To appear in IEEE Transactions on Reliability: https://doi.org/10.1109/TR.2022.3176680

Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations

Xiaoxue Ma, Imran Ashraf, and W.K. Chan

Abstract— **Atomicity is a desirable property for multithreaded programs. In such programs, a transaction is an execution of an atomic code region that may contain memory accesses on an arbitrary number of shared variables. When transactions are not conflicting with one another in a trace, they greatly simplify the reasoning of the program correctness. If a transaction incurs an atomicity violation in a trace, developers have to debug the code, but this is challenging. To achieve practical runtime performances, existing dynamic techniques for detecting such atomicity violations face a challenge: They are designed for either detecting all such atomicity violations without the capability of localizing the corresponding cross-thread transaction sequences or deliberately missing some atomicity violations in the trade of localizing some of them to support their atomicity violation detection. In this paper, we propose Davida, a novel technique to address this problem. Davida efficiently tracks selective transactions and cross-thread dependency sequences over transactions reachable from the currently active transactions of all the threads in a decentralized manner. We prove that Davida precisely accomplishes every atomicity violation in a trace with an actual sequence of transactions triggering the violation. The experimental results on 15 subjects showed that Davida outperformed Velodrome, the previous graph-based state-of-the-art technique, in both performance and completeness.**

Index Terms—**dynamic analysis, localization of transaction sequences, atomicity violation detection, decentralized approach**

I. INTRODUCTION

Atomicity is a desirable correctness criterion for multithreaded programs [30]. When units of work in program traces are not conflicting with one another, they greatly simplify the reasoning of the program correctness. Such a unit of work is called a **transaction**, and the region of code protected by the transaction is called an *atomic region*. On the other hand, if a program trace exposes an atomicity violation bug, dynamic techniques able to detect all the instances of the atomicity violations ensure that this concurrency bug is not coincidentally missed in the program testing process.

A shared variable may be accessed by multiple threads. If two threads each accesses the same shared variable and their order of accesses on it (such as a write-read order) must be followed to maintain the program semantics, then there is a *variable-level dependency* between the two accesses.

A specific case of atomicity violation is the one merely involving two threads and a small set of shared variables. Their detection is based on matching against an explicit and precise set of atomicity violation patterns [32][53], where each pattern specifies a sequence of the variable-level dependencies among this set of shared variables [27][32][37][51][52][53]. Since each violation of each pattern directly matches the sequence of variable-level dependencies in the trace under analysis, the sequence is readily available when the violation is detected. Developers can use such sequences for debugging the code.

However, the detection of atomicity violation in a more general case, which is referred to as *atomicity violation at the transaction level* [13][14][16][33], or **atomicity violation** for short, is different. Its goal is to determine whether every transaction can be viewed as being executed atomically without violating any variable-level dependencies in the trace. Moreover, like the special case above, if this is not the case, it is desirable to report each transaction that incurs an atomicity violation as well as the dependency sequence triggering the violation.

We briefly revisit a few terminologies for easing us to summarize our work in this section. The detailed introduction of these terminologies can be found in Section II. The status of a transaction can be *active* or *completed*. An active transaction [34] sequentially executes instructions, which may access shared variables or operate on lock objects, generating corresponding *events* of interest for atomicity violation detection. When a transaction starts, its status is active. A transaction is *completed* if it reaches the endpoint of the transaction. A *completed* transaction cannot execute any instruction. Suppose a transaction tx generates a pair of events $\langle e_a, e_b \rangle$. If the transaction cannot execute atomically (i.e., the status of the transaction cannot be changed from active to completed) without these two events being interfered by other threads' transactions, the transaction produces an ***atomicity violation*** due to the event pair e_a and e_b *when* executing e_b [13][14][16][33]. Moreover, a ***dependency*** $tx_i \rightsquigarrow tx_j$ from one transaction tx_i to another transaction tx_j is due to the pair of conflicting events of a variable-level dependency executed by tx_i and tx_j, respectively [3][16]. In a dependency sequence θ, every pair of consecutive transactions either has a variable-level dependency from the former transaction to the latter one or is executed by the same thread with the former transaction happening before the later one. The sequence is said to be ***increasing*** [16] if every transaction in θ generates events consistent with the temporal order of events in

This work was supported in part by Innovative Technology Fund of HKSAR (project no. 9440226) and CityU MF_EXT (project no. 9678180).

Xiaoxue Ma, and Imran Ashraf are with the Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong (email: {xiaoxuema2-c, iashraf3-c}@my.cityu.edu.hk).

W.K. Chan (*corresponding author*) is with the Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong (email: wkchan@cityu.edu.hk).

θ, otherwise ***non-increasing***. Suppose *tx* is the first transaction in the sequence θ and generates a pair of events $\langle e_a, e_b\rangle$ that causes an atomicity violation occurring on *tx*. The sequence θ is called a ***witness*** for this atomicity violation if the events e_a and e_b are the first and the last events in θ, respectively.

Precisely checking all atomicity violations with witnesses efficiently is challenging. A fundamental problem is due to the path-insensitive nature of dependency captured in the graph. As we will present in the motivating example, existing techniques [3][16] must locate a dependency sequence before determining whether or not it is increasing. A dependency may be reachable from multiple transactions along the edges in their directed graphs, forming different dependency sequences starting from these transactions. Nonetheless, a dependency may be increasing with respect to one transaction but non-increasing with respect to another one, making these techniques unable to eliminate the tracking of non-increasing dependency sequences with respect to each transaction. Besides, these techniques only keep one edge between the same two transactions. Only enforcing them to track more edges between the same two transactions cannot resolve all false negatives in detecting atomicity violations with witnesses, unless tracking all edges. However, storing the entire graph is stated to be "infeasible" [16]. Vector-clock-based techniques [33][34] process each dependency and merge the timestamps of the dependency into the vector clocks. They can only report the dependency that completes the cyclic dependency sequence when detecting an atomicity violation. However, the sequence reported by these techniques based on the timestamps in the VCs cannot distinguish whether or not it is a cyclic sequence involving two transactions only. As a result, the reported sequences may not be an actual sequence of transactions that forms the cyclic sequence and leads to the detected atomicity violation in the trace.

In this paper, we propose ***Davida***, a novel, sound, and online atomicity checker to address the above challenges. To the best of our knowledge, Davida is the *first* work to precisely maintain a reduced set of *increasing* dependency sequences for each active transaction sufficient for detecting all atomicity violations with witnesses. It is built on top of our two insights. (Insight I1) A dependency sequence θ can become a witness for an atomicity violation on a transaction *tx* only if a shorter prefix of θ is formed before a longer one; otherwise, θ will not be increasing with respect to *tx*. Moreover, suppose θ is an *increasing* dependency sequence starting from *tx* and τ is a dependency. In this case, if $\theta^\frown\langle\tau\rangle$ is non-increasing, then τ should be irrelevant to any possible atomicity violation on *tx*. As such, Davida can safely ignore these dependencies without losing the precision. (Insight I2) An atomicity violation on *tx* triggered by its event e_b occurs only when *tx* generates e_b. Thus, *tx* should be an active transaction when generating e_b. So, Davida can limit its efforts to only track these increasing dependency sequences reachable from active transactions of each thread.

Davida maintains a transactional happens-before tree (HBT) instance for each thread (for its active transaction). The set of HBT instances forms a forest. The root of each HBT instance is the active transaction of the corresponding thread. The HBT instance is dropped when the status of the transaction at the root changes to completed. Each parent-child edge in an HBT instance is a dependency. Any path starting from the root in the HBT instance represents an increasing dependency sequence. Davida ensures, by theorems, that the HBT instance of each thread captures a set of increasing dependency sequences where the root can reach each of them. As such, if a dependency sequence is increasing and non-increasing with respect to two active transactions, respectively, then it is captured in the HBT instance for the former transaction but not for the latter one.

