\mathsf{CHB}}$. In order to capture the intra-thread dependencies imposed by $\leq_{\mathsf{CHB}}$ and $\lessdot_{\mathsf{E}}$, we need auxiliary clocks. Consider an event e of the form $\langle t, \mathsf{acq}(\ell) \rangle$. All previously encountered events with operations on lock ℓ are $\leq_{\mathsf{CHB}}$-before e. Hence the timestamp of e must be after those assigned to such events. To do this, AeroDrome will maintain a vector clock $\mathbb{L}_\ell$ for each lock ℓ, that stores the timestamp of the last $\mathtt{rel}(\ell)$ seen so far; this will be used 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 $\mathbb{W}_x$ that stores the timestamp of the last write $\mathtt{w}(x)$-event and a clock $\mathbb{R}_{t,x}$ that stores the time of the last $\langle t, \mathtt{r}(x) \rangle$-event.

Recall that, when considering paths between events through transactions ($\xrightarrow{*}$), we need to make sure that consecutive transactions along the path are distinct. AeroDrome tracks this constraint by maintaining scalar variables $\mathsf{lastRelThr}_\ell$ and $\mathsf{lastWThr}_x$, which store the identifier of the thread that performed the last release on ℓ and write on x, respectively.

4.1.2 Initialization and Updates to State.

Each of the clocks $\mathbb{C}_t$ 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 NIL.

As new events are observed in the trace, the algorithm updates these vector clocks in a manner that is consistent with tracking the $\lessdot_{\mathsf{E}}$-relation. When processing a begin event $e = \langle t, \rhd \rangle$, the algorithm first increments the *local* component of $\mathbb{C}_t$ (line 35 - '$\mathbb{C}_t := \mathbb{C}_t[\mathbb{C}_t(t) + 1]$'). To understand why, let e_{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) $e_{\text{prev}} \lessdot_{\mathsf{E}} e'$, and (b) $\neg(e \lessdot_{\mathsf{E}} e')$. The increment of the local component ensures that this relationship between e, e_{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_{prev}. Finally, AeroDrome updates $\mathbb{C}_t^{\rhd}$ with the timestamp of the current event e stored in $\mathbb{C}_t$.

When processing an acquire event $e = \langle t, \mathsf{acq}(\ell) \rangle$, the algorithm makes sure that the timestamp of e is ordered after the timestamp of the last $\mathtt{rel}(\ell)$-event e_ℓ in the trace so far. This is achieved by updating '$\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{L}_\ell$' in the procedure CHECKANDGET (invoked at line 15); the procedure CHECKANDGET also checks for conflict serializability violation before updating $\mathbb{C}_t$, but more on that later. Of course, if e_ℓ is performed by the same thread t (line 14), then, this is already ensured and no explicit update is required.

At a write event $e = \langle t, \mathtt{w}(x) \rangle$, 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 $\mathbb{W}_x$ to be the timestamp of e (see line 32) and $\mathsf{lastWThr}_x$ to t, thus preserving the semantics of the clock $\mathbb{W}_x$ and the scalar variable $\mathsf{lastWThr}_x$. The updates performed at a read event are similar.

At a fork event $e = \langle t, \mathtt{fork}(u) \rangle$, the algorithm updates the clock of the child thread u ('$\mathbb{C}_u := \mathbb{C}_u \sqcup \mathbb{C}_t$' in line 20) so that all events of u are ordered after e. At a join event

Algorithm 1 *AeroDrome: Vector Clock Algorithm for Checking Violation of Conflict Serializability*

1: **procedure** INITIALIZATION
2: **for** $t \in \mathsf{Thr}$ **do**
3: $\mathbb{C}_t := \bot[1/t]$; $\mathbb{C}_t^{\triangleright} := \bot$;
4: **for** $\ell \in \mathsf{Locks}$ **do**
5: $\mathbb{L}_\ell := \bot$; $\mathsf{lastRelThr}_\ell := \mathsf{NIL}$;
6: **for** $x \in \mathsf{Vars}$ **do**
7: $\mathbb{W}_x := \bot$; $\mathsf{lastWThr}_x := \mathsf{NIL}$;
8: **for** $t \in \mathsf{Thr}$ **do** $\mathbb{R}_{t,x} := \bot$;

9: **procedure** CHECKANDGET(clk, t)
10: **if** $\mathbb{C}_\mathsf{t}^{\triangleright} \sqsubseteq \mathsf{clk}$ and t has an active transaction **then**
11: declare '**conflict serializability violation**';
12: $\mathbb{C}_\mathsf{t} := \mathbb{C}_\mathsf{t} \sqcup \mathsf{clk}$;

13: **procedure** ACQUIRE(t, ℓ)
14: **if** $\mathsf{lastRelThr}_\ell \neq t$ **then**
15: CHECKANDGET($\mathbb{L}_\ell$, t);

16: **procedure** RELEASE(t, ℓ)
17: $\mathbb{L}_\ell := \mathbb{C}_t$;
18: $\mathsf{lastRelThr}_\ell := t$;

19: **procedure** FORK(t, u)
20: $\mathbb{C}_u := \mathbb{C}_u \sqcup \mathbb{C}_t$;

21: **procedure** JOIN(t, u)
22: CHECKANDGET($\mathbb{C}_u$, t);

23: **procedure** READ(t, x)
24: **if** $\mathsf{lastWThr}_x \neq t$ **then**
25: CHECKANDGET($\mathbb{W}_x$, t);
26: $\mathbb{R}_{t,x} := \mathbb{C}_t$;

