Fast Atomicity Monitoring

HÜNKAR CAN TUNÇ*, Uber Technologies, Netherlands

YIFAN DONG*, Aarhus University, Denmark

ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

Atomicity is a fundamental abstraction in concurrency, specifying that program behavior can be understood by considering specific code blocks executing atomically. However, atomicity invariants are tricky to maintain while also optimizing for code efficiency, and atomicity violations are a common root cause of many concurrency bugs. To address this problem, several dynamic techniques have been developed for testing whether a program execution adheres to an atomicity specification, most often instantiated as *conflict-serializability*. The efficiency of the analysis has been targeted in various papers, with the state-of-the-art algorithms `RegionTrack` and `Aerodrome` achieving a time complexity $O(nk^3)$ and $O(nk(k+v+\ell))$, respectively, for a trace σ of n events, k threads, v locations, and ℓ locks.

In this paper we introduce `AtomSanitizer`, a new algorithm for testing conflict-serializability, with time complexity $O(nk^2)$. `AtomSanitizer` operates in an efficient streaming style, is theoretically faster than all existing algorithms, and also has a smaller memory footprint. Moreover, it is the first algorithm designed to use little locking when deployed in a concurrent monitoring setting. Experiments on standard benchmarks indicate that `AtomSanitizer` is always faster in practice than all existing conflict-serializability testers. Finally, we also implement `AtomSanitizer` inside the `TSAN` framework, for monitoring atomicity in real time. Our experiments reveal that `AtomSanitizer` incurs only a marginal time and space overhead over the data-race detection engine of `TSAN`, and thus is the first algorithm for conflict-serializability demonstrated to be suitable for a runtime monitoring setting.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Software testing and debugging**.

Additional Key Words and Phrases: concurrency, transactions, conflict-serializability, dynamic runtime analysis

ACM Reference Format:

Hünkar Can Tunç, Yifan Dong, and Andreas Pavlogiannis. 2026. Fast Atomicity Monitoring. *Proc. ACM Program. Lang.* 10, PLDI, Article 170 (June 2026), 24 pages. https://doi.org/10.1145/3808248

1 Introduction

Writing correct concurrent programs is a notoriously challenging task, as interprocess communication is inherently non-deterministic, encompassing unanticipated scheduling patterns that often

*Both authors contributed equally to this work.

Authors' Contact Information: Hünkar Can Tunç, Uber Technologies, Amsterdam, Netherlands, tunc@uber.com; Yifan Dong, Aarhus University, Aarhus, Denmark, yvan.dong@cs.au.dk; Andreas Pavlogiannis, Aarhus University, Aarhus, Denmark, pavlogiannis@cs.au.dk.

ACM 2475-1421/2026/6-ART170
https://doi.org/10.1145/3808248

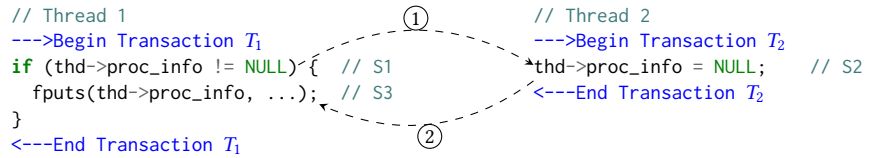


Fig. 1. An atomicity violation bug in MySQL 4.1.2 release, taken from [26].

escape the programmers' mental model. For the same reason, concurrency bugs are difficult to reproduce during debugging, and are commonly referred to as *heisenbugs* [34].

One fundamental building block in the mental model of programmers is *atomicity*, where the whole program consists of individual blocks that can be considered to execute atomically. Although this is not actually the case due to the scheduler, atomicity allows one to reason about programs in terms of atomic blocks, meaning that their interleaving cannot produce new behaviors. However, violations of atomicity are common in practice and are the source of many real-world concurrency bugs [26, 27, 37]. For example, Figure 1 depicts an atomicity violation in MySQL [18], taken from [26]. Two threads access concurrently the shared variable `thd->proc_info`. The developers assumed that `S1` and `S3` always execute atomically in thread 1 due to synchronization by code not shown in this snippet, thus `thd->proc_info` in `S3` will never be `NULL`. However, `S2` of thread 2 can, in fact, be interleaved with `S1` and `S3`, breaking this atomicity invariant and leading to a bug that has resurfaced several times [3, 10]. We remark that atomicity violations are different from data races; they occur when concurrent accesses from different threads interleave in an unexpected way, even, e.g., if such accesses are synchronizing, and can thus occur even on race-free programs.

The programming languages community has responded with techniques for detecting atomicity violations automatically. One standard approach is via dynamic analyses that report warnings based on violations of *conflict-serializability*. The analysis observes executions σ of the concurrent program which are annotated with intended atomic blocks, called transactions. Annotations are external to the analysis —they might come explicitly from the programmer [17], or be inferred automatically [27]. Although transactions may interleave in σ, the analysis must decide whether there is an equivalent execution σ', free of transaction interleavings, obtainable from σ by successive swaps of non-conflicting events. If not, σ witnesses an atomicity violation. The size of σ is usually huge, which puts pressure on the analysis to be as fast as possible. In the following, consider that σ contains n events, k threads, v locations, and ℓ locks.

In high level, conflict-serializability violations can be detected by creating a conflict graph of all transactions, whereby two transactions are ordered $T_1 \longrightarrow T_2$ iff each T_i contains an event e_i such that e_1 and e_2 conflict (both access the same shared resource and at least one modifies it) and e_1 executes before e_2 in σ. Then, a violation corresponds to a cycle $T_1 \longrightarrow \cdots \longrightarrow T_m$, representing the fact that these m transactions cannot execute serially without breaking the conflict order of the events in σ (see, e.g., the cycle $T_1 \longrightarrow T_2 \longrightarrow T_1$ in Figure 1). Constructing the conflict graph and checking for cycles can be easily performed in $O(n)$ time, but this approach has two notable disadvantages. First, the space usage is $\Theta(n)$, which can be prohibitive in practice. Second, practical applications pose an online setting, whereby σ is processed in a streaming style, and a violation is reported as early as it occurs. Consequently, there have been several efforts in designing atomicity-checking algorithms that operate in this online fashion and are as efficient as possible (see Table 1).

One of the first techniques for dynamic atomicity testing was developed in [2]. That algorithm, however, employed aggressive pruning of its data structures in order to keep its memory footprint small, which lead to incorrect results. This was reported in [12], which proposed an automata-based algorithm (which we refer to as `FM`) with running time $O(nk(v + \ell))$. At the same time,

Table 1. The complexity of atomicity algorithms on executions of n events, k threads, v locations, and ℓ locks. Some works give more fine-grained complexity bounds, e.g., involving the number of transactions in the trace.

Algorithm	Time	Space
FM [12]	$O(nk(v+\ell))$	$O(k(k+v+\ell))$
Velodrome [16]	$O(n^2)$	$O(n)$
DoubleChecker [5]	$O(n^2)$	$O(n)$
Aerodrome [32]	$O(nk(k+v+\ell))$	$O(k(k+v+\ell))$
RegionTrack [28]	$O(nk^3)$	$O(k(kv+\ell))$
AtomSanitizer [This Paper]	$O(nk^2)$	$O(k(k+v)+\ell)$

the graph-based algorithm Velodrome was developed [16], and has become a standard reference in atomicity testing. Velodrome maintains explicitly a conflict graph that can grow as large as $\Theta(n)$, with cycles representing atomicity violations. It has $O(n^2)$ time and $O(n)$ space complexity, which are prohibitive in theory, but is made performant in practice via graph-pruning heuristics. DoubleChecker [5] further improved of Velodrome by observing that precise cycle detection is required infrequently. Aerodrome was recently proposed as the first algorithm utilizing vector clocks as its main data structure [32], which are efficient and common in dynamic concurrency analyses. Aerodrome has time and space complexity $O(nk(k+v+\ell))$ and $O(k(k+v+\ell))$, respectively. The latest development in this sequence is RegionTrack, which also utilizes vector clocks, towards time and space complexity $O(nk^3)$ and $O(k(kv+\ell))$, respectively [28]. The running time of RegionTrack is incomparable to Aerodrome in general, but it becomes faster for reasonable input parameters (i.e., typically $k << v$). However, RegionTrack has increased space usage by a term vk^2. Finally, when deployed in a runtime monitoring setting, where various threads access the components of the algorithm concurrently, all existing testers require each thread to acquire a global lock. This incurs congestion and leads to noticeable monitoring overhead, beyond what is considered acceptable in an industrial scale (e.g., TSAN's reported 5×-15× time and 5×-10× memory overhead [42]).

The long study of atomicity checking, as witnessed by the above sequence of improvements, highlights the need for methods that are as performant as possible. *Is there an atomicity checker with even better complexity? Does it also lead to practical improvements? Is atomicity monitorable at overheads acceptable by industrial practice? Is the complexity increase of online monitoring necessary, compared to the $O(n)$ cost for offline detection?* This paper answers these questions in positive, by making the following contributions.

1. AtomSanitizer for conflict-serializability. We develop AtomSanitizer, a new algorithm for testing conflict-serializability. For a trace σ of n events, k threads, v locations, and ℓ locks, AtomSanitizer uses $O(nk^2)$ time and $O(k(k+v)+\ell)$ space. This is better than all existing algorithms in both time and space (see Table 1), for reasonable input parameters. In particular, AtomSanitizer is faster when $k << v$, while its space complexity is smaller by a term $k\ell$.

We further remark that the space complexity of AtomSanitizer is also bounded by $O(n+k^2)$, but the aforementioned space bound is better as long as $kv < n$, which is typically the case. Moreover, the term kv can be refined to $\sum_x k_x$, where k_x is the number of threads reading on variable x. Although $\sum_x k_x = O(kv)$ in the worst-case, in practice most variables are accessed by only a few threads, reducing the space complexity to $O(k^2+v+\ell)$. In contrast, existing algorithms do not enjoy this property. This makes AtomSanitizer the first algorithm with a memory footprint small enough to be employed in a runtime setting.

2. AtomSanitizer for increasing-cycle violations. We present a version of AtomSanitizer focused on increasing-cycle violations, which are specific conflict-serializability violations in

which the root cause can be localized to a single offending transaction, and has been utilized in prior work [5, 16, 28]. `AtomSanitizer` detects increasing-cycle violations in time $O(nk)$ and space $O(k(k+v)+\ell)$. Although this does not improve the time complexity of the state-of-the-art (in particular `RegionTrack` [28]), it improves the space complexity by a term $k\ell$, and in practice achieves a memory footprint of $O(k^2+v+\ell)$, by a similar analysis as above.

3. Lower bounds. We complement our improved upper bounds with two interesting lower bounds that characterize fundamental computational resources required to detect conflict-serializability violations in an streaming fashion. First, recall that, if we relinquish the practical requirement that a violation is reported in an online fashion as soon as it occurs (or shortly after, e.g., when the latest offending transaction completes), the problem can be trivially solved in $O(n)$ time. *Is it possible to achieve linear time also in the online setting?* We prove that this is unlikely, in the sense that the online problem must take $\Omega(n^{3/2-\epsilon})$ time, for any fixed $\epsilon > 0$, under the popular Online Matrix-Vector hypothesis [21]. Second, note that the space usage of `AtomSanitizer` is practically sublinear in n, but can become linear in n when there are $v = \Omega(n)$ locations. *Is a sublinear space bound always possible?* We prove that this is not the case, in the sense that any streaming algorithm for conflict-serializability (or even increasing-path) violations must use $\Omega(n)$ space when $v = \Omega(n)$, in the worst case.