Davida is a novel decentralized graph tracking approach. Suppose *u* and *v* are two threads and their HBT instances are HBT_u and HBT_v, respectively. Let the root transaction of HBT_v be δ_v. Suppose $\tau = tx_i \rightsquigarrow tx_j$ is a dependency, where transactions tx_i and tx_j are executed by threads *u* and *v*, respectively. Davida works as follows:

- HBT_v checks whether tx_i is reachable from its root. If this is the case, it reports an atomicity violation on tx_j (= δ_v) with the sequence of edges from the root δ_v to tx_i appended with τ as the witness.
- Each other HBT instance, including HBT_u and other HBT instances, makes its own decision. It will append τ to its tree only if tx_i is reachable from its root but tx_j is not reachable from its root.
- Each HBT instance for a completed transaction is dropped (to conserve memory and maintenance effort).

We have evaluated Davida on 15 programs in the large DaCapo, small Java Grande and microbenchmark benchmark suites that incur atomicity violations. The results show that Davida precisely detected all atomicity violations with witnesses. It precisely kept 7.19x fewer dependencies and searched 26.39x fewer transactions than Velodrome to complete the witness checking, not to mention to achieve a larger coverage on atomicity violations. It incurred 1.23x and 1.16x less memory and runtime overheads than Velodrome. It also confirms that Davida and RegionTrack detected the same set of atomicity violations, but RegionTrack could not locate any witness but with better runtime performance. Apart from using Davida as a standalone technique, it can work with RegionTrack. For instance, developers may use RegionTrack for pure atomicity violation detection. Once an atomicity violation is detected, Davida can be used to report the witness, which is not possible by using all other existing techniques.

This paper makes two main contributions: (1) It presents the first work to show the feasibility of precise and efficient tracking of all atomicity violations with witnesses in a decentralized approach. (2) It proposes a novel technique Davida to realize the approach.

The rest of this paper is as follows: Sections II to IV present preliminaries and motivations, Davida and the evaluation of Davida, respectively. Section V reviews the related work. Section VI concludes this work.

II. Preliminaries and Motivations

A. Scenario

Fig. 1 shows an exemplified program p_1. In the program p_1, threads t_1 to t_4 execute atomic regions m_1, m_2 followed by m_6,

```
Thread t1      Thread t2      Thread t3      Thread t4

m1{            m2{            m3{            m4{
  p = 1          x = 1          …              s = 0
  q = 0          z = 1        }                x = s
}              }                               z = s
                                             }

               m6{            m5{
                 y = 0          t = y + z
                 if p = 0       p = t + 1
                 …              q = t + 2
               }              }
```

Fig. 1. An exemplified program p_1

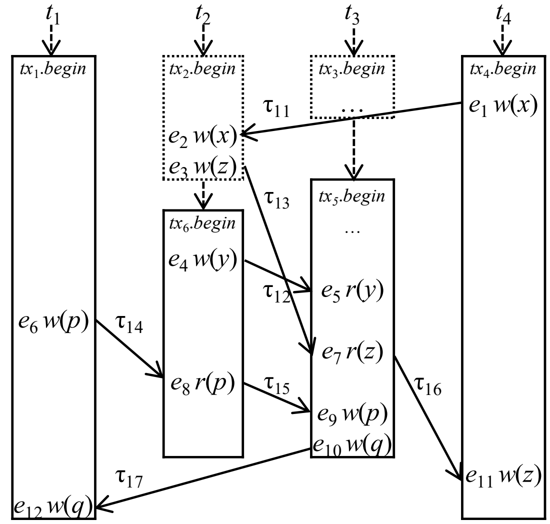

Fig. 2. Non-serializable trace α_1 of p_1 with AV on tx_1 and tx_4

m_3 followed by m_5, and m_4, respectively.

B. Preliminaries and Illustrations

Each event has an operation type: (1) $r(t, x)$ and $w(t, x)$: reading a value from variable x and writing a value to variable x by thread t, respectively. (2) $acq(t, m)$ and $rel(t, m)$: acquiring and releasing a lock m by thread t, respectively. (3) *begin* and *end*: marking the begin and end of an atomic region executed by a thread. A trace α is a sequence of events: $\alpha = \langle e_1, e_2, \ldots, e_i, \ldots, e_n\rangle$. The trace α is also well-formed [34]: all lock acquire and release events are well-matched, a lock cannot be acquired by more than one thread at any time, and all begin and end events are well-matched.

Illustration. Fig. 2 shows an execution trace $\alpha_1 = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\rangle$ of the program p_1 in Fig. 1.

A **(regular) transaction** $tx = \langle tx.begin, \ldots, e_x, \ldots, tx.end\rangle$ denotes the sequence of events executed by a thread t in between a matching pair of *begin* and *end* events. A **unary transaction** is formed by an event e itself. Each thread t executes a sequence of transactions $\langle tx_1, \ldots, tx_i, tx_{i+1}, \ldots, tx_n\rangle$ one by one. Each transaction has a timestamp, and the timestamp of tx_i is smaller than that of tx_{i+1} by 1. We denote the thread executing an event e and a transaction tx by $\boldsymbol{T(e)}$ and $\boldsymbol{T(tx)}$, respectively. A containment relation is denoted by $e_x \in tx$. For two transactions tx_i and tx_j of the same thread t, if tx_i is executed before tx_j, then tx_i and tx_j follow the ***program order***.

Illustration. In α_1, threads t_1 to t_4 execute transactions tx_1, tx_2 followed by tx_6, tx_3 followed by tx_5, and tx_4, respectively. Transactions tx_2 and tx_6, and tx_3 and tx_5 follow the program order. Transactions tx_1 to tx_6 are execution instances of atomic regions m_1 to m_6. To simplify the presentation, we only illustrate the accesses on shared variables in α_1.

In a prefix of a trace, a transaction tx is **active** if the prefix does not include event *tx.end*. It models the transaction currently executing in the execution at the moment represented by the prefix. A transaction is **completed** if event *tx.end* is included in the prefix. Our basic model is that each thread t has at most one active transaction in a prefix, denoted by $\boldsymbol{\delta_t}$.

Illustration. We assume that right after event e_3 has been executed, transaction tx_2 is completed, and tx_3 is completed before the execution of e_3. After executing α_1's prefix $\langle e_1, e_2, e_3, e_4, e_5, e_6\rangle$, transactions tx_1, tx_4, tx_5 and tx_6 are still active.

For two events e and e' in a trace α, e and e' ***conflict*** [16][3] if any condition below holds: (1) $T(e) = T(e')$. (2) e and e' acquire or release the same lock. (3) e and e' access the same variable, and at least one of them is a write event.

If e appears before e', e conflicts with e' and there is ***no*** conflicting event e'' in between them, acquiring the same lock, or accessing the same variable and conflicting with both of them, then e and e' form a ***happens-before* (*HB*) *dependency***, denoted as $e \rightarrowtail e'$. HB relation (e.g., e happens before e') is the transitive closure of HB dependencies in α. Each HB dependency $e_i \rightarrowtail e_j$ corresponds to a ***dependency*** $tx_i \rightsquigarrow tx_j$ at the transaction level where tx_i and tx_j are transactions, $e_i \in tx_i$, $e_j \in tx_j$ and $T(tx_i) \neq T(tx_j)$. $tx_i \rightsquigarrow tx_j$ can be formed by different HB dependencies.

Illustration. In trace α_1 of Fig. 2, events e_1, e_3, e_4, e_6, e_7, e_8 and e_{10} conflict with events e_2, e_7, e_5, e_8, e_{11}, e_9 and e_{12}, respectively, which produces seven cross-thread happens-before dependencies (i.e., $e_1 \rightarrowtail e_2$, $e_3 \rightarrowtail e_7$, $e_4 \rightarrowtail e_5$, $e_6 \rightarrowtail e_8$, $e_7 \rightarrowtail e_{11}$, $e_8 \rightarrowtail e_9$, and $e_{10} \rightarrowtail e_{12}$). These variable-level dependencies correspond to seven cross-thread dependencies at the transaction-level: $\tau_{11} = tx_4 \rightsquigarrow tx_2$, $\tau_{12} = tx_6 \rightsquigarrow tx_5$, $\tau_{13} = tx_2 \rightsquigarrow tx_5$, $\tau_{14} = tx_1 \rightsquigarrow tx_6$, $\tau_{15} = tx_6 \rightsquigarrow tx_5$, $\tau_{16} = tx_5 \rightsquigarrow tx_4$ and $\tau_{17} = tx_5 \rightsquigarrow tx_1$. Note that in online detection techniques, a dependency is generated when the event at the tail position of the dependency (e.g., event e_{12} of transaction tx_1 for dependency τ_{17}) is executed.