27: **procedure** WRITE(t, x)
28: **if** $\mathsf{lastWThr}_x \neq t$ **then**
29: CHECKANDGET($\mathbb{W}_x$, t);
30: **for** $u \in \mathsf{Thr} \setminus \{t\}$ **do**
31: CHECKANDGET($\mathbb{R}_{u,x}$, t);
32: $\mathbb{W}_x := \mathbb{C}_t$;
33: $\mathsf{lastWThr}_x = t$;

34: **procedure** BEGIN(t)
35: $\mathbb{C}_t(t) := \mathbb{C}_t(t) + 1$;
36: $\mathbb{C}_t^{\triangleright} := \mathbb{C}_t$;

37: **procedure** END(t)
38: **for** $u \in \mathsf{Thr} \setminus \{t\}$ **do**
39: **if** $\mathbb{C}_t^{\triangleright} \sqsubseteq \mathbb{C}_u$ **then**
40: CHECKANDGET($\mathbb{C}_t$, u);
41: **for** $\ell \in \mathsf{Locks}$ **do**
42: $\mathbb{L}_\ell := \mathbb{C}_t^{\triangleright} \sqsubseteq \mathbb{L}_\ell \ ? \ \mathbb{C}_t \sqcup \mathbb{L}_\ell : \mathbb{L}_\ell$;
43: **for** $x \in \mathsf{Vars}$ **do**
44: $\mathbb{W}_x := \mathbb{C}_t^{\triangleright} \sqsubseteq \mathbb{W}_x \ ? \ \mathbb{C}_t \sqcup \mathbb{W}_x : \mathbb{W}_x$;
45: **for** $u \in \mathsf{Thr}$ **do**
46: $\mathbb{R}_{u,x} := \mathbb{C}_t^{\triangleright} \sqsubseteq \mathbb{R}_{u,x} \ ? \ \mathbb{C}_t \sqcup \mathbb{R}_{u,x} : \mathbb{R}_{u,x}$;

$e = \langle t, \mathtt{join}(u)\rangle$, the algorithm updates $\mathbb{C}_t$ to $\mathbb{C}_t \sqcup \mathbb{C}_u$ so that all events of thread u are ordered before e.

Let us now consider the updates performed at an end-transaction event $e = \langle t, \triangleleft\rangle$. Let $e^{\triangleright}$ denote the matching begin transaction event. Observe that for an event f, if $e^{\triangleright} \lessdot_\mathsf{E} f$, then $e \lessdot_\mathsf{E} f$ because $\mathsf{txn}(e)$ is completed in σ. That is, all future events that are $\lessdot_\mathsf{E}$-after $e^{\triangleright}$ must be assigned a timestamp after that of e. This is ensured by updating clocks $\mathbb{C}_u$ for all threads u that satisfy $\mathbb{C}_t^{\triangleright} \sqsubseteq \mathbb{C}_u$ (lines 38 to 40), and clocks $\mathbb{L}_\ell$, $\mathbb{W}_x$, and $\mathbb{R}_{u,x}$ (lines 41 to 46).

4.1.3 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 $e = \langle t, op\rangle$ such that there is an earlier event e' (performed by a thread $t' \neq t$) that conflicts with e (and thus $e' \lessdot_\mathsf{E} e$). In this case, if $e^{\triangleright} \lessdot_\mathsf{E} e'$ (where $e^{\triangleright}$ is the begin event of $\mathsf{txn}(e)$), then there is an atomicity violation as per Theorem 2. 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 $e = \langle t, \triangleleft\rangle$ (with a matching begin event $e^{\triangleright}$). The algorithm detects a violation when there is another thread $u \neq t$ having an active transaction, with begin event $e_u^{\triangleright}$ and last event is e_u, such that $e^{\triangleright} \lessdot_\mathsf{E} e_u$ (line 39) and $e_u^{\triangleright} \lessdot_\mathsf{E} e$ (line 40). These checks for violations of conflict serializability are performed in CHECKANDGET (line 9), which takes two arguments: clk (vector timestamp) and t (thread identifier), and declares a violation if (a) thread t has an active transaction, and (b) clk is ordered after $\mathbb{C}_t^{\triangleright}$, which is the timestamp of the begin event of the (active) transaction of t (line 10). Whenever a violation is found, the algorithm exits. Otherwise, the algorithm continues after updating the value of the clock $\mathbb{C}_\mathsf{t}$ to $\mathbb{C}_\mathsf{t} \sqcup \mathsf{clk}$ (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 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 consitute a trivial atomic block, namely, one consisting of only that single event. These were called *unary* transactions in [19]. Our algorithm does not report a violation at unary transactions (in the procedure CHECKANDGET, lines 10 and 11) as these are not active transactions. The algorithm, nevertheless, is still correct as a unary transaction (corresponding to a read, write, acquire or join event) can only correspond to a cycle that involves another non-unary transactions.

We conclude this section with a theorem stating the correctness of Algorithm 1 (proof can be found in [44]).

Theorem 3. *On any trace σ, Algorithm 1 reports a violation of conflict serializability iff σ is not conflict serializable with a witness $T_0, \ldots T_{k-1}$ such that each T_i, except possibly one, is complete in σ.*

4.2 AeroDrome on Example Traces

Let us illustrate 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 AeroDrome.

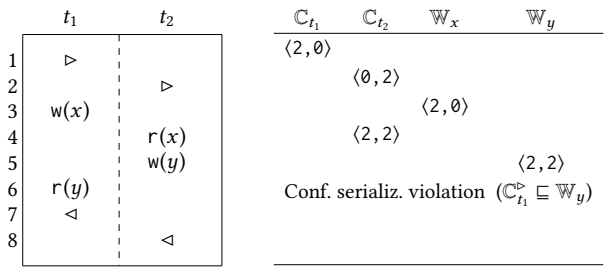

Figure 5. AeroDrome on Trace ρ_2.