4. Implementation and experiments. We implement `AtomSanitizer` inside the `RAPID` framework of dynamic analyses [29]. Our tool processes a trace σ and reports the first transaction which completes a cycle in the underlying conflict graph, as per standard. We compare the performance of our tool to existing algorithms in the literature, on standard benchmarks. `AtomSanitizer` is faster on each benchmark, achieving an average speedup of 2.0×, 4.5×, and 12× over `RegionTrack`, `Aerodrome`, and `Velodrome`, respectively. Moreover, `AtomSanitizer` is always faster than `RegionTrack` in detecting increasing-cycle violations (this is the only baseline that makes sound and complete reports of increasing-cycles), with an average speedup of 1.7×.

We also implement `AtomSanitizer` inside the widely used LLVM `TSAN` framework [42] for runtime analysis. The core data structure of `AtomSanitizer` is designed to use little locking in a runtime setting, and our implementation exploits this feature to avoid expensive synchronization between threads most of the time. We perform scalability experiments on a standard set of benchmarks, and our results show that `AtomSanitizer` incurs only a marginal slowdown and memory cost over the data-race detector engine of `TSAN` (1.49× and 1.06×, respectively, on average). Relative to the original execution, the average time and memory overhead is 4.63× and 3.83×, respectively, which generally aligns with industrial expectations [42]. To our knowledge, this is the first case of an atomicity tester that achieves such a small overhead, making it now possible to monitor atomicity at overheads that are industry standard.

2 Preliminaries

We begin with general notation on concurrent executions and the atomicity notions we consider.

2.1 Concurrent Execution Model

Events and traces. A trace is a sequence of events $\sigma = e_1, \ldots, e_n$, encapsulating the execution of a concurrent program. Given an event e of σ, we write $\sigma[:e]$ to denote the prefix of σ up to (including) e. In turn, an event is a tuple $e = \langle t, i, op \rangle$, where t is an identifier of the thread executing e, i is an incremental id for thread t (i.e. it denotes the i-th event of t), and op is the operation performed by e, which is one of the following.

1. $\texttt{begin}$, denoting that e is the beginning of a transaction.
2. $\texttt{end}$, denoting that e is the end of a transaction.
3. $\texttt{r}(x)$, denoting that e reads global variable x.
4. $\texttt{w}(x)$, denoting that e writes to global variable x.
5. $\texttt{acq}(\ell)$, denoting that e acquires the lock ℓ.
6. $\texttt{rel}(\ell)$, denoting that e releases the lock ℓ.

We often denote the $\texttt{begin}$ and $\texttt{end}$ events as $\triangleright$ and $\triangleleft$, respectively. We use $\texttt{tid}(e)$, $\texttt{op}(e)$, and $\texttt{id}(e)$, to denote the thread identifier, the operation, and identifier of e, respectively. If e accesses a variable v or a lock ℓ, we let $\texttt{var}(e)$ denote v or ℓ, respectively. We often denote an event e as $\langle t, op \rangle$, omitting its identifier. Note that we ignore the values read/written, as these are not important in our setting. We also don't explicitly consider fork/join events, though these can be handled naturally.

Traces have to be well-formed: each $\texttt{acq}(\ell)$ is paired with a corresponding $\texttt{rel}(\ell)$ in the same thread, and no lock is held by more than one thread simultaneously. Moreover, for any thread t, the sequence of $\texttt{begin}$ and $\texttt{end}$ events starts with $\texttt{begin}$, alternates between $\texttt{begin}$ and $\texttt{end}$, and ends on $\texttt{end}$[1]. We denote by $\texttt{Events}_\sigma$ the set of events, $\texttt{Variables}_\sigma$ the global variables, $\texttt{Locks}_\sigma$ the set of locks, $\texttt{Reads}_\sigma$ the set of read events, and $\texttt{Writes}_\sigma$ the set of write events in σ.

Transactions. Given a trace σ, a transaction T is a sequence of events in a thread t that begins with $\langle t, \triangleright \rangle$ and ends with the matching $\langle t, \triangleleft \rangle$, at which point T is completed. We use $T.\texttt{begin}$ to denote the begin event of T. Transaction T is assigned a unique, incremental id in thread t, accessed as $\texttt{id}(T)$. T is *unary* if it contains a single event besides $\triangleright$ and $\triangleleft$. T is *open* (or *active*) in a prefix $\sigma[:e]$ if it has executed its begin but not its end event in $\sigma[:e]$. We write $e \in T$ if e appears in T. We let $\texttt{txn}(e)$ be the transaction that e appears in. As per standard, we assume that every event belongs to a (possibly unary) transaction.

Conflicting events. Two events e_1 and e_2 *conflict*, denoted by $e_1 \asymp e_2$, if (i) $\texttt{tid}(e_1) = \texttt{tid}(e_2)$, or (ii) $\texttt{op}(e_1), \texttt{op}(e_2) \in \{\texttt{acq}(\ell), \texttt{rel}(\ell)\}$ for some lock ℓ, or (iii) $\texttt{op}(e_1), \texttt{op}(e_2) \in \{\texttt{w}(x), \texttt{r}(x)\}$ for some variable x and at least one writes to x.

Partial orders. We make frequent use of partial orders over events $\texttt{Events}_\sigma$, or subsets thereof. A (strict) partial order P, denoted $<^{\texttt{P}}_\sigma$ is a transitive, antisymmetric and irreflexive relation over $\texttt{Events}_\sigma$. We let $\leq^{\texttt{P}}_\sigma$ be the reflexive closure of P. Given two events e_1 and e_2, we say that e_1 is a $<^{\texttt{P}}_\sigma$*-immediate predecessor* of e_2 if (i) $e_1 <^{\texttt{P}}_\sigma e_2$ and (ii) there does not exist an event e such that $e_1 <^{\texttt{P}}_\sigma e <^{\texttt{P}}_\sigma e_2$. Note that $<^{\texttt{P}}_\sigma$-immediate predecessors are not unique in general, unless P is total.

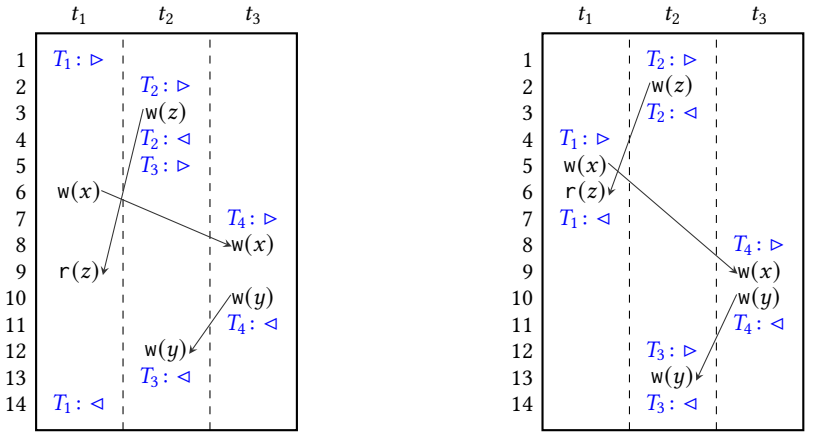
The trace σ defines a total order $<^{\texttt{TR}}_\sigma$ over $\texttt{Events}_\sigma$, with $e_1 <^{\texttt{TR}}_\sigma e_2$ iff e_1 occurs before e_2 in σ. The *thread order* $<^{\texttt{TO}}_\sigma$ is the smallest partial order such that $e_1 <^{\texttt{TO}}_\sigma e_2$ if $\texttt{tid}(e_1) = \texttt{tid}(e_2)$ and $e_1 <^{\texttt{TR}}_\sigma e_2$.

2.2 Atomicity as Serializability

Serial traces. As a natural consequence of concurrency, different transactions may overlap in a trace σ. We call σ *serial* if there is no such overlap, i.e., for any two transaction begin events e_1, e_2 such that $e_1 <^{\texttt{TR}}_\sigma e_2$, we have $e_3 <^{\texttt{TR}}_\sigma e_2$, where e_3 is the matching end event of e_1. *Serializability* captures the fact that, although σ might not be serial, its behavior is equivalent to a serial trace σ' that can be obtained by permuting adjacent non-conflicting events in σ. If such a transformation exists, σ is deemed *serializable*. Otherwise, σ witnesses an atomicity violation. In the following, we make these concepts formal.

[1]Wlog, we do not consider nested critical sections/transactions. These are handled straightforwardly in our implementation.

Conflict-serializability. For a trace σ, *conflict-happens-before* is the smallest partial order over Events_σ such that, for any two events e_1, e_2, if $e_1 \asymp e_2$ and $e_1 <_\sigma^{\mathsf{TR}} e_2$, then $e_1 <_\sigma^{\mathsf{CHB}} e_2$. We say that σ is *conflict-equivalent* to another trace σ' iff (i) $\text{Events}_\sigma = \text{Events}_{\sigma'}$, and (ii) $<_\sigma^{\mathsf{CHB}} = <_{\sigma'}^{\mathsf{CHB}}$. We say that σ is *conflict-serializable* if there exists a serial trace σ' such that σ conflict-equivalent to σ'. Otherwise, σ exhibits a *conflict-serializability* violation.



(a) σ_1 is conflict-serializable. (b) σ_2 is a serial permutation of σ_1

Fig. 2. A non-serial trace that is conflict-serializable.

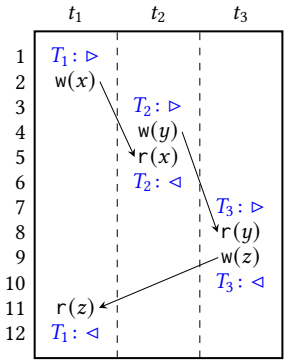
Examples. Consider the trace σ_1 in Figure 2a. It consists of 14 events executed by three threads t_1, t_2 and t_3. It has four transactions $T_1 = \triangleright, e_6, e_9, \triangleleft$, $T_2 = \triangleright, e_3, \triangleleft$, $T_3 = \triangleright, e_{12}, \triangleleft$ and $T_4 = \triangleright, e_8, e_{10}, \triangleleft$. We use an arrow $\longrightarrow$ to represent $<_\sigma^{\mathsf{CHB}}$ orderings across threads. The trace σ_1 is conflict-serializable, since we can construct a serial trace σ_2, shown in Figure 2b, that is conflict-equivalent to σ_1. In contrast, the trace σ_3 in Figure 3a is not conflict-serializable, since making the transactions atomic would require reversing the order of at least one of the $<_\sigma^{\mathsf{CHB}}$ arrows.

Increasing-cycle violations. When σ witnesses an atomicity violation, it is not clear which transactions to blame, and fixing it requires looking closely into the trace and possibly the program source. *Increasing-cycle violations* constitute a simple yet common class of atomicity violations on which there is a single candidate transaction to be blamed. In particular, σ exhibits an increasing-cycle violation on a transaction T if σ contains three distinct events e_1, e_2, and e_3 such that (i) e_1 and e_2 are the `begin` and `end` events of T, respectively, (ii) $e_3 \notin T$, and (iii) $e_1 <_\sigma^{\mathsf{CHB}} e_3 <_\sigma^{\mathsf{CHB}} e_2$.

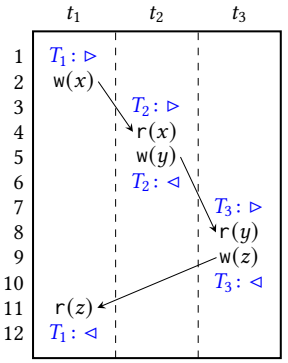
Examples. The trace σ_4 in Figure 3b witnesses an increasing-cycle violation on T_1, since (i) e_1 and e_{12} are the `begin` and `end` events of T_1, (ii) $e_9 \notin T_1$, and (iii) $e_1 <_{\sigma_4}^{\mathsf{CHB}} e_9 <_{\sigma_4}^{\mathsf{CHB}} e_{12}$. In contrast, σ_3 does not witness an increasing-cycle violation, although it is also not conflict-serializable.

3 Detecting Atomicity Violations

In this section we present `AtomSanitizer` for detecting conflict-serializability violations. We start with some basic lemmas that capture serializability (Section 3.1), followed by the presentation of `AtomSanitizer` (Section 3.2), and its correctness and complexity (Section 3.3). We conclude with some lower bounds on the time and space complexity of monitoring atomicity (Section 3.4).