A ***dependency sequence*** $\theta = tx_1 \rightsquigarrow tx_n$ is a non-empty sequence of dependencies at the transaction level and is formed by an underlying event sequence S. For all i, $\theta[i]$ refers to the i-th dependency in θ. $\theta[i]$.*source* and $\theta[i]$.*sink* refer to the transactions at the head and tail positions of the dependency $\theta[i] = tx_{source} \rightsquigarrow tx_{sink}$, respectively. For each sequence θ, $\theta[i]$.*sink* and $\theta[i+1]$.*source* should be executed by the same thread. The sequence θ is ***increasing***, denoted as $tx_1 \rightsquigarrow^+ tx_n$, if, for all $j < k$,

TABLE I. Existing Trace-based Techniques Classification

		Hybrid	Vector Clock-based		Centralized Graph-based	
Atomicity Violation		Davida	RegionTrack	AeroDrome	DoubleChecker	Velodrome
Trace-level	***No false positive***	✓	✓	✓	✓	✓
	No false negative	✓	✓	✓	✓	✓
Transaction-level	***No false positive***	✓	✓	Not supported	×	✓
	No false negative	✓	✓	Not supported	✓	×
	witness	✓	Not supported	Not supported	✓	✓

the event $S[j]$ either happens before the event $S[k]$ or is equal to that event. If θ contains exactly one dependency, it is *increasing*. θ is ***non-increasing*** if it is not increasing. Each dependency θ[i] as well as θ[i].*source* and θ[i].*sink* along the sequence θ is ***reachable*** from tx_1 in the sequence.

A ***cyclic dependency sequence*** $tx_1 \rightsquigarrow^c tx_i$ is an increasing dependency sequence $tx_1 \rightsquigarrow^+ tx_i$ appended with $tx_i \rightsquigarrow tx_1$.

When a cross-thread dependency $tx_i \rightsquigarrow tx_1$ is generated by the HB dependency $e_x \rightarrowtail e_m$, where $e_x \in tx_i$ and $e_m \in tx_1$, an ***atomicity violation*** (or **AV** for short) [16][33] on transaction tx_1 is triggered at event e_m if the cyclic dependency sequence $tx_1 \rightsquigarrow^c tx_i$ is formed. We refer to $tx_1 \rightsquigarrow^c tx_i$ as a ***witness*** for the atomicity violation on transaction tx_1. Note that one or more atomicity violations on tx_1 can be triggered by multiple events of tx_1 and $tx_1 \rightsquigarrow^+ tx_i$ can be formed via one or more dependency sequences. Thus, one atomicity violation on tx_1 may have one or multiple witnesses.

Illustration. In Fig. 2, there are five cycles in the graph: $c_1 = \langle\tau_{11}, \tau_{12}, \tau_{16}\rangle$, $c_2 = \langle\tau_{11}, \tau_{13}, \tau_{16}\rangle$, $c_3 = \langle\tau_{11}, \tau_{15}, \tau_{16}\rangle$, $c_4 = \langle\tau_{14}, \tau_{12}, \tau_{17}\rangle$ and $c_5 = \langle\tau_{14}, \tau_{15}, \tau_{17}\rangle$.

The event sequence s_1 underlying cycle c_1 is $\langle e_1, e_2, e_3, e_4, e_5, e_7, e_{11}\rangle$, in which the event $s_1[i]$ happens before the event $s_1[j]$ for all $i < j$. Observe that cycle c_1 is an *increasing dependency sequence*. The first and the last transactions of c_1 are both tx_4. So, event e_{11} triggers an *atomicity violation* on transaction tx_4. Cycle c_1 is a *witness* for the atomicity violation on tx_4. Specifically, in Fig. 1, witness c_1 indicates the execution instances of the atomic regions in the sequence $\langle m_4@t_4, m_2@t_2, m_6@t_2, m_5@t_3, m_4@t_4\rangle$ are inter-dependent to trigger the atomicity violation on the execution instance of m_4.

Cycle c_2 is another *witness* for the atomicity violation on tx_4. It represents $\langle m_4@t_4, m_2@t_2, m_5@t_3, m_4@t_4\rangle$.

Cycle c_5 is also increasing and is a witness for the atomicity violation on transaction tx_1, indicating $\langle m_1@t_1, m_6@t_2, m_5@t_3, m_1@t_1\rangle$.

Nevertheless, cycles c_3 and c_4 are *non-increasing* because e_9 and e_8 do not happen before e_7 and e_4, respectively. (The dependency sequences $\langle\tau_{15}, \tau_{16}\rangle$ and $\langle\tau_{14}, \tau_{12}\rangle$ are non-increasing.) So, c_3 and c_4 do not indicate atomicity violations on tx_4 or tx_1 at the transaction level.

RegionTrack [33] utilizes vector clocks to propagate the timestamps of transactions to capture the happens-before dependencies at the event, subregion and transaction levels. It is both sound and complete in detecting atomicity violations at the transaction-level and trace-level. RegionTrack [33] also formulates the precise checking of $tx.begin \rightarrowtail e_x$.

C. Motivations

As shown in TABLE I, existing techniques are classified into two categories: centralized-graph-based techniques and vector-clock-based techniques. All the existing techniques detect the non-serializable traces in a sound and complete manner. On the other hand, these centralized-graph-based techniques (DoubleChecker [3] and Velodrome [16]) only imprecisely or incompletely detect atomicity violations and report witnesses for detected atomicity violations at the transaction level. The vector-clock-based techniques (AeroDrome [34] and RegionTrack [33]) cannot detect any witnesses.

To analyze α_1, Velodrome and DoubleChecker construct a graph as shown in Fig. 2 **but** with τ_{15} removed, because they only keep one dependency between the same two transactions. Specifically, on processing a dependency $\tau = tx_i \rightsquigarrow tx_j$, if the graph already contains a dependency from tx_i to tx_j, Velodrome and DoubleChecker will not add τ to the graph. Moreover, to recover the temporal ordering information between dependencies for the same thread, when adding a dependency to the graph, Velodrome further adds additional timestamps at the head and tail positions of the dependency (e.g., $\tau_{11} = (tx_4, 1) \rightsquigarrow (tx_2, 2)$). (Ref [16] states that "a trace may contain millions of transactions, and storing the entire happens-before graph would be infeasible", signifying the challenge of keeping a right set of edges for the detection of every possible atomicity violation using a centralized-graph-based approach.)

On processing τ_{16} in α_1, Velodrome and DoubleChecker both find tx_4 in the graph already containing a dependency from tx_4 to some transactions. Starting from τ_{11} in the graph, they search the dependency sequence c_2 in the graph. Since DoubleChecker detects cycle c_2, it reports tx_4 that completes the cycle incurring an atomicity violation and c_2 being a witness. Velodrome further determines whether c_2 is increasing. It checks whether the timestamp at the tail position of τ_{11} is smaller than that at the head position of τ_{13}. Velodrome also reports tx_4 incurring an atomicity violation and c_2 being a witness.

Then, on processing τ_{17}, Velodrome and DoubleChecker find the graph containing a dependency from tx_1 to some other transaction. Starting from τ_{14} in the graph, they locate the dependency sequence c_4. As DoubleChecker detects cycle c_4, it directly reports tx_1 that completes the cycle incurring an atomicity violation and c_4 being a witness. However, since c_4 is non-increas-

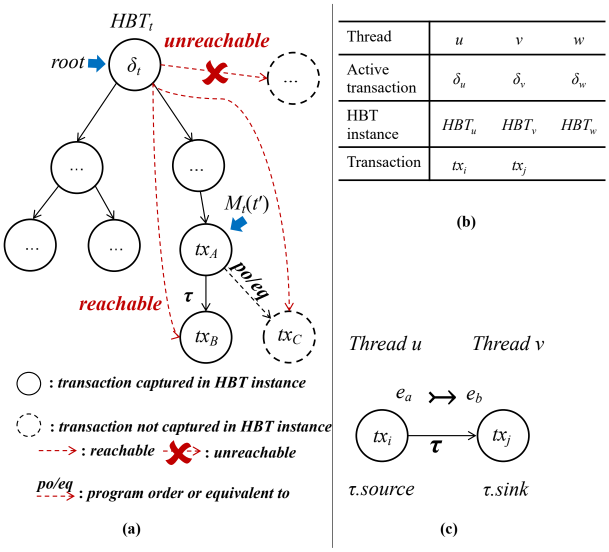

Fig. 3. Notations

ing, it does not indicate an atomicity violation on tx_1. Velodrome finds c_4 non-increasing. So, it reports neither tx_1 incurring an atomicity violation nor c_4 being a witness.