Let us begin with the simplest trace ρ_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 ith event e_i in the trace. We do not show the values of the clocks $\mathbb{R}_{t_1,x}$, $\mathbb{R}_{t_2,x}$. $\mathbb{R}_{t_1,y}$ or $\mathbb{R}_{t_1,y}$ as they are not important here. There are two threads and thus the size of each vector clock is 2. The clocks $\mathbb{C}_{t_1}$ and $\mathbb{C}_{t_2}$ 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 $\mathbb{C}_{t_1}$ and $\mathbb{C}_{t_2}$ become $\langle 2,0\rangle$ and $\langle 0,2\rangle$ after e_2. Further, these are also the values of the clocks $\mathbb{C}^{\triangleright}_{t_1}$ and $\mathbb{C}^{\triangleright}_{t_2}$ from this point onwards until the end of the execution. After processing $e_3 = \langle t_1, \mathtt{w}(x)\rangle$, the value of the clock $\mathbb{W}_x$ 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 $\mathbb{C}_{t_2}$ to $\langle 2,2\rangle$ (line 12). The clock $\mathbb{W}_y$ gets the value of $\mathbb{C}_{t_2} = \langle 2,2\rangle$ 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 $\mathbb{C}^{\triangleright}_{t_1} \sqsubseteq \mathbb{W}_y$ and declares an atomicity violation.

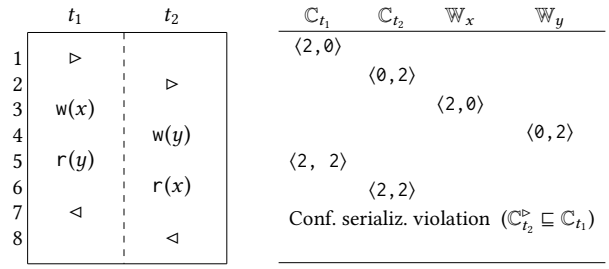

Figure 6. AeroDrome on Trace ρ_3.

Let us next consider the trace ρ_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, $\mathbb{C}_{t_1}$ 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, $\mathbb{C}^{\triangleright}_{t_1} = \langle 2,0\rangle$ and the clock $\mathbb{W}_y$ is $\langle 0,2\rangle$ thus failing the check $\mathbb{C}^{\triangleright}_{t_1} \sqsubseteq \mathbb{W}_y$ in line 10. The same explanation applies to the $\mathtt{r}(x)$ event e_6 in t_2 and thus no atomicity violation is reported here as well. Next, the algorithm processes the end event $e_7 = \langle t_1, \triangleleft\rangle$. At this point, the algorithm checks if any event in the currently active transaction of t_2 is ordered after e_1 (condition $\mathbb{C}^{\triangleright}_{t_1} \sqsubseteq \mathbb{C}_{t_2}$ in line 39 of Algorithm 1). This check succeeds since $\mathbb{C}^{\triangleright}_{t_1} = \langle 2,0\rangle$ and $\mathbb{C}_{t_2} = \langle 2,2\rangle$ at this point. The algorithm then checks if $\mathbb{C}^{\triangleright}_{t_2} \sqsubseteq \mathbb{C}_{t_1}$ 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 ρ_4 from Figure 4. This example illustrates how AeroDrome handles dependencies between transactions introduced by future events. The run of AeroDrome on ρ_4 is shown in Figure 7. We omit the updates to the clocks $\mathbb{R}_{t_i,u}$ ($i \in \{1,2,3\}$, $u \in \{x,y,z\}$) as they do not play a significant role in this example. All vector clocks have dimension 3 because there are three threads in ρ_4. As before, the clocks are initialized as follows: $\mathbb{C}_{t_1} = \langle 1,0,0\rangle$, $\mathbb{C}_{t_2} = \langle 0,1,0\rangle$ and $\mathbb{C}_{t_3} = \langle 0,0,1\rangle$; all other clocks are initialized to $\langle 0,0,0\rangle$. The begin events result in incrementing of local clocks and thus $\mathbb{C}_{t_1} = \langle 2,0,0\rangle$ after e_1. Further, the clock $\mathbb{W}_x$ gets updated to the value of $\mathbb{C}_{t_1}$ at the end of e_2. The next two events e_3 and e_4 are processed in a similar fashion. At event $e_5 = \langle t_2, \mathtt{r}(x)\rangle$, the clock $\mathbb{C}_{t_2}$ gets updated to $\langle 2,2,0\rangle$ (line 12 in Algorithm 1). 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_3 have clock values larger than $\mathbb{C}^{\triangleright}_{t_2}$ (line 39). However the write and read clocks are updated. Specifically, the clock $\mathbb{W}_y$ maintaining the timestamp to the last write to y is such that $\mathbb{C}^{\triangleright}_{t_2} \sqsubseteq \mathbb{W}_y$ and thus, the algorithm updates $\mathbb{W}_y$ to $\mathbb{W}_y \sqcup \mathbb{C}_{t_2} = \langle 2,2,0\rangle$ (line 44 in Algorithm 1). Event e_7 is a begin event and updates $\mathbb{C}_{t_3}$ to

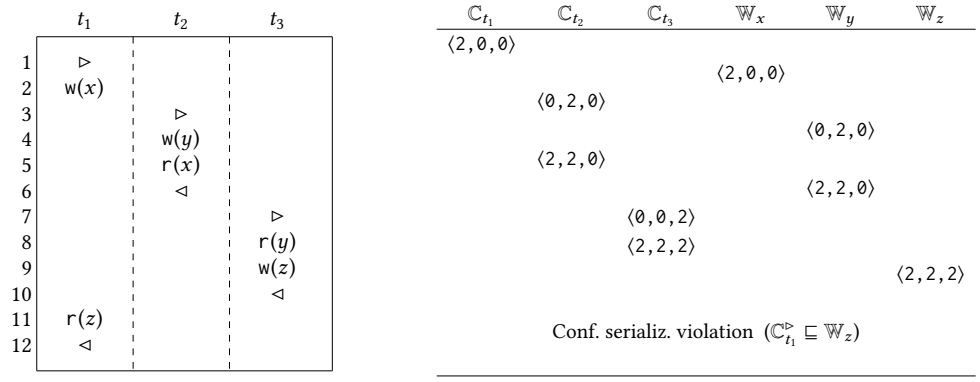

Figure 7. AeroDrome on Trace ρ_4.