(a) σ_3 witnesses a conflict-serializability violation.



(b) σ_4 witnesses an increasing-cycle violation.

Fig. 3. Two non-conflict-serializable traces.

3.1 Serializability Conditions

Transaction-happens-before. Given a trace σ, the *transaction-happens-before* relation $<_\sigma^{\mathsf{THB}}$ is the smallest transitive relation over the transactions of σ with the property that, for any two distinct transactions T_1 and T_2, if there exist $e_1 \in T_1$ and $e_2 \in T_2$ such that $e_1 <_\sigma^{\mathsf{CHB}} e_2$, then $T_1 <_\sigma^{\mathsf{THB}} T_2$. Conceptually, if $<_\sigma^{\mathsf{THB}}$ is a partial order, we can arrive at a serial trace σ' witnessing the serializability of σ by linearizing $<_\sigma^{\mathsf{THB}}$. This is formally captured in the following lemma.

Lemma 3.1 ([16]). *A trace σ is conflict-serializable iff $<_\sigma^{\mathsf{THB}}$ is irreflexive.*

For example, the trace σ_3 (Figure 3a) yields a cyclic $<_{\sigma_3}^{\mathsf{THB}}$ due to the orderings $T_1 \to T_2 \to T_3 \to T_1$. It is easy to see that $<_\sigma^{\mathsf{THB}}$ satisfies the following monotonicity property.

Remark 1. *For any event e of σ, we have $<_{\sigma[:e]}^{\mathsf{THB}} \subseteq <_\sigma^{\mathsf{THB}}$.*

Monotonicity implies that there is a smallest prefix of σ when a conflict-serializability violation occurs (i.e., $<_{\sigma[:e]}^{\mathsf{THB}}$ becomes cyclic). Different atomicity testers employ different techniques for detecting cycles in $<_\sigma^{\mathsf{THB}}$ by performing a single pass (i.e., streaming-style) of σ and checking whether $<_\sigma^{\mathsf{THB}}$ is irreflexive at each step. `AtomSanitizer` is based on the notions of latest remote and earliest local transactions, that we develop below.

Latest remote transactions and earliest local transactions. For a trace σ and two threads t_1, t_2, we define the *latest remote transaction* of t_2 known to t_1, $\mathrm{LRT}_\sigma^{t_1}(t_2)$, as follows.

- If t_1 is not executed in σ, then $\mathrm{LRT}_\sigma^{t_1}(t_2) = \bot$. Otherwise, let T_1 be the latest transaction of t_1 in σ.
- If t_2 has executed a transaction T_2 such that $T_2 <_\sigma^{\mathsf{THB}} T_1$, then $\mathrm{LRT}_\sigma^{t_1}(t_2)$ is the latest such transaction T_2, otherwise $\mathrm{LRT}_\sigma^{t_1}(t_2) = \bot$.

If $\mathrm{LRT}_\sigma^{t_1}(t_2) \neq \bot$, we define the *earliest local transaction* in t_1 for t_2 as $\mathrm{ELT}_\sigma^{t_1}(t_2) = T_1'$, where T_1' is the earliest transaction in t_1 such that $\mathrm{LRT}_\sigma^{t_1}(t_2) <_\sigma^{\mathsf{THB}} T_1'$. Note that such a transaction must exist, being either T_1 or some earlier transaction. On the other hand, if $\mathrm{LRT}_\sigma^{t_1}(t_2) = \bot$, we simply let $\mathrm{ELT}_\sigma^{t_1}(t_2) = \bot$. In words, $\mathrm{LRT}_\sigma^{t_1}(t_2)$ is the latest transaction of thread t_2 that t_1 is aware of, while $\mathrm{ELT}_\sigma^{t_1}(t_2)$ is the earliest transaction of t_1 that was made aware of that transaction of t_2.

Example. Consider the trace σ_5 in Figure 4. At the event e_6, we have $\mathrm{LRT}_{\sigma_5[:e_6]}^{t_3}(t_1) = T_1$ and $\mathrm{ELT}_{\sigma_5[:e_6]}^{t_3}(t_1) = T_2$. At the event e_{13}, we have $\mathrm{LRT}_{\sigma_5[:e_{13}]}^{t_3}(t_2) = T_3$ and $\mathrm{ELT}_{\sigma_5[:e_{13}]}^{t_3}(t_2) = T_5$. Finally,

at the end of σ_5, we have $\mathrm{LRT}^{t_3}_{\sigma_5}(t_2) = T_4$ and $\mathrm{ELT}^{t_3}_{\sigma_5}(t_2) = T_6$, as well as $\mathrm{LRT}^{t_3}_{\sigma_5}(t_1) = T_1$ and $\mathrm{ELT}^{t_3}_{\sigma_5}(t_1) = T_2$.

The operating principle of `AtomSanitizer` is based on the following lemma, which characterizes atomicity violations in terms of $<^{\mathsf{CHB}}_{\sigma}$, $\mathrm{LRT}^{t_1}_{\sigma}(t_2)$, and $\mathrm{ELT}^{t_1}_{\sigma}(t_2)$.

Lemma 3.2. *A trace σ is not conflict-serializable iff there exist two distinct events e' and e'' such that the following conditions hold, where $t' = \mathrm{tid}(e')$ and $t'' = \mathrm{tid}(e'')$: (i) $t' \neq t''$ and e' is the $<^{\mathsf{CHB}}_{\sigma}$-immediate predecessor of e'', (ii) $\mathrm{LRT}^{t'}_{\sigma[:e]}(t'') = \mathtt{txn}(e'')$, and (iii) $\mathrm{ELT}^{t'}_{\sigma[:e]}(t'').\mathtt{begin} \leq^{\mathsf{TO}}_{\sigma} \mathtt{txn}(e').\mathtt{begin}$, where e is a $<^{\mathsf{TR}}_{\sigma}$-immediate predecessor of e''.*

Intuitively, condition (i) captures that $\mathtt{txn}(e') <^{\mathsf{THB}}_{\sigma} \mathtt{txn}(e'')$ by the definition of $<^{\mathsf{THB}}_{\sigma}$, while conditions (ii) and (iii) imply that $\mathtt{txn}(e'') <^{\mathsf{THB}}_{\sigma} \mathtt{txn}(e')$. This is because (ii) states that t' has learned the transaction $\mathtt{txn}(e'')$, and (iii) states that there exists a transaction T' in t' such that $\mathtt{txn}(e'') <^{\mathsf{THB}}_{\sigma} T' \leq^{\mathsf{THB}}_{\sigma} \mathtt{txn}(e')$. Thus, we have a violation as per Lemma 3.1, since $<^{\mathsf{THB}}_{\sigma}$ is not irreflexive.

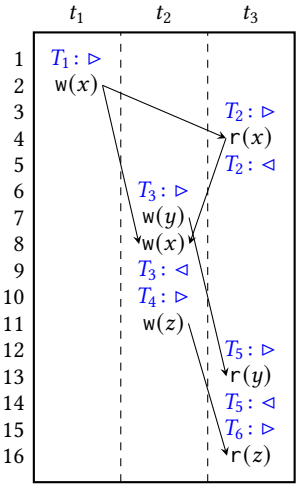


Fig. 4. A trace σ_5.

Example. Consider the trace σ_3 in Figure 3a. At the event e_{11}, Lemma 3.2, reports a conflict-serializability violation. The events e_9 and e_{11} correspond to e' and e'' in Lemma 3.2, respectively. Here we have that (i) e_9 is a $<^{\mathsf{CHB}}_{\sigma_3}$-immediate predecessor of e_{11}, (ii) $\mathrm{LRT}^{t_3}_{\sigma_3[:e_{10}]}(t_1) = T_1$, and (iii) $\mathrm{ELT}^{t_3}_{\sigma_3[:e_{10}]}(t_1).\mathtt{begin} \leq^{\mathsf{TO}}_{\sigma_3} T_3.\mathtt{begin}$. Now consider the trace σ_1 in Figure 2a. At the event e_{12}, we have that e_{10} is a $<^{\mathsf{CHB}}_{\sigma_1}$-immediate predecessor of e_{12}, but $\mathrm{LRT}^{t_3}_{\sigma_1[:e_{11}]}(t_2) = T_2 \neq \mathtt{txn}(e_{12})$, thus Lemma 3.2 does not report a conflict-serializability violation.

3.2 The AtomSanitizer Algorithm

We now present `AtomSanitizer` for detecting conflict-serializability violations based on Lemma 3.2.

Intuition. `AtomSanitizer` performs a single pass on the input trace σ, and produces a warning after processing the smallest prefix of σ that contains a conflict-serializability violation (recall Remark 1). Similarly to other conflict-serializability algorithms, after processing an event e, it maintains the maximal events of $<^{\mathsf{CHB}}_{\sigma[:e]}$, in order to derive corresponding $<^{\mathsf{THB}}_{\sigma[:e]}$ orderings. Its primary (and distinctive from other works) data structure consists of two vector clocks From_t and To_t, for each thread t, of size k each. After processing an event e, for each thread t_2, $\mathsf{From}_{t_1}(t_2)$ and $\mathsf{To}_{t_1}(t_2)$ store identifiers of a transaction T_2 of t_2 and a transaction T_1 of t_1, respectively, such that $T_2 <^{\mathsf{THB}}_{\sigma[:e]} T_1$. As a first attempt, in order to report violations based on Lemma 3.2, it is natural to try and maintain at all events e the following invariant

$$\mathsf{From}_{t_1}(t_2) = \mathtt{id}(\mathrm{LRT}^{t_1}_{\sigma[:e]}(t_2)) \qquad \text{and} \qquad \mathsf{To}_{t_1}(t_2) = \mathtt{id}(\mathrm{ELT}^{t_1}_{\sigma[:e]}(t_2)) \tag{1}$$

as, together with the $<^{\mathsf{CHB}}_{\sigma[:e]}$-maximal events, it allows to test for the conditions of Lemma 3.2.

Unfortunately, Eq. (1) appears difficult to maintain efficiently *at all events e*. This is because, when processing an event e of thread t_1, a transaction-happens-before path might be formed from

$\mathrm{LRT}^{t_1}_{\sigma}(t_2)$ to t_1 via some intermediate thread t_3. If the transaction $\mathrm{LRT}^{t_1}_{\sigma}(t_2)$ is not open, t_3 might be aware of a later transaction of t_2, and the information about $\mathrm{LRT}^{t_1}_{\sigma}(t_2)$ is lost (see Figure 5).

In order to overcome the above obstacle and regain the invariant of Eq. (1), it is natural to attempt to store some additional information in the state of the algorithm, perhaps at the cost of impacting its running time. One key idea behind `AtomSanitizer` is that this invariant is, in fact, *not necessary*. In particular, `AtomSanitizer` guarantees that Eq. (1) holds *if* $\mathrm{LRT}^{t_1}_{\sigma[:e]}(t_2)$ *is open in* $\sigma[:e]$. This is sufficient since, in Lemma 3.2, e'' is the currently processed event of t'' and thus $\mathrm{LRT}^{t'}_{\sigma[:e]}(t'') = \mathtt{txn}(e'')$ must be open in $\sigma[:e]$. Moreover, this relaxation makes the From_{t_1} and To_{t_1} clocks easy to maintain inductively, in terms of the corresponding clocks of the other threads.

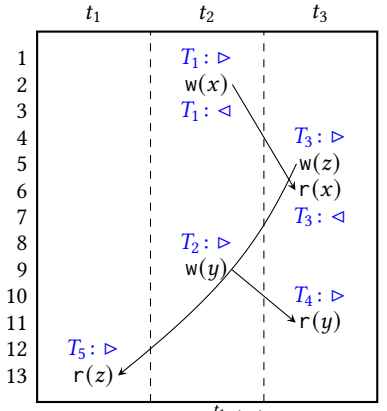


Fig. 5. A trace σ_6. $\mathrm{LRT}^{t_1}_{\sigma_6}(t_2) = T_1$ via T_3 of t_3, but this fact is not recoverable from $\mathrm{LRT}^{t_3}_{\sigma_6}(t_2)$, as it has progressed to T_2.