We can see that τ_{12} is increasing with respect to tx_4 (via τ_{11}) but non-increasing with respect to tx_1 (via τ_{14}). Determining τ_{12} as either increasing or non-increasing in a path-insensitive manner is infeasible. (Similarly, with respect to tx_4, τ_{16} is increasing following one path (via τ_{12}) but non-increasing following another path (via τ_{15})). In general, a dependency can be reachable and increasing with respect to one transaction and reachable but non-increasing with respect to another transaction at the same time. Giving up labeling the dependency sequence as increasing or non-increasing will falsely report some atomicity violations with witnesses. Labeling a dependency sequence as non-increasing cannot be done before exhausting all possible paths that can reach it using the existing approach, while labeling some but not all dependency sequences as increasing will miss reporting some atomicity violations.

The difficulty in labeling a dependency sequence as increasing makes Velodrome incur false negatives and DoubleChecker incur false positives in atomicity violation and witness detection. RegionTrack [33] and AeroDrome [34] are vector-clock-based techniques. AeroDrome tracks dependencies but not dependency sequences, and thus cannot detect any atomicity violation. RegionTrack precisely detects all atomicity violations but cannot report any witness for any atomicity violation, because it tracks the latest transaction timestamp of other threads that the current thread can see and discards all dependencies right after its online processing. Specifically, for α_1, AeroDrome cannot report any of tx_1 and tx_4 incurring atomicity violations. RegionTrack detects the atomicity violations on both tx_1 and tx_4 but cannot report any witnesses.

Besides, when debugging an atomicity violation, it is desirable to inform developers with a witness for each detected atomicity violation to diagnose how the transactions are inter-dependent to trigger the violation. Without being informed with witnesses, it is challenging for developers to precisely reason how these transactions are interleaved to trigger the atomicity violations for program debugging and fixing.

Developers need to manually inspect the code of all the atomic regions to understand the causes of the atomicity violations. For instance, as shown in Fig. 1, if developers are only provided with the execution instance of atomic region m_1, developers must inspect all the 6 atomic regions for the atomicity violation(s) on m_1.

On the other hand, as illustrated in Fig. 2, the atomicity violation on tx_1 only involves tx_6 and tx_5. Thus, without this information, developers need to inspect 3 more atomic regions without knowing witness c_4. In modern multithreaded programs, it is easy to have hundreds of atomic regions. If developers are only informed the execution instance of an atomic region incurs atomicity violations, it is time-consuming and tedious for them to find and understand, among all atomic regions in the source code, which atomic regions are involved in triggering the atomicity violations.

Moreover, Velodrome utilizes a depth-first graph search approach to report witness for the detected atomicity violation. On processing τ_{16} in α_1, after locating dependency τ_{11}, it finds that tx_2 involves in some dependency with transactions of thread t_3, and then locates τ_{13}. In the end, Velodrome locates cycle c_2. Since c_2 is increasing, it reports tx_4 incurring an atomicity violation and c_2 being the witness for the violation.

However, as illustrated in Fig. 2, cycles c_1 and c_2 are both witnesses for the atomicity violation on tx_4. Each of them represents a scenario of how an atomicity violation on the region m_4 is triggered. However, on the one hand, witness c_2 includes execution instances of atomic regions m_4, m_6 and m_5, and witness c_1 includes execution instances of m_4, m_2, m_6 and m_5. All execution instances in witness c_2 are included in witness c_1. On the other hand, developers need to manually inspect the code of the atomic code regions and construct the dependencies between the regions given by the detected witness. Then, if further retrieving all dependencies among all transactions in witness c_1, these dependencies can also construct witness c_2, but not vice versa. In other words, witness c_2 is completely embedded in witness c_1. Also, if only given witness c_2, developers may only eliminate the dependency τ_{13} between transactions tx_2 and tx_5 to avoid the atomicity violation on tx_4, but the violation on tx_4 may still be triggered and witnessed by c_1. If only given witness c_1, between the threads t_2 and t_3, developers could construct and eliminate both dependencies τ_{12} and τ_{13} to avoid the atomicity violation on tx_4, which is more desirable and precise. A limitation of Velodrome is that Velodrome cannot know the existence of the other witnesses (say c_1) before it exhausts all paths in the graph. It can only detect c_1 when it enumerates all cycles that start and end at tx_4 followed by determining whether c_1 is more comprehensive than some other cycles (say c_2). Consequently, the witness reported by Velodrome can only provide a partial

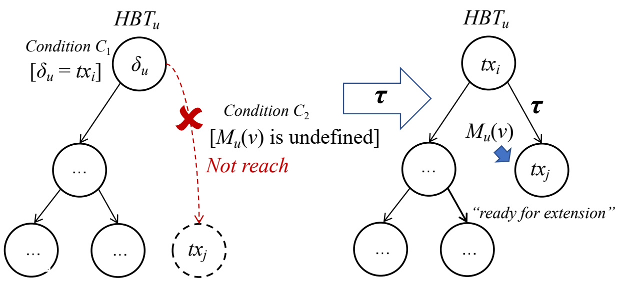

Fig. 4. Maintenance of HBT_u

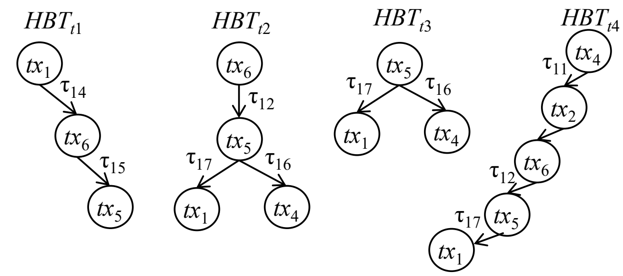

Fig. 5. Forest maintained by David in α_1 when e_{11} executes

view when debugging an atomicity violation.

Since RegionTrack discards all dependencies after its processing, it can only report the dependencies τ_{16} and τ_{17} when detecting the atomicity violations on tx_4 and tx_1, respectively. In other words, RegionTrack cannot detect any witnesses (e.g., c_1, c_2 and c_5) for the detected atomicity violations.

III. Our Approach: Davida

This section presents Davida. Davida is built on an underlying online framework that generates dependencies. (Davida adopts RegionTrack [33] for this purpose in the experiment.) Owing to the complexity of our model, Fig. 3 summarizes the major notations we will use in this section to present Davida.

A. Transactional Happens-Before Tree (HBT)

A ***transactional happens-before tree (HBT)*** instance HBT_t for thread t is a rooted and directed tree $\langle V, E\rangle$, where V is a set of transactions and $E \subseteq V \times V$ is a set of dependencies in a trace. We simply refer to the **root** of HBT_t as the active transaction $\boldsymbol{\delta_t}$ of thread t at the moment of HBT assigning with a new root node. Each path from the root δ_t to a node tx_B via an edge $\tau = tx_A \rightsquigarrow tx_B$ of the tree represents an increasing dependency sequence in the trace (i.e., $\delta_t \rightsquigarrow^+ \tau$).

We simply say that δ_t **reaches** tx_B and τ, or $\boldsymbol{tx_B \in HBT_t}$ and $\boldsymbol{\tau \in HBT_t}$, respectively, for short. (Note that both tx_B and τ are contained in HBT_t). On the contrary, if a transaction tx_C is not a node in $\boldsymbol{HBT_t}$, we say that both HBT_t and δ_t **do not reach** tx_C, denoted by $\boldsymbol{tx_C \notin HBT_t}$.