$\langle 0,0,2\rangle$. Now at the $\mathtt{r}(y)$ event e_8, the clock $\mathbb{C}_{t_3}$ gets updated with $\mathbb{W}_y$ which at this point evaluates to $\langle 2,2,0\rangle$, thus giving $\mathbb{C}_{t_3} = \langle 2,2,2\rangle$. The write clock $\mathbb{W}_z$ then gets updated to $\langle 2,2,2\rangle$ after e_9. More clock updates happen at e_{10} (though not shown in Figure 7) Finally, an atomicity violation is detected at event $e_{11} = \langle t_1, \mathtt{r}(z)\rangle$; the algorithm checks if the clock $\mathbb{W}_z$ *knows* some event in t_1 ($\mathbb{C}^{\triangleright}_{t_1} \sqsubseteq \mathbb{W}_z$) and declares 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 $\mathbb{R}_{t,x}$ 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 $O(|\mathsf{Thr}|V)$, where $|\mathsf{Thr}|$ 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 $O(|\mathsf{Thr}|V)$ to $O(V)$. To understand the optimization, we need to first understand the role served by clocks $\mathbb{R}_{t,x}$. First, these clocks help detect atomicity violation — at a write event $e = \langle t, \mathtt{w}(x)\rangle$, a violation is reported if there is a thread $u \neq t$ such that $\mathbb{C}^{\triangleright}_t \sqsubseteq \mathbb{R}_{u,x}$ (line 10 invoked from line 31 in Algorithm 1). Second, these clocks are used to update $\mathbb{C}_t$ — at a write event $e = \langle t, \mathtt{w}(x)\rangle$, we set $\mathbb{C}_t := \bigsqcup_{u\neq t} \mathbb{C}_t \sqcup \mathbb{R}_{u,x}$ (line 12 invoked iteratively at line 31).

The reduction in the number of clocks is achieved by instead maintaining one clock (per memory location) for each of the above two purposes instead of maintaining $O(|\mathsf{Thr}|)$ many clocks (per memory location). First, for updating clocks correctly at write events, we will maintain a single clock $\mathbb{R}_x$ for each location x. This clock stores the value $\bigsqcup_u \mathbb{R}_{u,x}$ at each point while processing the trace. Next, to perform checks for violations of conflict serializability, we will have another clock $\mathbb{chR}_x$ (*check read*). This clock will store the value $\bigsqcup_u \mathbb{R}_{u,x}[0/u]$ at each point in the analysis. Based on the invariants maintained by the algorithm, one can show that checking $\mathbb{C}^{\triangleright}_t \sqsubseteq \bigsqcup_{u\neq t} \mathbb{R}_{u,x}$ is equivalent to checking $\mathbb{C}^{\triangleright}_t \sqsubseteq \mathbb{chR}_x$. This optimization and other useful optimizations that improve the performance of AeroDrome, are outlined in greater detail in [44].

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_{end} for the number of non-end events and end events in the trace (and thus $n = n_{\text{non-end}} + n_{\text{end}}$ is the size of the trace). We will denote by $|\mathsf{Thr}|$, 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* $O(|\mathsf{Thr}|(n_{non\text{-}end} + (|\mathsf{Thr}| + L + V)n_{end}))$ *time and* $O(|\mathsf{Thr}|(|\mathsf{Thr}| + V + L))$ *space.*

The complexity observations easily follow from the description of the algorithm and the optimization discussed in Section 4.3.

5 Experimental Evaluation

In this section, we describe our implementation of AeroDrome and the results of evaluating it on benchmark programs. Appendix A discusses the accompanied artifact that describes our overall experimental workflow and can be used to replicate our results.

5.1 Implementation

We have implemented AeroDrome in a prototype tool Rapid, available publicly [41]. Rapid is written in Java and analyzes traces generated by concurrent programs to detect 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 Rapid therefore first extracts an execution trace from a concurrent programs and then analyzes the same trace against all candidate algorithms. We use RoadRunner [15] 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) cubic time and analyzes traces by building a directed graph, with transactions as nodes in the graph and where the edges correspond to $\prec_{\mathsf{Txn}}$ relation between transactions. There was no publicly available implementation of Velodrome that analyzes logged executions. Thus, we also implement this algorithm in Rapid. We use the Java graph library JGraphT [46] 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 DoubleChcker's publicly available implementation [4] 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 and Benchmarks

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` or the static `main` methods in Java are often not intended to be atomic. Thus, atomicity specifications need to be specially identified by developers, by supplying manual annotations [20]. In the absence of such static annotations, we use atomicity specifications from prior work [5] whenever possible (Table 1). For the benchmarks (Table 2) for which no specifications were available, we declare all methods except the `main` and `run` methods to be atomic.

Benchmarks and Setup. Our benchmark programs (Table 1 and Table 2) are derived from the DaCaPo benchmark suite [6] adapted to run with RoadRunner [15], Java Grande Forum [57] and microbenchmarks from [59] and have been used in prior work [5]. 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 in Column 1. Observe that the number of events in the execution traces can vary from a few hundred to billions of events and our algorithm can scale to such large traces. 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 '✗' 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' represents timeout after 10 hours. Column 10 reports the speed-up of AeroDrome over Velodrome.