AtomSanitizer. We now describe `AtomSanitizer` in detail. The state of the algorithm is represented in the following components.

- For each thread t, we have the vector clocks From_t and To_t, as outlined above.
- For each location x, we have (i) a variable $\mathtt{lastWrite}_x$, storing the most recent write on x, (ii) a map $\mathtt{lastReads}_x$, storing the most recent read of each thread, and (iii) a set $\mathtt{lastReadTids}_x$, storing the threads that have performed a read on x after the most recent write on x.
- For each lock ℓ, we have a variable $\mathtt{lastRelease}_\ell$ storing the most recent release on ℓ.

`AtomSanitizer` is implemented in three parts.

- Algorithm 1 is the top level, defining a set of handlers for processing each event e according to its type. It keeps track of (a superset of) the maximal events of $<^{\mathsf{CHB}}_{\sigma[:e]}$, and performs checks on whether $\sigma[:e]$ witnesses a conflict-serializability violation, by calling Algorithm 2.
- Algorithm 2 handles the conflict-serializability checks, based on the clocks $\mathsf{From}_{t_1}(t_2)$ and $\mathsf{To}_{t_1}(t_2)$ maintained collectively (for all threads) in a data structure called `DS`. It also updates these vector clocks by performing the appropriate data structure call in Algorithm 3.
- Algorithm 3 implements the data structure `DS` for maintaining the clocks $\mathsf{From}_{t_1}(t_2)$ and $\mathsf{To}_{t_1}(t_2)$.

AtomSanitizer event handlers (Algorithm 1). We now describe each handler, referring to Algorithm 1 for details. We do not list handlers for transaction `begin` and `end` events, as these are only required to keep track of the id of each transaction, in a straightforward manner. We use w, r and rel to denote the write, read and release events here.

Function `read(e)`. When processing a read event $e = \mathtt{r}(x)$, it may case an ordering $\mathtt{w}(x) <^{\mathsf{CHB}}_{\sigma[:e]} e$, where $\mathtt{w}(x)$ is the latest write on x. The algorithm calls `checkAndUpdate` (Line 1) to check whether this ordering creates a cycle in $<^{\mathsf{THB}}_{\sigma[:e]}$, and to update `DS` accordingly.

Function `write(e)`. When processing a write event $e = \mathtt{w}(x)$, this can result in k orderings $\mathtt{r}(x) <^{\mathsf{CHB}}_{\sigma[:e]} e$, one for the last read $\mathtt{r}(x)$ of each thread t', as well as one additional ordering $\mathtt{w}'(x) <^{\mathsf{CHB}}_{\sigma[:e]} e$, where $\mathtt{w}'(x)$ is the latest write on x. For each of these events, the algorithm calls `checkAndUpdate` (Line 1 and 30), as in the case of `read(e)`.

Algorithm 1: `AtomSanitizer` event handlers.

```
1  function initialize()
2      foreach x ∈ Variables_σ do
3          lastWrite_x ← ⊥
4          lastReads_x ← ⊥
5          lastReadTids_x ← ∅
6      foreach ℓ ∈ Locks_σ do
7          lastRelease_ℓ ← ⊥
8  function release(e)
9      ℓ ← var(e)
10     lastRelease_ℓ ← e
11 function read(e)
12     ⟨x, t⟩ ← ⟨var(e), tid(e)⟩
13     w ← lastWrite_x
14     lastReads_x[t] ← e
15     lastReadTids_x ←
           lastReadTids_x ∪ {t}
16     checkAndUpdate(w, e)
17 function acquire(e)
18     ℓ ← var(e)
19     rel ← lastRelease_ℓ
20     checkAndUpdate(rel, e)
21 function write(e)
22     x ← var(e)
23     tids ← lastReadTids_x
24     foreach t ∈ tids do
25         r ← lastReads_x[t]
26         checkAndUpdate(r, e)
27     w ← lastWrite_x
28     lastWrite_x ← e
29     lastReadTids_x ← ∅
30     checkAndUpdate(w, e)
```

Function `acquire(e)`. When processing an acquire event $e = \mathtt{acq}(\ell)$, this can result in one ordering $\mathtt{rel}(\ell) <^{\mathsf{CHB}}_{\sigma[:e]} e$, where $\mathtt{rel}(\ell)$ is the latest release on ℓ. This is handled as previously (Line 1).

Function `release(e)`. When processing a release event $e = \mathtt{rel}(\ell)$, the relation $<^{\mathsf{CHB}}_{\sigma[:e]}$ is only updated trivially, since all events that are $<^{\mathsf{CHB}}_{\sigma[:e]}$-ordered before e are also $<^{\mathsf{CHB}}_{\sigma[:e']}$ before e', where e' is the $<^{\mathsf{TO}}_{\sigma}$-immediate predecessor of e. Therefore conflict-serializability checks can be avoided here, and the function only performs simple bookkeeping.

Algorithm 2: Checking conflict serializability.

```
1  function checkAndUpdate(v, e)
2      if tid(v) ≠ tid(e) then
3          n_1 ← ⟨tid(v), id(txn(v))⟩
4          n_2 ← ⟨tid(e), id(txn(e))⟩
5          if DS.reachable(n_2, n_1) then
6              declare Conflict serializability violation
7          insertEdge(n_1, n_2)
8  function insertEdge(⟨t_1, j_1⟩, ⟨t_2, j_2⟩)
9      foreach t'_1 ∈ Thr_σ do
10         foreach t'_2 ∈ Thr_σ \ {t'_1} do
11             if t'_1 = t_1 then j'_1 ← j_1
12             else j'_1 ← DS.predecessor(⟨t_1, j_1⟩, t'_1)
13             if t'_2 = t_2 then j'_2 ← j_2
14             else j'_2 ← DS.successor(⟨t_2, j_2⟩, t'_2)
15             DS.addSuccessor(⟨t'_1, j'_1⟩, ⟨t'_2, j'_2⟩)
```

Conflict-serializability checking (Algorithm 2). This part of the algorithm handles new $<^{\mathsf{CHB}}_{\sigma}$ orderings passed by the handlers of Algorithm 1, so as to check for violations and update the conflict graph.

Function `checkAndUpdate(v, e)`. When a handler of Algorithm 1 discovers a new ordering $v <^{\mathsf{CHB}}_{\sigma[:e]} e$, it invokes `checkAndUpdate(v, e)`. This function first queries `DS` to check whether Lemma 3.2 applies, yielding a conflict-serializability violation (Line 2). Otherwise, it proceeds to record the new edge $\mathtt{txn}(v) <^{\mathsf{THB}}_{\sigma[:e]} \mathtt{txn}(e)$ in `DS` (Line 2).

Function `insertEdge`$(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$. This function handles an edge insertion recording the ordering $\mathtt{txn}(v) <^{\mathsf{THB}}_{\sigma[:e]} \mathtt{txn}(e)$ in `DS`. Each transaction T is represented as a pair $\langle t, j\rangle$, where t is the thread of T and j is the id of the begin event of T. Since $<^{\mathsf{THB}}_{\sigma[:e]}$ is transitively closed, this function

inserts in DS all transitive orderings implied through this edge, which can be discovered by making appropriate calls to DS (Line 2 and Line 2).

Algorithm 3: Data structure.

```
1  function initialize()
2      foreach t ∈ Thr_σ do
3          foreach t' ∈ Thr_σ do
4              From_t(t') ← ⊥; To_t(t') ← ⊥
5  function reachable(⟨t1, j1⟩, ⟨t2, j2⟩)
6      if t1 = t2 then return j1 ≤ j2
7      if successor(⟨t1, j1⟩, t2) ≤ j2 then
8          return True
9      else return False
10 function successor(⟨t1, j1⟩, t2)
11     if From_t2(t1) ≥ j1 then return To_t2(t1)
         else return ⊥
12 function predecessor(⟨t1, j1⟩, t2)
13     if To_t1(t2) ≤ j1 then return From_t1(t2)
         else return ⊥
14 function addSuccessor(⟨t1, j1⟩, ⟨t2, j2⟩)
15     if From_t2(t1) < j1 then
16         From_t2(t1) ← j1; To_t2(t1) ← j2
17     else if From_t2(t1) = j1 ∧ To_t2(t1) > j2 then
18         To_t2(t1) ← j2
```

Data Structure (Algorithm 3). The data structure DS succinctly maintains a conflict graph by manipulating the vector clocks $\mathsf{From}_{t_1}(t_2)$ and $\mathsf{To}_{t_1}(t_2)$, for every pair of threads t_1 and t_2, initially set to $\bot$ (Line 3). It provides the following interface, which is used by Algorithm 2.

Function `successor(`$\langle t_1, j_1\rangle, t_2$`)`. This function returns the earliest transaction T_2 in thread t_2 such that $T_1 <^{\mathsf{THB}}_{\sigma[:e]} T_2$, where T_1 is the j_1-th transaction of thread t_1. It first checks whether t_2 is aware of T_1 (i.e., whether $T_1 <^{\mathsf{THB}}_{\sigma[:e]} T_2'$, where T_2' is the latest transaction of t_2) by using $\mathsf{From}_{t_2}(t_1)$, which stores the id of the latest transaction in t_1 that t_2 knows of (thus, t_2 knows of all the earlier transactions of t_1). If not, then no such transaction T_2 exists, and the function returns $\bot$. Otherwise, $\mathtt{id}(T_2)$ is guaranteed to be stored in $\mathsf{To}_{t_2}(t_1)$, which is returned.

Function `predecessor(`$\langle t_1, j_1\rangle, t_2$`)`. This function is symmetric to `successor`, and returns the id of the latest transaction T_2 of t_2 such that $T_2 <^{\mathsf{THB}}_{\sigma[:e]} T_1$, where T_1 is the j_1-th transaction of t_1.

Function `reachable(`$\langle t_1, j_1\rangle, \langle t_2, j_2\rangle$`)`. This function returns True iff $T_1 <^{\mathsf{THB}}_{\sigma[:e]} T_2$, where each T_i is the j_i-th transaction of thread t_i. It is implemented via a simple `successor` query.

Function `addSuccessor(`$\langle t_1, j_1\rangle, \langle t_2, j_2\rangle$`)`. This function records a new ordering $T_1 <^{\mathsf{THB}}_{\sigma[:e]} T_2$, where each T_i is the j_i-th transaction of thread t_i. It starts by checking if T_1 is a later transaction of t_1 known to t_2 than the one t_2 is currently aware of, and if so, it updates $\mathsf{From}_{t_2}(t_1)$ and $\mathsf{To}_{t_2}(t_1)$ accordingly (Line 3). Otherwise, it checks if T_1 is the latest transaction of t_1 known to t_2, but T_2 is a later transaction of t_2 that becomes aware of T_1 than the current transaction of t_2. If so, it updates $\mathsf{To}_{t_2}(t_1)$, accordingly. If none of the above holds, it is guaranteed that the ordering $T_1 <^{\mathsf{THB}}_{\sigma[:e]} T_2$ is already present in the data structure, and is ignored.

Example. For brevity, we write $\mathsf{From}_i(j) = T$ and $\mathsf{To}_i(j) = T$ to mean that they store $\mathtt{id}(T)$. Consider the trace σ_7 in Figure 6. Edge numbers depict the order in which $<^{\mathsf{CHB}}_{\sigma_7}$ relations are processed. The data structure DS is updated as follows. First, executing e_7 updates DS to $\mathsf{From}_{t_4}(t_3) = T_2$ and $\mathsf{To}_{t_4}(t_3) = T_3$, capturing that $T_2 <^{\mathsf{THB}}_{\sigma_7[:e_7]} T_3$. Second, executing e_{11} updates DS to $\mathsf{From}_{t_2}(t_1) = T_1$ and $\mathsf{To}_{t_2}(t_1) = T_4$, capturing that $T_1 <^{\mathsf{THB}}_{\sigma_7[:e_{11}]} T_4$.