Suppose $tx_A \in HBT_t$ and $tx_C \notin HBT_t$ are two transactions such that tx_A happens before tx_C by the program order. In this case, we have: δ_t happens before tx_A and tx_A happens before tx_C, indicating that δ_t happens before tx_C. We say that tx_C is **reachable** from HBT_t, denoted by $\boldsymbol{HBT_t \rhd tx_C}$. Similarly, if $tx_C \notin HBT_t$ and there is no such $tx_A \in HBT_t$ such that tx_A happens before tx_C, tx_C is said to be **unreachable** from both HBT_t and δ_t, denoted by $\boldsymbol{HBT_t \ntriangleright tx_C}$.

To simplify our subsequent presentation further, for brevity, if $tx_A \in HBT_t$, we also say tx_A is reachable from HBT_t. Moreover, in case we need to be specific to tx_A containing in HBT_t, we will use the notation $tx_A \in HBT_t$ and state it as δ_t reaching tx_A.

Davida maintains a transactional happens-before forest $F = \{\langle HBT_t, M_t\rangle \mid t \in Tid\}$, where Tid represents the set of threads in the trace. At any moment, for each thread t, it maintains an HBT instance HBT_t and a hash map M_t. Since in our model, each transaction has a unique index and the timestamps of the transactions executed by the same thread are continuous numbers, we design M_t as a variant of vector clock [26] (where a vector clock keeps the transaction timestamp of each thread). Specifically, the entry $M_t(t)$ maps to the root δ_t. Also, the entry $M_t(t')$ for each other thread t' (where $t' \neq t$) maps to the transaction having the ***smallest*** timestamp among the transactions executed by t' (where $t' \neq t$) captured in HBT_t. (Note that, by definition, $M_t(t') \in HBT_t$.)

If the entry $M_t(t')$ is not assigned with any transaction, it indicates that no transaction of thread t' is reached from the root δ_t yet. We simply refer to the entry $M_t(t')$ with an assigned value as **defined**, otherwise **undefined**.

To efferently determine whether or not a transaction tx_C of thread t' is reachable from δ_t as illustrated in Fig. 3(a), Davida checks whether $M_t(t')$ is defined or not. This is because, by definition, $M_t(t') \in HBT_t$ and we have either $M_t(t') = tx_C$ or $M_t(t')$ happening before tx_C by the program order of thread t'. (Note that the case of tx_C happening before $M_t(t')$ by the program order is infeasible because, by definition, $M_t(t')$ should have kept a transaction with the smallest timestamp, producing a contradiction that tx_C happens before $M_t(t')$.) On the other hand, if $M_t(t')$ is undefined, it indicates that δ_t is still unable to reach any transaction of t' yet, and so, tx_C is unreachable from δ_t.

We further revisit the notations shown in Fig. 3(b) that are used in the following presentation. We denote three threads and their HBT instances as u, v, w (where $u \neq v \neq w$), and HBT_u, HBT_v, and HBT_w, respectively. The roots of these HBT instances are denoted as the active transactions δ_u, δ_v, and δ_w of threads u, v and w. Threads u and v execute tx_i and tx_j, respectively. On processing an event $e_b \in tx_j$, a cross-thread dependency $\tau = tx_i \rightsquigarrow tx_j$ (denoted as τ) triggered by the happens-before dependency $e_a \rightarrowtail e_b$ is generated, as depicted in Fig. 3(c), where event $e_a \in tx_i$ appears earlier than e_b in the trace. For ease of presentation, we refer to tx_i and tx_j as the **source** and **sink** transactions of τ, denoted as $\tau.source$ and $\tau.sink$ (i.e., $\tau.source = tx_i$ and $\tau.sink = tx_j$), respectively.

Subsections B and C present how Davida tracks increasing dependency sequences and detects atomicity violations with witnesses, respectively.

B. Maintaining Increasing Dependency Sequences

1) Illustration: We first illustrate how Davida handles trace α_1 in Fig. 2 to maintain the increasing dependency sequences from each transaction while it is active. Davida processes dependency sequence $\langle \tau_{11} = tx_4 \rightsquigarrow tx_2, \tau_{12} = tx_6 \rightsquigarrow tx_5, \tau_{13} = tx_2 \rightsquigarrow tx_5, \tau_{14} = tx_1 \rightsquigarrow tx_6, \tau_{15} = tx_6 \rightsquigarrow tx_5, \tau_{16} = tx_5 \rightsquigarrow tx_4$ and $\tau_{17} = tx_5 \rightsquigarrow tx_1\rangle$ in turn. On processing the beginning event of tx_4, the HBT instance HBT_{t4} is set to empty. The root δ_{t4} of HBT_{t4} is set to tx_4. On processing τ_{11}, Davida adds $\tau_{11}.sink$ (i.e., tx_2) to HBT_{t4} as a child of δ_{t4} and updates $M_{t4}(t_2)$ to tx_2. The parent-child edge from δ_{t4} to tx_2 represents τ_{11}. Next, Davida processes τ_{12}. At this moment, $M_{t4}(t_2)$ is mapped to tx_2 and $M_{t4}(t_3)$ is undefined. Since $\tau_{12}.source$ (i.e., tx_6) follows tx_2 by the program order and $\langle\tau_{11}, \tau_{12}\rangle$ is increasing, Davida adds tx_6 to HBT_{t4} as a child of tx_2 and adds tx_5 to HBT_{t4} as a child of tx_6. Davida also updates $M_{t4}(t_3)$ to tx_5. Then, on processing τ_{13}, since $M_{t4}(t_3)$ is mapped to tx_5 (and infers that tx_3 follows tx_5 by the program order as well) and $tx_5 \in HBT_{t4}$, Davida does not capture τ_{13} into HBT_{t4}. On processing τ_{14}, since $M_{t4}(t_1)$ is undefined, so, $\tau_{14}.source \notin HBT_{t4}$, indicating τ_{14} is irrelevant to any possible atomicity violation that might occur on δ_{t4}. Thus, Davida does not capture τ_{14} into HBT_{t4}. Then, on processing τ_{15}, because $M_{t4}(t_3)$ is already mapped to tx_5, indicating that HBT_{t4} reaches $\tau_{15}.sink$ (i.e., tx_5) already. So, Davida does not capture τ_{15} into HBT_{t4}. When handling τ_{16}, Davida does not capture τ_{16} into HBT_{t4} because $\tau_{16}.sink$ is δ_{t4}. On processing τ_{17}, Davida captures τ_{17} into HBT_{t4} by appending it after τ_{12}.

Similarly, Davida captures dependencies τ_{14} and τ_{15} into HBT_{t1}. However, during the executions of tx_2 and tx_3, no dependency starting from each of tx_2 and tx_3 is generated. Besides, when τ_{12} is generated, tx_2 has been completed. Thus, Davida captures no dependency into both HBT_{t2} and HBT_{t3} during the executions of tx_2 and tx_3. After processing τ_{11}, two transactions tx_6 and tx_5 start. Davida clears HBT_{t2} and HBT_{t3}. Then, δ_{t2} and δ_{t3} are updated to tx_6 and tx_5, respectively. Davida captures dependencies τ_{12}, τ_{16} and τ_{17} in HBT_{t2}. Similarly, Davida also captures dependencies τ_{16} and τ_{17} in HBT_{t3}. Fig. 5 shows the forest maintained by Davida in α_1 when e_{11} executes.

2) Our Design: Algorithm 1 shows the main algorithm of Davida. It consists of three components: *initialization*, *maintenance of* HBT_u, and *maintenance of every* HBT_w where $w \neq u$ or v when processing dependency $\tau = tx_i \rightsquigarrow tx_j$.

Initialization: Whenever a thread, say t, starts an active transaction δ_t, Davida calls *initialization* at line 2. Line 3 clears the HBT instance HBT_t because the preceding transaction should have been completed before the current transaction starts. (Also, any atomicity violations and witnesses on the preceding completed transaction have been reported.) Then, it assigns δ_t as the root of HBT_t at line 4. It clears the map M_t and sets $M_t(t)$ to δ_t at lines 5-6.