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.

Table 1. Trace characteristics and running times for benchmarks with atomicity specifications from DoubleChecker.

1	2	3	4	5	6	7	8	9	10
Program	Events	Threads	Locks	Variables	Transactions	Atomic?	Velodrome (s)	AeroDrome (s)	Speed-up
avrora	2.4B	7	7	1079K	498M	✗	TO	1.5	> 24000
elevator	280K	5	50	725	22.6K	✓	162	1.7	97
hedc	9.8K	7	13	1694	84	✗	0.07	0.06	1.16
luindex	570M	3	65	2.5M	86M	✗	581	674	0.86
lusearch	2.0B	14	772	38M	306M	✗	TO	5.5	> 6545
moldyn	1.7B	4	1	121K	1.4M	✗	TO	54.9	> 650
montecarlo	494M	4	1	30.5M	812K	✗	TO	0.75	> 48000
philo	613	6	1	24	0	✓	0.02	0.02	1
pmd	367M	13	223	12.9M	81M	✗	3.1	3.8	0.82
raytracer	2.8B	4	1	12.6M	277M	✓	TO	55m40s	> 10.7
sor	608M	4	2	1M	637K	✗	6.9	9.6	0.72
sunflow	16.8M	16	9	1.2M	2.5M	✗	67.9	0.65	104.5
tsp	312M	9	2	181M	9	✗	4.2	5.7	0.73
xalan	1.0B	13	8624	31M	214M	✗	1.6	2.0	0.8

Table 2. Trace characteristics and running times for benchmarks with naive atomicity specifications.

1	2	3	4	5	6	7	8	9	10
Program	Events	Threads	Locks	Variables	Transactions	Atomic?	Velodrome (s)	AeroDrome (s)	Speed-up
batik	186M	7	1916	4.9M	15M	✗	52.7	65.5	0.81
crypt	126M	7	1	9M	50	✗	92.1	104	0.88
fop	96M	1	115	5M	25M	✓	88.3	92.5	0.95
lufact	135M	4	1	252K	642M	✗	2.4	2.9	0.82
series	40M	4	1	20K	20M	✗	61.0	15.3	3.98
sparsematmult	726M	4	1	1.6M	25	✗	1210	1197	1.01
tomcat	726M	4	1	1.6M	25	✗	3.4	4.5	0.75

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. In this benchmark, the number of nodes in the graph analyzed by Velodrome is about 9000, at the point where the violation is reported. This coupled with the cubic 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 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 AeroDrome does not give a huge speedup over Velodrome, we discovered that the number of nodes in Velodrome's graph analysis is fairly small owing to garbage collection; 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 performance of Velodrome is comparable to that of our algorithm AeroDrome. 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 nodes in the transaction graph constructed by Velodrome 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, and as a result, the graph based algorithm runs faster.

6 Related Work

Multi-threaded programs are challenging to reason about. Atomicity is a principled concept that lets programmers reason about coarse behaviors of programs, without being concerned about fine grained thread interleavings. Ensuring atomicity of concurrent program blocks is therefore an important question [37] 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 [17, 20]. These type systems rely on commutativity of operations and are inspired from Lipton's theory of reduction [33] and the concept of purity [18]. Extensions to type inference [54] and to programs with non-blocking synchronization [62] have been developed. The work in [17] 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 [33] has been a prominent theme in this space, most notably the analysis employed by Atomizer [13]. This approach however leads to false alarms. The notion of conflict serializability was introduced concurrently by Flanagan et. al. [19] and Farzan et. al. [12], inspired from the theory of concurrency control in databases [47]. However, Farzan et. al. [12] do not account for any lock operations which are crucially used in most Java like concurrent programs, making their algorithm prone to false positives. Further, their algorithm relies on maintaining sets of locks, threads and variables, similar in spirit to the Goldilocks algorithm [10] for detecting HB races. As in the case of data race detection [14, 28], such an algorithm is expected to be orders of magnitude slower than a vector clock algorithm for the same problem. More importantly, the algorithm in [12] is automata-theoretic, warranting a global centralized observer that analyzes events in a serial fashion. In contrast, our algorithm AeroDrome allows for a distributed implementation — one can attach the analysis metadata (vector clocks and other scalar variables, in our case) to the various objects (like threads, locks and memory locations) being tracked. The analysis can then be performed with only little synchronization between these metadata, allowing our vector clock algorithm to leverage parallelism. 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. Causal atomicity [11] is a weaker criterion for atomicity and asks if there is an equivalent trace where one particular transaction (instead of all transactions) 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 [49] and AVIO [39] resort to directed exploration of thread interleavings to expose subtle atomicity violations. Penelope [58] detects 2 thread atomicity violations using directed interleaving exploration. The work in [2, 38, 63, 64] is also based on exercising specific thread schedules. SMT solving based predictive analysis techniques [60] have been developed, but tend to not scale. The work of Samak et. al. [53] synthesizes directed unit tests for catching atomicity violations. The work in [11, 55] 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 [26, 48].

Like most concurrency bugs, atomicity bugs are hard to fix. Naive fixes such as enforcing atomic regions using locks can introduce new bugs, affect the performance of programs and moreover can be inadequate in ensuring atomicity. Several approaches have been proposed [25, 30, 30–32, 34, 36] for automated repair of atomicity violation 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 atomicity [11], view serializability [63] or reduction based atomicity characterizations as in [13, 64]. Other promising lines of work include improving the efficiency of the proposed dynamic analysis for atomicity by incorporating ideas from data race detection. This includes the classic *epoch* optimizations [14], static analysis for redundancy elimination [16] and optimal check placement [51], and advances concerning instrumentation [7, 8, 65, 66].