Third, executing e_{17} updates DS to $\mathsf{From}_{t_2}(t_1) = T_5$ and $\mathsf{To}_{t_2}(t_1) = T_7$, capturing that $T_5 <^{\mathsf{THB}}_{\sigma_7[:e_{17}]} T_7$. Fourth, executing e_{18} updates DS to $\mathsf{From}_{t_4}(t_2) = T_4$ and $\mathsf{To}_{t_4}(t_2) = T_6$, capturing that $T_4 <^{\mathsf{THB}}_{\sigma_7[:e_{18}]} T_6$. Note that $\mathsf{From}_{t_4}(t_1)$ is still $\bot$, missing the ordering $T_1 <^{\mathsf{THB}}_{\sigma_7[:e_{18}]} T_6$. This is because $T_1 <^{\mathsf{THB}}_{\sigma_7[:e_{18}]} T_6$ is formed transitively, through T_4, but t_2 is aware of the later transaction T_5 of t_1 at this point (due to edge 3). Fifth, executing e_{19} updates DS to $\mathsf{From}_{t_3}(t_2) = T_7$ and $\mathsf{To}_{t_3}(t_2) = T_2$, capturing that $T_7 <^{\mathsf{THB}}_{\sigma_7[:e_{19}]} T_2$. In addition, DS also updates $\mathsf{From}_{t_3}(t_1) = T_5$ and $\mathsf{To}_{t_3}(t_1) = T_2$, $\mathsf{From}_{t_4}(t_2) = T_7$ and $\mathsf{To}_{t_4}(t_2) = T_3$, $\mathsf{From}_{t_4}(t_1) = T_5$ and $\mathsf{To}_{t_4}(t_1) = T_3$ by computing the transitive closure, capturing respectively $T_5 <^{\mathsf{THB}}_{\sigma_7[:e_{19}]} T_2$, $T_7 <^{\mathsf{THB}}_{\sigma_7[:e_{19}]} T_3$, and $T_5 <^{\mathsf{THB}}_{\sigma_7[:e_{19}]} T_3$. Finally, when executing e_{20}, we have $\mathsf{From}_{t_4}(t_1) = T_5$ and $\mathsf{To}_{t_4}(t_1) = T_3$. Since $T_3.\mathtt{begin} \leq^{\mathsf{TO}}_{\sigma_7} \mathtt{txn}(e_7).\mathtt{begin}$ (in fact $\mathtt{txn}(e_7) = T_3$), AtomSanitizer reports a conflict-serializability violation.

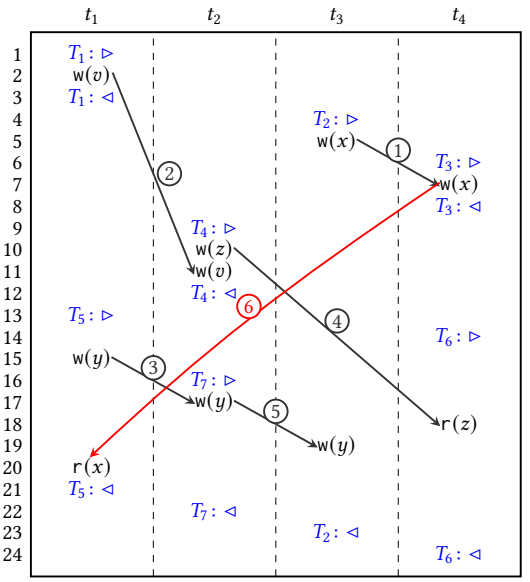


Fig. 6. A trace σ_7 processed by AtomSanitizer. Numbers indicate the sequence of $<^{\mathsf{CHB}}_{\sigma_7}$ edges processed by AtomSanitizer. Edge 5 leads to a violation report.

3.3 Correctness and Complexity

We now state the correctness and complexity of AtomSanitizer. The principle of operation of AtomSanitizer is by keeping track of the latest (open) remote transactions and earliest local transactions, as described in Section 3.1. Formally, it maintains the following invariant.

Lemma 3.3. *After* AtomSanitizer *has processed an event e, for all threads t_1, t_2, if $\mathrm{LRT}^{t_1}_{\sigma[:e]}(t_2)$ is open in $\sigma[:e]$, we have $\mathsf{From}_{t_1}(t_2) = \mathtt{id}(\mathrm{LRT}^{t_1}_{\sigma[:e]}(t_2))$ and $\mathsf{To}_{t_1}(t_2) = \mathtt{id}(\mathrm{ELT}^{t_1}_{\sigma[:e]}(t_2))$.*

In turn, the invariant of Lemma 3.3 combines with Lemma 3.2 to yield the correctness of AtomSanitizer, as stated below.

Lemma 3.4. AtomSanitizer *reports a conflict-serializability violation iff σ is not conflict-serializable. Moreover, if σ is not conflict-serializable, then* AtomSanitizer *reports the first event e of σ such that $<^{\mathsf{THB}}_{\sigma[:e]}$ is cyclic.*

Next, we turn our attention to the complexity of AtomSanitizer. At a high level, each event e that is not a write event results in a single call to checkAndUpdate, which takes $O(k^2)$ time, determined by the two nested loops in Algorithm 2 (Line 2 and Line 2) for performing a transitive closure operation after inserting a single edge in Line 2. On the other hand, when e is a write event, checkAndUpdate may be called $O(k)$ times, due to the loop in Line 1 of Algorithm 1, resulting in $O(k^3)$ time. Owing to the use of the sets $\mathtt{lastReadTids}_x$, however, an amortization argument shows that across all events of σ, the total number of calls to checkAndUpdate is $O(n)$, leading to $O(k^2)$ time per event of σ, on average. In particular, this holds because $\mathtt{lastReadTids}_x$ is reset after each write on x (Line 1), and thus the iterations of that loop can be amortized to the read events that were executed after the last write on x.

Lemma 3.5. *Given a trace σ of n events, k threads, v variables and ℓ locks,* `AtomSanitizer` *runs in $O(nk^2)$ time and $O(k(k+v)+\ell)$ space.*

We remark that the number of locks contribute to the space complexity by a term $O(\ell)$, as opposed to $O(k\ell)$ in existing algorithms. Moreover, although `AtomSanitizer` uses $O(k)$ space for each location x in the worst case, a more precise bound is $O(k_x)$, where k_x is the number of threads reading from x. In practice, most variables are accessed/read by a few threads, decreasing the space complexity to $O(k^2+v+\ell)$. The following theorem concludes the guarantees of `AtomSanitizer`.

Theorem 3.6. *Let σ be an input trace of n events, k threads, v variables and ℓ locks.* `AtomSanitizer` *reports a violation iff σ is not conflict-serializable, and uses $O(nk^2)$ time and $O(k(k+v)+\ell)$ space. If σ is not conflict-serializable,* `AtomSanitizer` *reports the first event e of σ such that $<^{\mathsf{THB}}_{\sigma[:e]}$ is cyclic.*

We further remark that the space usage of `AtomSanitizer` is also bounded by $O(n+k^2)$, but the space bound of Theorem 3.6 is preferred as long as $kv < n$ (which is typically the case).

3.4 Lower Bounds

Finally, in this section we establish two fundamental lower bounds on the time and space complexity of monitoring conflict-serializability.

A super-linear online time lower bound. Recall that, if we relinquish the practical requirement that a violation is reported in an online fashion as soon as it occurs, conflict-serializability violations can be detected in $O(n)$ time, by simply recording the direct $<^{\mathsf{THB}}_{\sigma}$ orderings and checking for the existence of a cycle. `AtomSanitizer` operates in an online setting, and although it also achieves linear running time when $k = O(1)$, its complexity is generally superlinear. Here we prove that such an overhead for the online problem is likely unavoidable, using techniques from fine-grained complexity theory. In particular, the Online Matrix-Vector multiplication (OMv) problem is stated as follows. Given a $m \times m$ matrix M and an online sequence of vectors $v_1, \ldots, v_m$, after a possibly polynomial-time preprocessing of M, the task is to compute each product Mv_i before v_{i+1} is revealed. The corresponding OMv hypothesis [21] states that any algorithm for OMv requires $\Omega(m^{3-\epsilon})$ time, for any fixed $\epsilon > 0$. We establish the following super-linear lower bound based on OMv, which states that the detection on conflict-serializability violations in an online manner is a fundamentally more difficult problem.

Lemma 3.7. *Any algorithm for conflict-serializability violations over traces σ that reports a violation when it occurs (in particular, at the first event e for which $<^{\mathsf{THB}}_{\sigma[:e]}$ is cyclic, or even when $\mathtt{txn}(e)$ closes) must take $\Omega(n^{3/2-\epsilon})$ time, for any fixed $\epsilon > 0$, under the OMv hypothesis.*

We further remark that the lower bound of Lemma 3.7 holds for a "reasonable" number of threads, in particular, for $k = \Theta(\sqrt{n})$ (as opposed to, e.g., $k = \Theta(n)$). In this case, `AtomSanitizer` runs in $O(n^2)$ time, as per Theorem 3.6, and is thus within a $\sqrt{n}$ factor (i.e., sublinear) from optimal.

A linear space lower bound. Note that the space usage of `AtomSanitizer` can become linear in n when there are many locations, e.g., $v = \Omega(n)$. Can we detect atomicity violations using always sublinear space (i.e., not only when $v = o(n)$)? The following lemma states that this is impossible.

Lemma 3.8. *Any streaming algorithm for conflict-serializability/increasing-path violations even over traces σ with only two threads and two transactions must use $\Omega(n)$ space.*

4 Detecting Increasing-Cycle Violations

In this section we turn our attention to increasing-cycle violations. Towards this, we adapt the conditions of Lemma 3.2 to work for the increasing-cycle setting. This will allow us to tune `AtomSanitizer` to report increasing-cycle violations by making slight changes to Algorithm 2.

Latest remote begins and earliest local events. For a trace σ and two threads t_1, t_2, we define the *latest remote begin* of t_2 known to t_1 $\mathrm{LRB}^{t_1}_{\sigma}(t_2)$, as follows.

- If t_1 is not executed in σ, then $\mathrm{LRB}^{t_1}_{\sigma}(t_2) = \bot$. Otherwise, let e_1 be the latest event of t_1 in σ.
- If t_2 has a transaction begin event $\mathtt{begin}_2$ such that $\mathtt{begin}_2 <^{\mathsf{CHB}}_{\sigma} e_1$, then $\mathrm{LRB}^{t_1}_{\sigma}(t_2)$ is the latest such begin event $\mathtt{begin}_2$, otherwise $\mathrm{LRB}^{t_1}_{\sigma}(t_2) = \bot$.

If $\mathrm{LRB}^{t_1}_{\sigma}(t_2) \neq \bot$, we define the *earliest local event* in t_1 for t_2 as $\mathrm{ELE}^{t_1}_{\sigma}(t_2) = e'_1$, where e'_1 is the earliest event in t_1 such that $\mathrm{LRB}^{t_1}_{\sigma}(t_2) <^{\mathsf{CHB}}_{\sigma} e'_1$.

Example. Consider the trace σ_8 in Figure 7. At event e_7, we have $\mathrm{LRT}^{t_3}_{\sigma_8[:e_7]}(t_1) = T_1$, while $\mathrm{LRB}^{t_3}_{\sigma_8[:e_7]}(t_1) = \bot$. At event e_9, $\mathrm{LRB}^{t_3}_{\sigma_8[:e_9]}(t_1) = T_1$, and $\mathrm{ELE}^{t_3}_{\sigma_8[:e_9]}(t_1) = e_9$.