Maintenance of HBT_u*:* When a dependency $\tau = tx_i \rightsquigarrow tx_j$ is generated at the moment of an event $e_b \in tx_j$ being generated, the HBT instance HBT_u of thread u checks whether τ should be added to HBT_u at lines 9-12. HBT_u first checks whether the following two conditions hold at line 9: (Condition C1) tx_i is the active transaction δ_u of thread u; and (Condition C2) HBT_u does not reach any transaction of thread v, which is checked by examining whether $M_u(v)$ is defined. As shown in Fig. 4, if both conditions hold, HBT_u adds tx_j as a child of tx_i at line 10 to capture τ. It also updates $M_u(v)$ to tx_j at line 11 to indicate that tx_j is the transaction of thread v with the smallest timestamp kept in HBT_u. This entry $M_u(v)$ indicates that τ is "ready for extension" to form a longer increasingly dependency sequence with respect to the root.

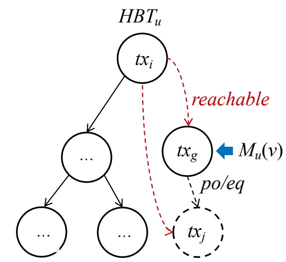

Fig. 6. Skip case of capturing dependency τ for HBT_u

■ The two conditions (C1) and (C2) are specially designed to eliminate cases where Davida can skip the dependency maintenance of HBT_u without compromising the soundness guarantee. Recall that the state of a transaction at any moment can only be active or completed but not both. There are four possible combinations of the statuses of the two transactions tx_i and tx_j. Suppose that tx_j is a completed transaction. In this case, it is impossible for tx_j to generate any new events that produces a new dependency. They are infeasible cases.

■ Therefore, generating a dependency τ is feasible only when tx_j is active. The status of tx_i at the moment of the event $e_b \in tx_j$ being generated can be active or completed, however. The two cases are handled below. (1) Suppose that tx_i is a completed transaction. In this case, tx_i cannot further generate new events, and thus no new atomicity violation on tx_i can be further triggered. Since any previous atomicity violations with witnesses on tx_i have been detected by Davida before processing τ, this dependency τ can be safely skipped by HBT_u. This case is represented precisely by the negation of condition (C1). (2) Suppose that tx_i is the active transaction of thread u at this moment (i.e., the root of HBT_u is tx_i). Without loss of generality, suppose further that $M_u(v)$ maps to a transaction tx_g of thread v. (Note that the thread executing tx_j is v as well.) In this case, we have $tx_g \in HBT_u$, as illustrated in Fig. 6. There are two sub-cases to consider. First, suppose $tx_g \neq tx_j$. In this case, tx_g happens before tx_j (i.e., tx_j is already reachable from HBT_u). Second, suppose $tx_g = tx_j$. In this case, tx_j should already be contained in HBT_u. So, in either case, there is no need to further maintain HBT_u to capture τ. This is precisely represented by the negation of Condition (C2).

■ As a result, only when both conditions (C1) and (C2) are satisfied at line 9, Davida proceeds to capture τ into HBT_u by adding tx_j as a child of tx_i at line 10.

Maintenance of every HBT_w *where* $w \neq u$ *or* v*:* We design HBT in the way that each increasing dependency sequence in an HBT instance is extended *incrementally* by appending additional dependencies one by one. For instance, in Fig. 2, on processing dependency τ_{15}, τ_{15} is appended to τ_{14} that is already kept in HBT_{t1}. To do so, Algorithm 1 calls *checkOtherTrees* at lines 13-14. The function *checkOtherTrees* extends the increasing

TABLE II. Case Analysis of Capturing Dependency τ into HBT_w

		DS_1	DS_2	DS_1	DS_2	DS_1	DS_2	DS_1	DS_2
$M_w(u)$	$M_w(v)$	✓	✓	✓	×	×	✓	×	×
defined	defined	(-) ①		(×) ②		(×) ③		(×) ④	
defined	undefined	(×) ⑤		(+) or (-) ⑥		(×) ⑦		(×) ⑧	
undefined	defined	(×) ⑨		(×) ⑩		(-) ⑪		(×) ⑫	
undefined	undefined	(×) ⑬		(×) ⑭		(×) ⑮		(-) ⑯	

Case ① to Case ⑯ (-): skip (+): capture (×): infeasible
$DS_1 = HBT_w \rhd tx_i$ $DS_2 = HBT_w \rhd tx_j$

dependency sequences maintained in HBT_w at lines 16-31.

TABLE II depicts the complete case analysis of whether a dependency $\tau = tx_i \rightsquigarrow tx_j$ should be captured into HBT_w, corresponding to the updates in Fig. 7. Specifically, Davida considers four conditions:

- Whether the root δ_w reaches any transaction of thread u (i.e., is $M_w(u)$ defined?)
- Whether the root δ_w reaches any transaction of thread v (i.e., is $M_w(v)$ defined?)
- Whether tx_i is reachable from δ_w (i.e., is $HBT_w \rhd tx_i$?)
- Whether tx_j is reachable from δ_w (i.e., is $HBT_w \rhd tx_j$?)

In total, 16 cases, from Case ① to Case ⑯, are considered.

■ In TABLE II, there are 12 ***infeasible*** cases as shown in Fig. 7(a)–(e): ②③④⑤⑦⑧⑨⑩⑫⑬⑭⑮. Each such case satisfies at least one pair of self-contradicting conditions: (1) $M_w(u)$ is defined and $HBT_w \ntriangleright tx_i$, (2) $M_w(u)$ is undefined and $HBT_w \rhd tx_i$, (3) $M_w(v)$ is defined and $HBTw \ntriangleright tx_j$, (4) $M_w(u)$ is undefined and $HBT_w \rhd tx_j$. Each of cases (1) and (3) is self-contradictory because when $M_w(u)$ and $M_w(v)$ are defined, we must have $M_w(u) \in HBT_w$ and $M_w(v) \in HBT_w$, respectively. Since $M_w(u)$ and $M_w(v)$ must have timestamps not larger than tx_i and tx_j, respectively, so, tx_i and tx_j must be reachable from δ_w (i.e., $HBT_w \rhd tx_i$ and $HBT_w \rhd tx_j$ must be held), respectively, contradicting to $HBT_w \ntriangleright tx_i$ and $HBT_w \ntriangleright tx_j$, respectively. Each of cases (2) and (4) is self-contradictory because when $M_w(u)$ and $M_w(v)$ are undefined, δ_w could not reach any transaction of threads u and v, respectively. It is infeasible for tx_i and tx_j reachable from δ_w, contradicting to the given conditions of $HBT_w \rhd tx_i$ and $HBT_w \rhd tx_j$, respectively.

■ In TABLE II, there are three cases to ***skip*** $\tau = tx_i \rightsquigarrow tx_j$ from being captured into HBT_w: ①⑪⑯. In cases ①⑪, the condition $HBT_w \rhd tx_j$ holds already. Specifically, $M_w(v)$ is defined and tx_j is reachable from HBT_w. Note that $M_w(u)$ can be defined (i.e., ①) or undefined (i.e., ⑪). There are two sub-cases to consider. First, suppose that $M_w(v)$ maps to a transaction tx_z of thread v. In this case, we should have $tx_z \in HBT_w$ and tx_z happens before tx_j due to the program order of thread v. Thus, the dependence sequence $\delta_w \rightsquigarrow^+ tx_z$ is captured in HBT_w and $\delta_w \rightsquigarrow^+ tx_j$ is formed by appending $tx_z \rightsquigarrow^+ tx_j$ to $\delta_w \rightsquigarrow^+ tx_z$. Second, suppose that $M_w(v)$ maps to tx_j such that $tx_j \in HBT_w$. The dependence sequence $\delta_w \rightsquigarrow^+ tx_j$ has already been kept in HBT_w. In either sub-case, HBT_w needs not to further capture τ to form $HBT_w \rhd tx_j$ as illustrated in Fig. 7(f).