Acknowledgments

We thank the anonymous reviewers for several comments that helped improve the paper. Umang Mathur is partially supported by a Google PhD Fellowship. Mahesh Viswanathan is partially supported by NSF CCF 1901069.

References

[1] 2019. *Trace logs used in Section 5.* https://drive.google.com/drive/folders/10tW4fL1iWp8MSrmh-Kaj_qqztjAGzLmf?usp=sharing

[2] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. 2005. Optimized Run-time Race Detection and Atomicity Checking Using Partial Discovered Types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. ACM, New York, NY, USA, 233–242. https://doi.org/10.1145/1101908.1101944

[3] Rahul Agarwal and Scott D. Stoller. 2004. Type Inference for Parameterized Race-Free Java. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–160.

[4] Swarnendu Biswas. 2014. *DoubleChecker.* https://sourceforge.net/p/jikesrvm/research-archive/45/ Accessed: 2020-01-15.

[5] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 28–39. https://doi.org/10.1145/2594291.2594323

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. OCTET: Capturing and Controlling Cross-Thread Dependences Efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 693–712. https://doi.org/10.1145/2509136.2509519

[8] Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. 2016. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-thread Dependences. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. ACM, New York, NY, USA, Article 20, 13 pages. https://doi.org/10.1145/2851141.2851143

[9] Lee Chew and David Lie. 2010. Kivati: Fast Detection and Prevention of Atomicity Violations. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 307–320. https://doi.org/10.1145/1755913.1755945

[10] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

[11] Azadeh Farzan and P. Madhusudan. 2006. Causal Atomicity. In *Proceedings of the 18th International Conference on Computer Aided Verification* (Seattle, WA) *(CAV '06)*. Springer-Verlag, Berlin, Heidelberg, 315–328. https://doi.org/10.1007/11817963_30

[12] Azadeh Farzan and P. Madhusudan. 2008. Monitoring Atomicity in Concurrent Programs. In *Proceedings of the 20th International Conference on Computer Aided Verification* (Princeton, NJ, USA) *(CAV '08)*. Springer-Verlag, Berlin, Heidelberg, 52–65. https://doi.org/10.1007/978-3-540-70545-1_8

[13] Cormac Flanagan and Stephen N Freund. 2004. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/964001.964023

[14] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[15] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. ACM, New York, NY, USA, 1–8. https://github.com/stephenfreund/RoadRunner

[16] Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 255–280.

[17] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for Atomicity: Static Checking and Inference for Java. *ACM Trans. Program. Lang. Syst.* 30, 4, Article 20 (Aug. 2008), 53 pages. https://doi.org/10.1145/1377492.1377495

[18] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2004. Exploiting Purity for Atomicity. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 221–231. https://doi.org/10.1145/1007512.1007543

[19] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 293–303. https://doi.org/10.1145/1375581.1375618

[20] Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. ACM, New York, NY, USA, 338–349. https://doi.org/10.1145/781131.781169

[21] Cormac Flanagan and Shaz Qadeer. 2003. Types for Atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA) *(TLDI '03)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/604174.604176

[22] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3064176.3064183

[23] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 179 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360605

[24] Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. 1999. Toward Integration of Data Race Detection in DSM Systems. *J. Parallel Distrib. Comput.* 59, 2 (Nov. 1999), 180–203. https://doi.org/10.1006/jpdc.1999.1574

[25] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[26] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 675–681.

[27] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI '17)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[28] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2018. Data Race Detection on Compressed Traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3236024.3236025

[29] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[30] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 994–1009. https://doi.org/10.1145/3314221.3314620

[31] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: Fixing Concurrency Bugs Based on Memory Access Patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 589–600. https://doi.org/10.1145/3238147.3238198

[32] Yiyan Lin and Sandeep S. Kulkarni. 2014. Automatic Repair for Multi-Threaded Programs with Deadlock/Livelock Using Maximum Satisfiability. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/2610384.2610398

[33] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. https://doi.org/10.1145/361227.361234

[34] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and Generating High Quality Patches for Concurrency Bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 715–726. https://doi.org/10.1145/2950290.2950309

[35] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 677–691. https://doi.org/10.1145/3037697.3037735

[36] Peng Liu and Charles Zhang. 2012. Axis: Automatically Fixing Atomicity Violations through Solving Control Constraints. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 299–309.

[37] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[38] Shan Lu, Soyeon Park, and Yuanyuan Zhou. 2012. Finding Atomicity-Violation Bugs Through Unserializable Interleaving Testing. *IEEE Trans. Softw. Eng.* 38, 4 (July 2012), 844–860. https://doi.org/10.1109/TSE.2011.35

[39] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 37–48. https://doi.org/10.1145/1168857.1168864

[40] Umang Mathur. 2019. *Artifact for "Atomicity Checking in Linear Time using Vector Clocks"*. https://doi.org/10.5281/zenodo.3605759

[41] Umang Mathur. 2019. *RAPID*. https://github.com/umangm/rapid Accessed: 2020-01-15.

[42] Umang Mathur. 2020. *umangm/rapid v1.1*. https://doi.org/10.5281/zenodo.3605709

[43] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 145 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276515

[44] Umang Mathur and Mahesh Viswanathan. 2020. Atomicity Checking in Linear Time using Vector Clocks. *CoRR* abs/2001.04961 (2020). arXiv:2001.04961 https://arxiv.org/abs/2001.04961

[45] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

[46] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V Sichi. 2019. JGraphT–A Java library for graph data structures and algorithms. *arXiv preprint arXiv:1904.08355* (2019).

[47] Christos Papadimitriou. 1986. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., New York, NY, USA.