The following lemma is analogous to Lemma 3.2, and states the principle of operation of `AtomSanitizer` for detecting increasing-cycle violations.

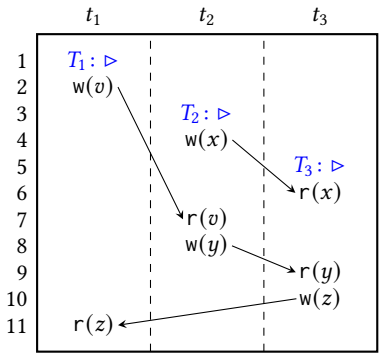


Fig. 7. A trace σ_8.

Lemma 4.1. *The trace σ has an increasing-cycle violation iff there exist two distinct events e_1 and e_2 with $\mathrm{tid}(e_1) = t_1$ and $\mathrm{tid}(e_2) = t_2$ such that the following conditions hold: (i) $t_1 \neq t_2$ and e_1 is a $<^{\mathsf{CHB}}_{\sigma}$-immediate predecessor of e_2, (ii) $\mathrm{LRB}^{t_1}_{\sigma[:e]}(t_2)$ is the* `begin` *event of* $\mathtt{txn}(e_2)$*, and (iii) $\mathrm{ELE}^{t_1}_{\sigma[:e]}(t_2) \leq^{\mathsf{TO}}_{\sigma} e_1$, where e is a $<^{\mathsf{TR}}_{\sigma}$-immediate predecessor of e_2.*

Algorithm 4: Checking increasing-cycle violations with `AtomSanitizer`.

```
1  function checkAndUpdate(v, e)
2      if tid(v) ≠ tid(e) then
3          n1 ← ⟨tid(v), id(v)⟩
4          n2 ← ⟨tid(e), id(txn(e).begin)⟩
5          if DS.reachable(n2, n1) then
6              declare Increasing-cycle violation
7          insertEdge(v, e)
8  function insertEdge(u, v)
9      t1 ← tid(u)
10     nv ← ⟨tid(v), id(v)⟩
11     foreach t ∈ Thr_σ do
12         if t = t1 then j ← id(txn(u).begin)
13         else j ← DS.predecessor(⟨t1, id(u)⟩, t)
14         DS.addSuccessor(⟨t, j⟩, nv)
```

AtomSanitizer for increasing-cycle violations. We modify `AtomSanitizer` to detect increasing-cycle violations by maintaining in the data structure `DS` the latest remote begins $\mathrm{LRB}^{t_1}_{\sigma[:e]}(t_2)$ and earliest local events $\mathrm{ELE}^{t_1}_{\sigma[:e]}(t_2)$, when processing event e of σ, as opposed to latest remote transactions $\mathrm{LRT}^{t_1}_{\sigma[:e]}(t_2)$ and earliest local transactions $\mathrm{ELT}^{t_1}_{\sigma[:e]}(t_2)$, respectively. We achieve this by replacing Algorithm 2 with Algorithm 4. Moreover, now the vector clocks From and To store event ids (see Line 4 and Line 4) instead of the transaction ids. The overall `AtomSanitizer` framework remains the same, in that Algorithm 4 also implements `checkAndUpdate` and `insertEdge`, only `insertEdge` is now somewhat simpler.

Function `checkAndUpdate(`v, e`)`. This function is similar to the corresponding one in Algorithm 2, with the difference that n_1 in Line 4 represents the event v conflicting with e, as opposed to the transaction of v.

Function `insertEdge(`u, v`)`. This function is similar to the corresponding one in Algorithm 2, and is responsible for registering a new ordering $v <^{\mathsf{CHB}}_{\sigma[:e]} e$ captured in `checkAndUpdate` (Line 4). The main difference is that the transitive closure only requires backward propagation (Line 4), as opposed to both forward and backward done in Algorithm 2. This is because e cannot have any $<^{\mathsf{CHB}}_{\sigma[:e]}$ successors. In turn, this reduces the running time for inserting an edge to $O(k)$, as opposed to $O(k^2)$ in the case of Algorithm 2.

Example. For brevity, we write $\mathsf{From}_i(j) = e$ and $\mathsf{To}_i(j) = e$ to mean that they store $\mathtt{id}(e)$. Consider the trace σ_9 in Figure 8. First, executing e_4 updates DS to $\mathsf{From}_{t_2}(t_1) = e_1$ and $\mathsf{To}_{t_2}(t_1) = e_4$, capturing that $e_1 <^{\mathsf{CHB}}_{\sigma_9[:e_4]} e_4$. Second, executing e_9, updates DS to $\mathsf{From}_{t_3}(t_2) = e_8$ and $\mathsf{To}_{t_3}(t_2) = e_9$, capturing that $e_8 <^{\mathsf{CHB}}_{\sigma_9[:e_9]} e_9$. In addition, DS also updates $\mathsf{From}_{t_3}(t_1) = e_1$ and $\mathsf{To}_{t_3}(t_1) = e_9$, capturing $e_1 <^{\mathsf{CHB}}_{\sigma_9[:e_9]} e_9$, by computing the transitive closure. Finally, when executing e_{12}, we have $\mathsf{From}_{t_3}(t_1) = e_1$ and $\mathsf{To}_{t_3}(t_1) = e_9$. Since $e_9 \leq^{\mathsf{TO}}_{\sigma_9} e_{10}$, `AtomSanitizer` reports an increasing-cycle violation.

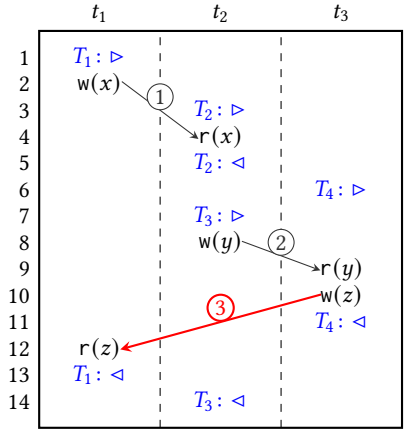


Fig. 8. A trace σ_9 with an increasing-cycle violation. Numbers indicate the order of $<^{\mathsf{CHB}}_{\sigma_9}$ edges inferred by `AtomSanitizer`. Edge 3 leads to a violation report.

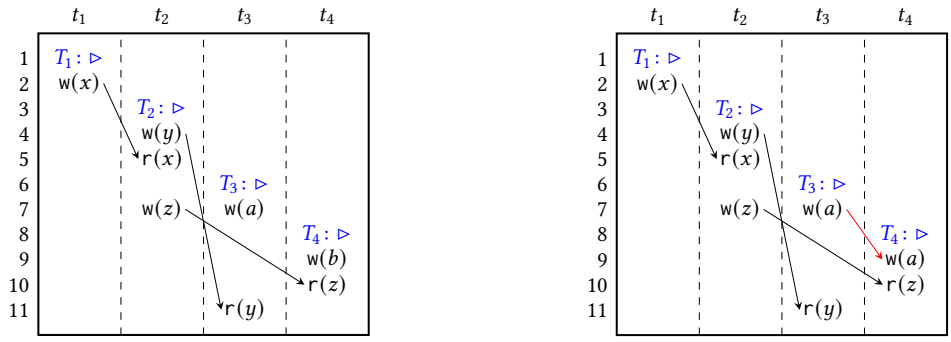
Correctness and complexity. The correctness of `AtomSanitizer` for increasing-cycle violations follows from Lemma 4.1, while its time complexity is similar to that for conflict-serializability (Lemma 3.5), taking into consideration that each call to `insertEdge` now takes $O(k)$ amortized time, as opposed to $O(k^2)$. Formally, we have the following theorem.

Theorem 4.2. *Let σ be an input trace of n events, k threads, v variables and ℓ locks.* `AtomSanitizer` *(increasing-cycle) reports a violation iff σ has an increasing-cycle violation , and uses $O(nk)$ time and $O(k(k+v)+\ell)$ space.*

Comparison to existing work. Standard algorithms for the conflict-happens-before order, and thus increasing-cycle violations, operate in $O(nk)$ time and $O(k(k+v+\ell))$ space [28, 30]. `AtomSanitizer` has the same time complexity, but uses a factor k less per lock in space. Moreover, as already stated in the case of conflict-serializability violations in Section 3.2, `AtomSanitizer` only uses $O(1)$ space for each pair (t, x) such that thread t accesses variable x, which in practice is less than storing a vector clock per variable, in contrast to [28, 30].

5 Concurrent Runtime Setting

We now turn our attention to the online setting, where atomicity must be monitored during program execution. This poses the challenge of sharing the data structure DS among threads, and accessing/updating it concurrently. To avoid data races (which can lead to wrong results), such accesses must be appropriately locked. Existing atomicity checkers address this challenge by forcing each thread to acquire a global lock at each step, which can cause considerable congestion. Instead,



(a) A trace σ_{10} where $T_3 \not<^{\mathsf{THB}}_{\sigma_{10}} T_4$.

(b) A trace σ_{11} where $T_3 <^{\mathsf{THB}}_{\sigma_{11}} T_4$.

Fig. 9. In σ_{10} (a), the transactions T_3 and T_4 do not have outgoing $<^{\mathsf{THB}}_{\sigma_{10}}$-orderings, and thus Algorithm 5 can process the events e_{10} and e_{11} concurrently, without acquiring a global lock, even though it has to perform a (backwards) transitive-closure to make t_3 and t_4 aware of T_1. In contrast, in σ_{11} (b), T_3 has an outgoing ordering (marked in red), making Algorithm 5 acquire a global lock when processing e_{11}.

the data structure DS of `AtomSanitizer` optimizes for the case when the transaction of the thread executing the current step is not known to any other thread, which is by far the most common case. In such cases, updates on DS can be performed concurrently via a combination of read and write locks, while completely avoiding the use of a global lock, and therefore reducing congestion. Figure 9 provides an illustration.

Online `AtomSanitizer`. Algorithm 5 shows the pseudocode of a wrapper that allows concurrent accesses to DS. For the online setting, `AtomSanitizer` consists of the same components (Algorithm 1, Algorithm 2, Algorithm 3), but calls to `DS.reachable` and `DS.insertEdge` from Algorithm 2 are replaced by calls to the same function in Algorithm 5. Conceptually, DS is shared between threads, so that From_t and To_t belong to thread t. Concurrent accesses to DS are protected by (i) a pair of read/write locks $\mathbb{L}_t$ for each thread t, and (ii) a global read/write lock $\mathbb{L}$. We now describe the functions of Algorithm 5 in more detail.

Function `DS.reachable`. This function simply wraps the reachability call to DS around read-acquiring the lock $\mathbb{L}_{t_2}$ of the target thread t_2. The function is called from thread t_1, and no other thread ever write-acquires $\mathbb{L}_{t_1}$, thus t_1 does not need to read-acquire its own lock.

Function `insertEdge`. This function operates as follows. First, it calls `insertEdgeBackwards`, which attempts to insert the edge while making only backwards propagation (Line 5). If it fails, `insertEdge` performs the full transitive closure, by acquiring the global lock $\mathbb{L}$, as well as all locks $\{\mathbb{L}_{t_i}\}_i$ (Line 5), to avoid racing with any other thread updating DS. Finally, the variable $\texttt{latestTxn}_t$ stores the id of the latest transaction T that thread t has executed, and it is appropriately initialized every time t starts a new transaction (not explicitly shown in the pseudocode). The flag $\texttt{hasOutEdge}_t$ marks whether $\texttt{latestTxn}_t$ has outgoing edges, helping to avoid locking the whole DS in future updates. It is set to True when inserting an edge (v, e) where v belongs to the latest transaction of its thread, and T is not unary (Line 5) and is set to False when T is closed (not explicitly shown in the pseudocode).