In case ⑯, all the four conditions are not satisfied. τ is irrelevant to δ_w because $HBT_w \ntriangleright tx_i$ holds and tx_i is not reachable from any sequence captured in HBT_w. So, τ cannot be appended to any sequence captured in HBT_w to form $\delta_w \rightsquigarrow^+ tx_j$. In other

Algorithm 1. HBT Instances Maintenance

```
1  Input: τ = tx_i ⇝ tx_j ← a new dependency due to HB dependency e_a ↣ e_b
         t, δ_t ← a thread t starts a new transaction δ_t
         u, v ← threads of tx_i and tx_j, i.e., T(tx_i), T(tx_j)
2  function initialization(t, δ_t)
3  | HBT_t.clearTree()
4  | HBT_t.setRoot(δ_t)
5  | M_t.clear()
6  | M_t(t) = δ_t
7  end function

8  function captureDependency(τ, u, v)
9  | if tx_i is δ_u and M_u(v) = null then
10 | | HBT_u.addChild(tx_i, tx_j) //add τ to HBT_u
11 | | M_u(v) = tx_j
12 | end if
13 | Tid = Tid \ {u, v}
14 | for all w ∈ Tid do CheckOtherTrees(τ, w, u, v)
15 end function

16 function CheckOtherTrees(τ, w, u, v)
17 | if M_w(u) ≠ null and M_w(v) = null then
18 | | tx_y = M_w(u)
19 | | e_p = HBT_u.getDependencyWithParent(tx_y)
20 | | if isIncreasing(e_p, τ) then
21 | | | if tx_y is tx_i then
22 | | | | HBT_w.addChild(tx_y, tx_j) //add τ to HBT_w
23 | | | | M_w(v) = tx_j
24 | | | else
25 | | | | HBT_w.addChild(tx_y, tx_i) //add program order
26 | | | | HBT_w.addChild(tx_i, tx_j) //add τ to HBT_w
27 | | | | M_w(v) = tx_j
28 | | | end if
29 | | end if
30 | end if
31 end function
```

words, $\delta_w \rightsquigarrow^+ tx_j$ cannot be formed even with the presence of τ. So, as illustrated in Fig. 7(g), τ should not be captured into HBT_w.

■ In TABLE II, there is ***only one*** case that HBT_w cannot exclude τ from capturing for the detection of atomicity violation on δ_w and its witness. Specifically, in case ⑥, $M_w(u)$ is defined, $M_w(v)$ is undefined, and we have $HBT_w \rhd tx_i$ and $HBT_w \ntriangleright tx_j$. The function *checkOtherTrees* is specially designed to precisely handle this case at line 17 in Algorithm 1. Let tx_y be a transaction and $tx_y \in HBT_w$ so that tx_y and tx_i are executed by the same thread u. Also, let $e_p \in tx_y$ be the last event in the event sequence underlying the dependence sequence $\delta_w \rightsquigarrow^+ tx_y$ at line 19, and $e_a \in tx_i$ be the event of tx_i underlying the dependence τ. There are two-subcases to consider, depending on whether e_p happens before e_a due to program order of thread u. Suppose e_p happens before e_a. (This scenario is depicted in Fig. 7(i).) Since tx_y is reachable from HBT_w, thus, appending τ to $\delta_w \rightsquigarrow^+ tx_y$ is also increasing, which is ensured by line 20 in the algorithm. So, HBT_w appends τ the node tx_y to capture $\delta_w \rightsquigarrow^+ tx_j$ at lines 22 and 25-26 to make tx_j reachable from HBT_w. Then the algorithm updates $M_w(v)$ from undefined to tx_j at lines 23 and 27 to reflect that HBT_w can reach some transection (i.e., tx_j) of thread v. (Note that after the maintenance, we have $HBT_w \rhd tx_j$.) Next, suppose e_p does not happen before e_a. As depicted in Fig. 7(h), since $\langle e_p, e_a\rangle$ does not follow their temporal order in the trace, the dependence sequence $\delta_w \rightsquigarrow^+ tx_y$ appended with $\langle\tau\rangle$ is non-increasing. Thus, HBT_w will not capture τ because $\delta_w \rightsquigarrow^+ tx_y$ appended with $\langle\tau\rangle$ will never trigger an atomicity violation on δ_w.

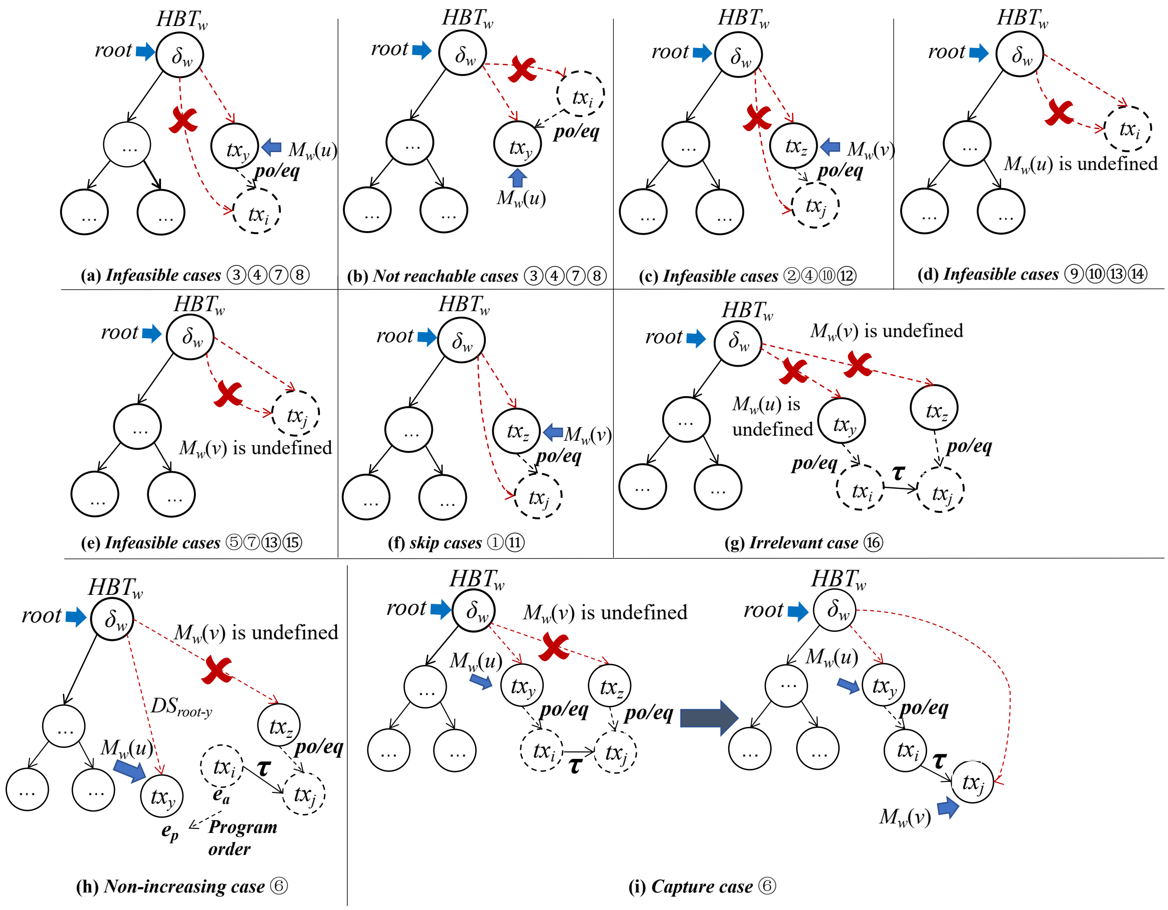

Fig. 7. Maintenance of HBT_w

■ As a result, HBT_w soundly captures a reduced set of increasing dependency sequences reachable from δ_w, which can infer any increasing dependency sequences reachable from δ_w.