[48] Chang-Seo Park and Koushik Sen. 2008. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 135–145. https://doi.org/10.1145/1453101.1453121

[49] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[50] Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

[51] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: Static Check Placement for Dynamic Race Detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 141–156. https://doi.org/10.1145/3062341.3062350

[52] Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

[53] Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing Tests for Detecting Atomicity Violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 131–142. https:

//doi.org/10.1145/2786805.2786874

[54] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2005. Automated Type-Based Analysis of Data Races and Atomicity. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/1065944.1065956

[55] Koushik Sen and Mahesh Viswanathan. 2006. Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In *Proceedings of the 18th International Conference on Computer Aided Verification* (Seattle, WA) *(CAV'06)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[56] Ilya Sergey. 2019. *What Does It Mean for a Program Analysis to Be Sound?* https://blog.sigplan.org/2019/08/07/what-does-it-mean-for-a-program-analysis-to-be-sound Accessed: 2020-01-15.

[57] Lorna A Smith and J Mark Bull. 2001. A multithreaded java grande benchmark suite. In *Proceedings of the third workshop on Java for high performance computing*.

[58] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

[59] Christoph von Praun and Thomas R. Gross. 2003. Static Conflict Analysis for Multi-threaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. ACM, New York, NY, USA, 115–128. https://doi.org/10.1145/781131.781145

[60] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. 2010. Trace-Based Symbolic Analysis for Atomicity Violations. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Paphos, Cyprus) *(TACAS'10)*. Springer-Verlag, Berlin, Heidelberg, 328–342.

[61] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 520–531.

[62] Liqiang Wang and Scott D. Stoller. 2005. Static Analysis of Atomicity for Programs with Non-blocking Synchronization. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. ACM, New York, NY, USA, 61–71. https://doi.org/10.1145/1065944.1065953

[63] Liqiang Wang and Scott D. Stoller. 2006. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA) *(PPoPP '06)*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/1122971.1122993

[64] Liqiang Wang and Scott D. Stoller. 2006. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Trans. Softw. Eng.* 32, 2 (Feb. 2006), 93–110. https://doi.org/10.1109/TSE.2006.1599419

[65] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. 2015. Array Shadow State Compression for Precise Dynamic Race Detection (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 155–165. https://doi.org/10.1109/ASE.2015.19

[66] Benjamin P. Wood, Man Cao, Michael D. Bond, and Dan Grossman. 2017. Instrumentation Bias for Dynamic Data Race Detection. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 69 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133893

[67] Min Xu, Rastislav Bodík, and Mark D. Hill. 2005. A Serializability Violation Detector for Shared-Memory Server Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/1065010.1065013

A Artifact Appendix

A.1 Abstract

This artifact appendix describes how to replicate our results from Section 5. Our evaluation comprises of generating trace logs of benchmark programs from Table 1 and Table 2, and running AeroDrome and Velodrome [19] analyses on them. We expect the speed-ups of AeroDrome over Velodrome to be similar to those reported in Table 1 and Table 2. All analyses are implemented in our tool Rapid and we provide Python scripts for automating the workflow.

A.2 Artifact check-list (meta-information)

- **Algorithm:** AeroDrome.
- **Program:** Provided with the artifact (also see Section 5.2).
- **Data set:** Instructions and scripts for generating trace logs from benchmarks programs have been provided. Trace logs used in our original experiments can be downloaded from [1].
- **Execution:** Experiments to be conducted as sole user. Generating trace logs from scratch can take several hours for large benchmarks.
- **How much disk space required (approximately)?:** Approximately 500GB space required to save trace logs. Individual trace logs can be as large as 100GB.
- **How much time is needed to prepare workflow (approximately)?:** All scripts are provided.
- **How much time is needed to complete experiments (approximately)?:** If all traces need to be generated, then about a day. If traces are obtained from [1], then as much as the timeout set. We used a timeout of 10 hours per benchmark.
- **Publicly available?:** Yes. Artifact available at [40]. Rapid available at [41] (archived at [42]).
- **Code licenses (if publicly available)?:** MIT License.
- **Data licenses (if publicly available)?:** None.
- **Archived (provide DOI)?:** Yes [40].

A.3 Description

A.3.1 How to access. Publicly available [40]. It extracts to less than 250MB.

A.3.2 Hardware dependencies. No special hardware required.

A.3.3 Software dependencies. Java 1.8 or higher, Ant 1.10 or higher, Python 2.7 or higher.

A.3.4 Data sets. Traces can be generated using benchmark programs provided. Alternatively, they can be downloaded from [1].

A.4 Installation

Obtain the artifact from [40] and extract.

A.5 Experiment workflow

A.5.1 Directory Structure. The overall directory of the artifact is shown in Figure 8. The directory `benchmarks/` contains our benchmark programs. The directory `atomicity_specs/` contain atomicity specifications for each benchmark (see Section 5.2). The directory `scripts/` contains our scripts for automating the workflow. The directory `RoadRunner` has been obtained from [15]. `README.md` is a more verbose description of the experimental workflow, and `LICENSE.txt` is an MIT License agreement for the artifact.

```
AE/
|--- LICENSE.txt
|--- README.md
|--- RoadRunner/
|--- atomicity_specs/
|--- benchmarks/
|--- scripts/
```

Figure 8. Directory structure of the artifact

A.5.2 Overall Workflow. The overall workflow is as follows.

1. **Generating Trace Logs**. We need to generate trace logs from benchmark programs. There are two options here:
   (a) **Option-1.** Download trace logs directly from [1].
   (b) **Option-2** (time consuming). Use RoadRunner to generate raw trace logs and then filter those based on the provided atomicity specifications, described below.
      (i) **Logging.** We will use the logging and instrumentation facility provided by RoadRunner [15] to generate traces.
      (ii) **Filtering.** We will filter out some events based on atomicity specifications in `atomicity_specs/`.
2. **Performing Analyses** We then analyze the final trace logs (obtained int he previous step) using Rapid [41]. Rapid can perform several kinds of analyses on a trace log:
   - The class `MetaInfo` can be used to determine basic information about the log, including the total number of events, threads, variables, locks etc.
   - The class `Aerodrome` determines atomicity violations using our proposed algorithm Aerodrome.
   - The class `Velodrome` determines atomicity violations using the prior state-of-the-art algorithm Velodrome [19].