Function `insertEdgeBackwards`. Finally, this function attempts to perform a backwards edge propagation from the thread $t_2 = \mathrm{tid}(e)$. If successful, this only modifies the clocks From_{t_2} and To_{t_2} owned by t_2. In high level, it first read-acquires the global lock $\mathbb{L}$ (Line 5) to avoid racing with a thread that might be performing a full transitive closure computation, thereby also modifying the

clocks From_{t_2} and To_{t_2}. To perform the backwards propagation, t_2 also needs to read the clocks From_{t_1} and To_{t_1}. For this, it read-acquires the lock $\mathbb{L}_{t_1}$. This acquisition is done in a loop (Line 5) to avoid deadlocks occurring when t_1 simultaneously read-acquires $\mathbb{L}_{t_2}$. After t_2 has read-acquired $\mathbb{L}_{t_1}$, it again checks whether $\texttt{txn}(e)$ has an outgoing edge. Note that such an edge might have been added between the call to `insertEdgeBackwards` made by `insertEdge` (Line 5) and when $\mathbb{L}$ was read-acquired in `insertEdgeBackwards` (Line 5). If $\texttt{txn}(e)$ indeed has an outgoing edge at this point, `insertEdgeBackwards` fails and returns False to `insertEdge`, forcing the latter to perform a full transitive closure (Line 5). Otherwise, `insertEdgeBackwards` proceeds to perform the backward propagation (Line 5). Finally, lines 38–40 update the flag $\texttt{hasOutEdge}_{t_1}$ to indicate whether $\texttt{latestTxn}_{t_1}$ has outgoing edges, similarly to lines 11–13 in `insertEdge`.

Algorithm 5: Concurrent wrapper around DS.

```
1  function reachable(⟨t1, j1⟩, ⟨t2, j2⟩)
2      Rlock(L_t2)
3      r ← DS.reachable( ⟨t1, j1⟩, ⟨t2, j2⟩ )
4      RUnlock(L_t2)
5      return r

6  function insertEdge(v, e)
7      t1 ← tid(v)
8      if insertEdgeBackwards(v, e) then
9          return
10     // executes rarely in practice
11     T ← atomic_load(latestTxn_t1)
12     if unary(v) = False ∧ T = txn(v) then
13         atomic_store(hasOutEdge_t1, True)
14     WLock(L)
15     foreach t ∈ Thr_σ do
16         Rlock(L_t)
17     full transitive closure as in Algorithm 2
18     foreach t ∈ Thr_σ do
19         RUnlock(L_t)
20     WUnlock(L)

21 function insertEdgeBackwards(v, e)
22     t1 ← tid(v) ; t2 ← tid(e)
23     n_e ← ⟨t2, id(txn(e))⟩
24     Rlock(L)
25     while True do
26         Rlock(L_t1)
27         if tryWLock(L_t2) then break
28         else RUnlock(L_t1)
29     if atomic_load(hasOutEdge_t2) = True then
30         // executes rarely in practice
31         WUnlock(L_t2); RUnlock(L_t1); RUnlock(L);
32         return False
33     foreach t ∈ Thr_σ do
34         if t = t1 then  j ← id(txn(v))
35         else  j ← DS.predecessor(⟨t1, id(txn(v))⟩, t)
36         DS.addSuccessor(⟨t, j⟩, n_e)
37     WUnlock(L_t2); RUnlock(L_t1); RUnlock(L)
38     T ← atomic_load(latestTxn_t1)
39     if unary(v) = False ∧ T = txn(v) then
40         atomic_store(hasOutEdge_t1, True)
41     return True
```

Correctness. We now outline the correctness for Algorithm 5, by focusing on three key aspects.

Deadlock-freeness. First, note that the algorithm cannot deadlock: `reachable` does not have nested locking, whereas in all other cases, acquiring a thread lock of the form $\mathbb{L}_{t_i}$ (in lines 16, 26, 27) is preceded by accessing the global lock $\mathbb{L}$ (in lines 14, 24). Note, however, that the hot path is in lines 25 to 28 in `insertEdgeBackwards`, where $\mathbb{L}$ is read-acquired, thus allows the concurrent execution of multiple threads, avoiding congestion. If multiple readers hold $\mathbb{L}$, deadlocks are avoided due to the use of $\texttt{tryWLock}(\mathbb{L}_{t_2})$ at line 27. This can cause a livelock, which in practice resolves quickly.

Mutual exclusion. Second, the algorithm guarantees mutual exclusion: every time a full transitive closure is started by a thread, the data structure is protected by $\mathbb{L}$ (line 14) so that no other thread can modify it. Moreover, the locks $\mathbb{L}_t$ of all threads t have been read-acquired (line 16), which also

avoids write-read races. All other data structure updates are such that thread t_2 only writes on the From_{t_2} and To_{t_2} clocks at line 36, which avoids write-write races. Moreover, write-read races are avoided because whenever thread t_2 accesses the vector clocks From_{t_1} and To_{t_1} of some other thread t_1 (line 35), it has first read-locked $\mathbb{L}_{t_1}$ (line 26) and write-locked $\mathbb{L}_{t_2}$ (line 27).

A benign race condition. Finally, notice a subtlety with regards to the accesses of the flag $\mathtt{hasOutEdge}_{t_1}$. Here, a race condition may cause some thread t executing lines 13 and 40 to set $\mathtt{hasOutEdge}_{t_1}$, even if thread t_1 has progressed with a new transaction in the meantime. This does not threaten correctness, but may only cause t_1 to execute line 29 later and thus may cause congestion. In practice, however, this occurs rarely to warrant the overhead of avoiding this race condition via locking.

6 Experimental Evaluation

In this section, we report on an implementation of `AtomSanitizer`, both as a standalone atomicity checker, and as a runtime monitor inside `TSAN`. We also report on an experimental evaluation of `AtomSanitizer` in each setting, and compare it to other atomicity checkers. All experiments are run on an Ubuntu 22.04 machine with 2.4GHz CPU and 64GB of memory.

6.1 Standalone Experiments

Implementation. We implement `AtomSanitizer` inside the `RAPID` framework for dynamic analyses [29] in Java, closely following the pseudocode in Section 3 (for conflict-serializability violations) and Section 4 (for increasing-cycle violations).

Baselines. We compare the performance of `AtomSanitizer` to the standard and SOTA atomicity checkers (i) `Velodrome` [16], (ii) `Aerodrome` [32], and (iii) `RegionTrack` [28]. For this, we port `RegionTrack` into `RAPID`, which already implements `Velodrome` and `Aerodrome`. Each tool processes the same input trace and reports whether it is conflict-serializable. We also compare `AtomSanitizer` and `RegionTrack` for increasing cycles, since they are the only tools that report increasing cycles in a sound and complete manner. By default, `RegionTrack` performs conflict-serializability and increasing-cycle checking simultaneously. For a fair comparison, we have separated the two components, which reduces its running time for each individual task.

Benchmarks. Our benchmark traces are derived from the DeCaPo benchmark suite [6], Java Grande suite [43], and microbenchmarks [49], taken from the experimental setup of `Aerodrome` [32] and also used in other works [5, 28]. Benchmarks ending in "`_ext`" are our own variants of existing benchmarks that scale up the number of threads k. We report the average running time of each tool over 10 runs, imposing a timeout of 2 hours in each run.

Results. The results are shown in Table 2, for both conflict-serializability and increasing-cycle violations. For conflict-serializability, we observe that `AtomSanitizer` is faster than each of the baselines, on each benchmark. The geometric mean of the speedup across all benchmarks is 2.0×, 4.5×, and 12× over `RegionTrack`, `Aerodrome`, and `Velodrome`, respectively. Overall, `AtomSanitizer` and `RegionTrack` stand out as the faster algorithms compared to `Aerodrome` and `Velodrome`, with `AtomSanitizer` being decisively the fastest. The comparison on increasing-cycle violations is similar, with `AtomSanitizer` being faster than `RegionTrack` on each benchmark. The geometric mean of the speedup over all benchmarks is 1.7×.

Table 2. Offline experiments on conflict-serializability and increasing-cycle violations. Entries marked with * witness a violation (both conflict-serializability and increasing-cycle).

Benchmark	n	k	v	Conflict Serializability (ms)				Increasing Cycle (ms)	
				AtomSanitizer	RegionTrack	Aerodrome	Velodrome	AtomSanitizer	RegionTrack
philo	557	6	24	3	5	5	4	3	3
hedc*	10.9K	7	1.69K	10	14	29	26	9	10
sunflow_ext*	4.59M	130	612K	94	428	569	1,376	113	389
sunflow*	16.7M	16	1.26M	1,374	2,431	73,710	42,910	1,286	2,318
series*	40.1M	4	20.0K	275	540	822	44,767	270	503
tomcat*	68.3M	48	6.10M	938	4,837	7,889	18,930	1,007	4,260
fop	95.8M	1	5.30M	4,415	4,768	12,040	8,299	4,043	4,378
raytracer_ext	105M	129	634K	22,336	40,238	371,176	TO	20,146	39,874
montecarlo_ext*	114M	129	8.13M	96	637	754	2,399	102	424
crypt*	126M	7	9.00M	4,314	8,311	15,949	36,019	5,564	6,087
lufact*	135M	4	252K	316	381	733	385	307	379
lufact_ext*	149M	129	252K	693	2,647	3,672	3,625	725	2,413
batik*	189M	6	4.90M	4,871	5,571	16,464	48,630	4,413	4,822
moldyn_ext	233M	129	13.3M	48,933	202,136	411,028	TO	62,091	148,275
tsp*	312M	9	181.M	240	306	777	435	249	322
pmd*	367M	13	12.9M	241	379	699	538	234	362
montecarlo	494M	4	30.5M	36,685	45,255	146,191	TO	37,205	45,022
luindex*	570M	3	2.49M	28,121	32,002	77,320	44,565	25,063	29,662
sor*	608M	4	1.00M	1,409	1,715	3,024	3,740	1,455	1,252
series_ext	694M	129	347K	8,066	14,610	19,732	TO	7,927	13,929
sparsematmult*	726M	4	1.59M	66,967	131,849	241,929	540,892	85,044	106,658
xalan*	1.07B	13	31.3M	2,029	5,622	13,466	52,534	2,038	4,768
moldyn	1.75B	4	121K	112,041	149,492	439,822	TO	103,337	140,665
Total	-	-	-	**344,466**	**654,175**	**1,857,798**	**18,850,074**	**362,630**	**556,772**

6.2 Runtime Experiments

In this section, our goal is to evaluate the time and memory performance of AtomSanitizer in a concurrent runtime setting, and compare it to the data race detection engine of TSAN, which is an industry standard in concurrent runtime analyses.

Implementation. We implement AtomSanitizer inside TSAN [42] in C++, following the description in Section 5. We also enable the default optimizations of TSAN, such as thread pooling and storing only the four latest events per memory address.

Benchmarks. Our benchmark set consists of 13 popular open-source projects known for high concurrency, also used in related works (e.g., [46]). Since these programs do not have explicit transactions, we create an atomicity specification for each, following a process used in [5], and has been provided the specifications of prior work (e.g., [28, 32]). In high level, this process starts with a fine-grained specification which marks each function as atomic. Then, as long as an atomicity violation is reported, it refines the specification by removing the transactions surrounding the function on which the atomicity violation was reported. This creates atomicity specifications for which we expect no atomicity violations, while retaining the atomicity requirement of many

Table 3. Online experiments on conflict-serializability (AtomSanitizer) and race detection (TSAN) inside the TSAN framework. "Origin" refers to the original, uninstrumented benchmark. "OvhTsan" and "OvhOrig" denote the ratios of AtomSanitizer over TSAN and Origin respectively. "Total/Max" refers to time/memory measurements, respectively. "Total Overhead" is the ratio of the total running times; "Max Overhead" is the ratio of the maximum memory usages.

Benchmark	Time (s)					Memory (MB)				
	Origin	AtomSanitizer	TSAN	**OvhOrig**	**OvhTsan**	Origin	AtomSanitizer	TSAN	**OvhOrig**	**OvhTsan**
x265	4	19	13	4.66	1.54	127	458	450	3.60	1.02
x264	55	315	143	5.74	2.21	34	125	121	3.62	1.03
pbzip2	15	18	18	1.17	1.00	6	38	30	6.81	1.26
libwebp	14	48	42	3.53	1.16	117	371	362	3.18	1.02
libvpx	10	33	18	3.16	1.80	21	84	78	4.05	1.08
lbzip2	6	92	56	14.69	1.66	45	126	118	2.81	1.06
pigz	27	56	31	2.05	1.78	2	41	29	21.64	1.38
pixz	531	583	588	1.10	0.99	876	1,497	1,489	1.71	1.01
xz	52	205	183	3.93	1.12	875	2,649	2,640	3.03	1.00
ImageMagick	21	320	290	15.06	1.10	681	2,041	2,031	2.99	1.00
Rocksdb	4	24	23	5.99	1.05	25	69	67	2.76	1.04
7z	10	198	53	20.04	3.78	195	609	601	3.12	1.01
ffmpeg	2	8	5	5.49	1.81	38	156	157	4.06	0.99
Total/Max	**752**	**1,922**	**1,462**	**2.56**	**1.31**	**876**	**2,649**	**2,640**	**2.72**	**1.01**
Geo Mean	-	-	-	**4.63**	**1.49**	-	-	-	**3.83**	**1.06**

functions, so that benchmarking remains informative. The absence of violations is also necessary to meaningfully compare AtomSanitizer and TSAN, as otherwise an atomicity violation would make AtomSanitizer halt early, and appear (unfairly) faster. We report the average running time and memory used by AtomSanitizer and TSAN over 20 runs.

Results. Our results are shown in Table 3. As a first observation, we see that the overhead of TSAN over the uninstrumented program is 3.1× on (geometric) average, and 13.8× in the worst case (ImageMagic), in terms of time. This matches the expectations of the developers for an overhead $\leq$ 15× in time [42]. In terms of memory, TSAN has an overhead of 3.5× on average, and 14.5× in the worst case (pigz). Although the average case is well within the $\leq$ 10× bound expected by the developers, the worst case exceeds it. This, however, occurs on a benchmark with a very small memory footprint (2MB), and thus is insignificant in practice.

Compared to TSAN, AtomSanitizer incurs a reasonable time overhead, averaging 1.49× over all benchmarks. The largest overhead occurs on x264 and 7z, reaching 2.21×, and 3.78×, respectively. For the two benchmarks, we observe that AtomSanitizer performs a fully transitive closure more times, which is the main reason for the slow down, as it requires heavier locking (see Section 5). Finally, we observe that the memory footprint of AtomSanitizer is nearly identical to TSAN, with the geometric mean of the overhead being only 1.06×. Compared to the original, uninstrumented executions, the geometric mean of the overheads over time and memory are 4.63× and 3.83×, respectively. The time overhead is always $\leq$ 15×, except for one case that goes up to $\simeq$ 20× (7z). The memory overhead is always $\leq$ 6×, except for one case that goes up to $\simeq$ 21× (pigz). Note, however, that TSAN also suffers a high memory overhead on this benchmark ($\simeq$ 14×). Overall, our

results support the hypothesis that AtomSanitizer can be incorporated in a runtime monitoring setting, incurring acceptable overheads in both time and memory.

7 Related Work

Detecting atomicity vulnerabilities automatically has been an active research topic and targeted by a range of methods. Atomizer [13] was a precursor of Velodrome, based on Lipton's reduction theory [25]. Techniques like AtomFuzzer [36] and PENELOPE [44] aim at generating schedules that expose atomicity violations, while others focus on synthesizing tests for the same task [40]. Algorithms like AtomSanitizer can speed up such approaches, by efficiently testing whether the generated execution indeed constitutes an atomicity violation. Since atomicity invariants are not always explicit, several approaches aim at inferring them automatically [27, 37]. Finally, atomicity has also been targeted by static analyses, most prominently using types and effects [15, 17, 41].

View serializability is another atomicity notion, which is more general than conflict-serializability. Intuitively, the serial witness trace σ^* can now be obtained from the observed trace σ by swapping conflicting events, as long every read observes the same write in σ and σ^*. However, testing view serializability is much harder: it is NP-complete in general [35], and even hard when parameterized by the number of threads [31].

Dynamic analyses are a standard approach to concurrency bugs. Besides atomicity violations, they have been targeting data races [14, 20, 24, 38], deadlocks [47], memory vulnerabilities [8], and linearizability violations [1, 11, 19, 23]. Their efficiency has been an active pursuit in both theory and practice [7, 22, 30, 39, 46].

8 Conclusion

We have introduced AtomSanitizer, a new algorithm for dynamically testing for atomicity violations. AtomSanitizer is theoretically faster than all previous algorithms for the problem, while it also has small space complexity. Our experimental results show that the theoretical advantage of AtomSanitizer is also realized in practice, while its lightweight time and memory footprint make it suitable for a runtime monitoring setting. Interesting future directions include transferring the insights of this work towards monitoring transactional consistency in databases, against a range of popular isolation levels [4, 9, 33].

Data-Availability Statement

Data for our experiments and the experimental setup can be found in the accompanying artifact [48]. All proofs and additional details of experiments can be found in the technical report [45].

Acknowledgments

This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN, and by a research grant from STIBOFONDEN.

References

[1] Parosh Aziz Abdulla, Samuel Grahn, Bengt Jonsson, Shankaranarayanan Krishna, and Om Swostik Mishra. 2025. Efficient Linearizability Monitoring. *Proc. ACM Program. Lang.* 9, PLDI, Article 225 (June 2025), 24 pages. doi:10.1145/3729328

[2] Rajeev Alur, Ken McMillan, and Doron Peled. 2000. Model-Checking of Correctness Conditions for Concurrent Objects. *Information and Computation* 160, 1 (2000), 167–188. doi:10.1006/inco.1999.2847

[3] Shane Bester. 2008. *Atomicity violation leading to crash in MySQL 6.0.7.* MYSQL AB. https://bugs.mysql.com/bug.php?id=38816

[4] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 165:1–165:28. doi:10.1145/3360591

[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)*. Association for Computing Machinery, New York, NY, USA, 28–39. doi: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. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190. doi:10.1145/1167515.1167488

[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. *SIGPLAN Not.* 48, 10 (Oct. 2013), 693–712. doi:10.1145/2544173.2509519

[8] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: bit-encoding online memory leak detection. *SIGPLAN Not.* 41, 11 (Oct. 2006), 61–72. doi:10.1145/1168918.1168866

[9] Jack Clark, Alastair F. Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 754–768. doi:10.1145/3627703.3650080

[10] Jeremy Cole. 2008. *Atomicity violation leading to crash in MySQL 5.4.3.* MySQL AB. https://bugs.mysql.com/bug.php?id=38883

[11] Michael Emmi and Constantin Enea. 2017. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.* 2, POPL, Article 25 (Dec. 2017), 27 pages. doi:10.1145/3158113

[12] Azadeh Farzan and P. Madhusudan. 2008. Monitoring Atomicity in Concurrent Programs. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer, Berlin, Heidelberg, 52–65. doi:10.1007/978-3-540-70545-1_8

[13] Cormac Flanagan and Stephen N Freund. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.* 39, 1 (Jan. 2004), 256–267. doi:10.1145/982962.964023

[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)*. Association for Computing Machinery, New York, NY, USA, 121–133. doi:10.1145/1542476.1542490

[15] 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. doi:10.1145/1377492.1377495

[16] 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)*. Association for Computing Machinery, New York, NY, USA, 293–303. doi:10.1145/1375581.1375618

[17] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. *SIGPLAN Not.* 38, 5 (May 2003), 338–349. doi:10.1145/780822.781169

[18] Jocelyn Fournier. 2004. *Atomicity violation leading to crash in MySQL 4.1.2.* MySQL AB. https://bugs.mysql.com/bug.php?id=3596

[19] Phillip B. Gibbons, John L. Bruno, and Steven Phillips. 2002. Black-Box Correctness Tests for Basic Parallel Data Structures. *Theory of Computing Systems* 35, 4 (Aug. 2002), 391–432. doi:10.1007/s00224-002-1046-6

[20] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. Yashme: detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 830–845. doi:10.1145/3503222.3507766

[21] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing (STOC '15)*. Association for Computing Machinery, New York, NY, USA, 21–30. doi:10.1145/2746539.2746609

[22] Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.CONCUR.2021.16*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Paris, France, 16:1–16:23. doi:10.4230/LIPIcs.CONCUR.2021.16

[23] Zheng Han Lee and Umang Mathur. 2025. Efficient Decrease-and-Conquer Linearizability Monitoring. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 345 (Oct. 2025), 28 pages. doi:10.1145/3763123

[24] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 443–457. doi:10.1145/3009837.3009857

[25] Richard J. Lipton. 1975. Reduction: a method of proving properties of parallel programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. doi:10.1145/361227.361234

[26] 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)*. Association for Computing Machinery, New York, NY, USA, 329–339. doi:10.1145/1346281.1346323

[27] 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* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/1168857.1168864

[28] Xiaoxue Ma, Shangru Wu, Ernest Pobee, Xiupei Mei, Hao Zhang, Bo Jiang, and Wing-Kwong Chan. 2021. RegionTrack: A Trace-Based Sound and Complete Checker to Debug Transactional Atomicity Violations and Non-Serializable Traces. *ACM Transactions on Software Engineering and Methodology* 30, 1 (Dec. 2021), 7:1–7:49. doi:10.1145/3412377

[29] Umang Mathur. 2024. Rapid: Dynamic Analysis for Concurrent Programs. https://github.com/focs-lab/rapid. Accessed: 2024-12-03.

[30] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 710–725. doi:10.1145/3503222.3507734

[31] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Saarbrücken Germany, 713–727. doi:10.1145/3373718.3394783

[32] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 183–199. doi:10.1145/3373376.3378475

[33] Lasse Møldrup and Andreas Pavlogiannis. 2025. AWDIT: An Optimal Weak Database Isolation Tester. *Artifact for "AWDIT: An Optimal Weak Database Isolation Tester"* 9, PLDI (June 2025), 209:1540–209:1564. doi:10.1145/3742465

[34] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 267–280. doi:10.5555/1855741.1855760

[35] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. doi:10.1145/322154.322158

[36] 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)*. Association for Computing Machinery, New York, NY, USA, 135–145. doi:10.1145/1453101.1453121

[37] 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)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/1508244.1508249

[38] Andreas Pavlogiannis. 2020. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (2020), 29 pages. doi:10.1145/3371085

[39] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient Predictive Race Detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 747–762. doi:10.1145/3385412.3385993

[40] 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)*. Association for Computing Machinery, New York, NY, USA, 131–142. doi:10.1145/2786805.2786874

[41] 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. doi:10.1145/1065944.1065956

[42] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. doi:10.1145/1791194.1791203

[43] L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, Colorado) *(SC '01)*. Association for Computing Machinery, New York, NY, USA, 8. doi:10.1145/582034.582042

[44] 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)*. Association for Computing Machinery, New York, NY, USA, 37–46. doi:10.1145/1882291.1882300

[45] Hünkar Can Tun, Yifan Dong, and Andreas Pavlogiannis. 2026. Fast Atomicity Monitoring. arXiv:2604.11369 [cs.PL] https://arxiv.org/abs/2604.11369

[46] Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Cirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 223–238. doi:10.1145/3620666.3651358

[47] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (jun 2023), 26 pages. doi:10.1145/3591291

[48] Hünkar Can Tunç, Yifan Dong, and Andreas Pavlogiannis. 2026. *Fast Atomicity Monitoring*. doi:10.5281/zenodo.19615809

[49] Christoph von Praun and Thomas R. Gross. 2003. Static conflict analysis for multi-threaded object-oriented programs. *SIGPLAN Not.* 38, 5 (May 2003), 115–128. doi:10.1145/780822.781145