A.5.3 Getting Started. Downloaded the artifact from [40] and set `$AE_HOME`:

```
> export AE_HOME=/path/to/AE/
```

Also, you need to change the variable `home` in the file `scripts/util.py` (line 17) to be the value of `$AE_HOME`. Also set the environment variables `JAVA_HOME` and `JVM_ARGS` in the same file appropriately. Next download Rapid from GitHub [41] or from the archive [42] in `$AE_HOME/rapid/` and install:

```
> cd $AE_HOME/rapid/; ant jar
```

A.5.4 Generating Trace Logs.
Option-1. Readers interested in simply reproducing the results can download the traces used in our experiments from [1] and jump to Appendix A.5.5 directly. Next, replace the `benchmarks/` folder:

```
> rm -rf $AE_HOME/benchmarks/
> unzip /path/to/downloaded/zip -d $AE_HOME/
> mv $AE_HOME/asplos20-ae-traces $AE_HOME/benchmarks/
```

Option-2.

1. **Download and install Roadrunner**

```
> cd $AE_HOME
> git clone git@github.com:stephenfreund\
/RoadRunner.git
```

```
> wget https://raw.githubusercontent.com/umangm/\
rapid/master/notes/PrintSubsetTool.java.txt \
-O $AE_HOME/RoadRunner/src/rr/simple/\
PrintSubsetTool.java
> cd $AE_HOME/RoadRunner; ant; source msetup
```

2. **Download and install Rapid**

```
> cd $AE_HOME
> git clone git@github.com:umangm/rapid.git
> cd $AE_HOME/rapid; ant jar
```

3. **Move to `scripts/` folder.** We will now execute some scripts and for this, we will change the working directory:

```
> cd $AE_HOME/scripts/
```

4. **Extract execution logs.** To generate full trace for a single benchmark:

```
> python gen_trace.py <b>
```

Here, <b> could be something like `philo`. To generate traces for all benchmarks:

```
> python gen_trace.py
```

This step generates files `full_trace.rr` in the directory `$AE_HOME/benchmarks/<b>/`, either for particular benchmark or for all benchmarks based on the command.

5. **Atomicity specifications.** To modify the trace to account for the atomicity specifications for a single benchmark:

```
> python atom_spec.py <b>
```

To account for the atomicity specifications for all benchmarks:

```
> python atom_spec.py
```

This step generates files `$AE_HOME/benchmarks/<b>/trace.std` (either for particular benchmark or for all benchmarks). At this point, the files `full_trace.rr` can be deleted.

A.5.5 Performing Analyses.

Our experiments perform 3 different analysis on the traces: (a) metadata analysis to collect information about the different kinds of events in traces, (b) Velodrome analysis, and (c) AeroDrome analysis.

Obtaining Trace metadata. To generate metadata information from the trace of a single benchmark <b>:

```
> python metainfo.py <b>
```

When the files `trace.std` are available for all benchmarks, run:

```
> python metainfo.py
```

This step generates three files in `$AE_HOME/benchmarks/<b>/`: (i) the file `metainfo.err` contains error information from the Java commands run in the python script `metainfo.py` and should ideally be empty, (ii) `metainfo.txt` contains the actual output (including the number of different kinds of events); refer to `$AE_HOME/README.md` for a description of the contents of this file, (iii) `metainfo.tim` reports the time taken by the system.

AeroDrome analysis. For a single benchmark <b>, run:

```
> python aerodrome.py <b>
```

To analyze the traces for all benchmarks, run:

```
> python aerodrome.py
```

This step generates three files in `$AE_HOME/benchmarks/<b>/` - `aerodrome.txt`, `aerodrome.err` and `aerodrome.txt`. Their description can be found in `$AE_HOME/README.md`.

Velodrome analysis For a single benchmark <b>, run:

```
> python velodrome.py <b>
```

To analyze the traces of all benchmarks, run:

```
> python velodrome.py
```

As before this step generates files `velodrome.txt`, `velodrome.err` and `velodrome.tim`, whose description can be found in the readme file `$AE_HOME/README.md`.

A.6 Evaluation and expected result

The workflow described in Appendix A.5 can be used to generate the data showed in Table 1 and Table 2. The primary objective of the evaluation is to measure the speedup of AeroDrome analysis over Velodrome analysis. We expect that Aerodrome outperforms Velodrome on all benchmarks where the speedup of AeroDrome (over Velodrome) is more than 10×. The exact speed-ups may vary depending upon the hardware and other processes running, but orders of magnitude (for speedup) should stay the same. Of course, results can vary when the the traces used are different from those used in our experiments [1]. The metadata analysis described in Appendix A.5 can be used to generate the total number of events, threads, locks and memory locations (often referred to as variables).

A.7 Experiment customization

All the different analysis described in the workflow (Appendix A.5) can be performed for an execution of any concurrent Java program. For this, see the instructions[4] in Rapid [41, 42] to generate a trace from a benchmark. After this, if an atomicity specification is available, one can account for it by using the script `atom_spec.py`. If not, simply use an empty file for an atomicity specification and use the same script. Finally, all the three analyses can be run using scripts provided (`metainfo.py`, `aerodrome.py` and `velodrome.py`).

A.8 Notes

Contact umathur3@illinois.edu regarding any questions.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

[4] `$AE_HOME/rapid/notes/Generate_RoadRunner_traces.md`