B. Detecting Atomicity Violations and Localizing Witnesses

1) Illustration: In trace α_1 in Fig. 2, when dependency τ_{16} appears, $M_{t4}(t_3)$ has been mapped to $\tau_{16}.source$ (i.e., tx_5), and the beginning event $tx_4.begin$ happens before e_7. An atomicity violation on tx_4 is detected. At this moment, HBT_{t4} is $\langle\langle tx_4, tx_2, tx_6, tx_5\rangle, \langle\tau_{11}, \tau_{12}\rangle\rangle$ and the root of HBT_{t4} is tx_4. To localize the witness, HBT_{t4} attempts to reproduce a path ρ_1 for the increasing dependency sequence $tx_4 \rightsquigarrow^{+} tx_5$ appended with τ_{16}, it starts from adding the sink transaction of τ_{16} (i.e., $\tau_{16}.sink$ to the path and retrieving the transaction at the entry $M_{t4}(t_3)$ (i.e., tx_5). tx_4 and tx_5 are added to path $\rho_1 = \langle tx_5, tx_4\rangle$. tx_5 is set to be the currently visiting node. Then, Davida finds the parent node of tx_5 which is tx_6, adds it to path $\rho_1 = \langle tx_6, tx_5, tx_4\rangle$ and sets it as the currently visiting node. Similarly, Davida adds tx_2 and tx_4 to ρ_1 $= \langle tx_4, tx_2, tx_6, tx_5, tx_4\rangle$ in turn. Since tx_4 is the root node of HBT_{t4}, the witness localization ends. Davida reproduces a cyclic dependency sequence as the corresponding witness $\rho_1 = \langle tx_4, tx_2, tx_6, tx_5, tx_4\rangle$, which is cycle c_1 in Fig. 2. Similarly, when dependency τ_{17} appears, an atomicity violation on tx_1 is also detected, and the witness reported is $\rho_2 = \langle tx_1, tx_6, tx_5, tx_1\rangle$, which is cycle c_5 in Fig. 2.

2) Our Design: Algorithm 2 presents how Davida detects atomicity violations and further localizes a witness for every detected atomicity violation.

Consider the processing of dependency $\tau = tx_i \rightsquigarrow tx_j$ produced by the underlying HB dependency $e_a \rightarrowtail e_b$. Recall that tx_j is the active transaction of v, which is δ_v. HBT_v determines whether the two conditions hold: (Condition D1) tx_i is reachable from δ_v (i.e., the increasing dependence sequence $tx_j \rightsquigarrow^{+} tx_i$ can be formed by HBT_v); (Condition D2) $tx_j \rightsquigarrow^{+} tx_i$ appended with $\langle\tau\rangle$ forms a cyclic dependency sequence $tx_j \rightsquigarrow^{c} tx_i$, representing an atomicity violation on tx_j .

■ There are four possible combinations of the two conditions (D1) and (D2). However, (D2) is satisfiable only when (D1) is satisfied because if $tx_j \rightsquigarrow^{+} tx_i$ does not exist in HBT_v, there is no need to append τ after it. Therefore, Algorithm 2 first checks whether $M_v(u)$ maps to any transaction of thread u at line 5.

■ Consider the case where $M_v(u)$ is undefined. In this case, $tx_j \rightsquigarrow^{+} tx_i$ will not exist in HBT_v. Thus, the cyclic dependency sequence $tx_j \rightsquigarrow^{c} tx_i$ cannot be formed, indicating no atomicity violation on tx_j. Next, consider the case where $M_v(u)$ maps to a transaction of u, denoted as tx_y. First, suppose that thread u executes tx_i before tx_y (i.e., tx_y follows tx_i by program order). In this case, tx_i is not reachable from HBT_v (recall that $\delta_v = tx_j$). So,

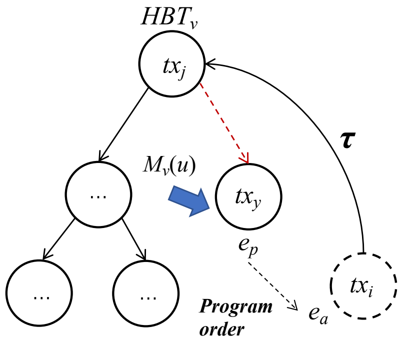

Fig. 8. Detection of Atomicity Violation on tx_j

neither $tx_j \rightsquigarrow^+ tx_i$ can be formed, nor $tx_j \rightsquigarrow^c tx_i$ can be formed, indicating no atomicity violation on δ_v. Next, suppose that thread u executes tx_y before tx_i (i.e., $tx_y \rightsquigarrow^+ tx_i$). In this case, tx_i is reachable from tx_j by appending $tx_j \rightsquigarrow^+ tx_y$ with $tx_y \rightsquigarrow^+ tx_i$. So, τ can be appended after $tx_j \rightsquigarrow^+ tx_y$ to form the cyclic sequence $tx_j \rightsquigarrow^c tx_i$, indicating an atomicity violation on tx_j. Algorithm 2 handles this case at line 6 (i.e., conditions (D1) and (D2) are both satisfied). An atomicity violation on tx_j is reported at line 7, and the witness $tx_j \rightsquigarrow^c tx_i$ is localized at line 8. Last, suppose $tx_y = tx_i$. In this case, we have $tx_i \in HBT_v$. Let $e_p \in tx_y$ be the last event in the event sequence underlying $tx_j \rightsquigarrow^+ tx_i$. Since tx_j is the first transaction in every increasing dependency sequence in HBT_v, the event $tx_j.begin$ must happen before e_p and $\langle tx_j.begin, e_p\rangle$ is increasing. If thread u executes e_a before e_p and thus $\langle e_p, e_a\rangle$ is non-increasing, then $\langle tx_j.begin, e_p, e_a\rangle$ is also non-increasing and $tx_j.begin$ must not happen before e_p. So, e_a and e_b cannot be appended after the event sequence underlying $tx_j \rightsquigarrow^+ tx_y$ to form $tx_j \rightsquigarrow^c tx_i$, indicating no atomicity violation on tx_j. On the other hand, if thread u executes e_p before e_a (i.e., $\langle e_p, e_a\rangle$ is increasing as depicted in Fig. 8), then $\langle tx_j.begin, e_p, e_a\rangle$ is increasing and $tx_j.begin$ will also happen before e_a. So, e_a and e_b can be appended after the event sequence underlying $tx_j \rightsquigarrow^+ tx_y$ to form $tx_j \rightsquigarrow^c tx_i$, indicating an atomicity violation on tx_j. Algorithm 2 handles this case at line 9 (i.e., conditions (D1) and (D2) are both satisfied). An atomicity violation on tx_j is reported at line 10, and the witness $tx_j \rightsquigarrow^c tx_i$ is localized at line 11. By doing so, Davida detects all atomicity violations and localizes a witness for every atomicity violation in a trace.

C. Correctness of Davida

Theorems 1-3 guarantee the correctness of Davida.

Theorem 1. *Suppose that transaction tx_j is the root of HBT_v Suppose further that a dependency τ' is non-increasing with respect to a dependency τ already kept in HBT_v, where the thread executing $\tau.sink$ and $\tau'.source$ are the same. Then τ' will not involve in any witness that leads to atomicity violations on tx_j in trace α.*

Theorem 2. *Davida reports an atomicity violation on transaction tx_j of thread v in trace α if and only if an event $e_b \in tx_j$ and a cross-thread dependency $tx_i \rightsquigarrow tx_j$ triggered by $e_a \rightarrowtail e_b$ are generated, where $e_a \in tx_i$, such that $tx_j \rightsquigarrow^+ tx_i$ formed by HBT_v and appended with $tx_i \rightsquigarrow tx_j$ construct a cyclic dependency sequence $tx_i \rightsquigarrow^c tx_j$.*

Algorithm 2. Atomicity Violation Detection and Witness Localization

```
1  Input:  τ = txi ⇝ txj ← a new cross-thread dependency by ea ↣ eb
           current ← currently visiting node
2  Output: witness ← a cyclic dependency sequence on τ.sink
3  function AVdetection(τ, ea)
4    u, v = T(txi), T(txj)
5    txy = Mv(u)
6    if isProgramOrder(txy, txi) then
7      report atomicity violation on txj
8      witnessLocalization(txy, τ)
9    else if isEqual(txy, txi) and txj.begin ↣ ea then
10     report atomicity violation on txj
11     witnessLocalization(txy, τ)
12   end if
13 end function

14 function witnessLocalization(txy, τ)
15   witness.append(txj)
16   if txy is txi then
17     witness.append(txi)
18   else
19     witness.append(txi)
20     witness.append(txy)
21   end if
22   current = txy
23   while true do
24     current = HBTv.getParent(current)
25     witness.append(current)
26     if current is txj then //find the root node, loop ends
27       break
28     end if
29   end while
30 end function
```

Theorem 3. *Davida localizes a witness for an atomicity violation on transaction tx_j of thread v through HBT_v in trace α if and only if an atomicity violation is reported on tx_j when an event $e_b \in tx_j$ and a cross-thread dependency $tx_i \rightsquigarrow tx_j$ triggered by $e_a \rightarrowtail e_b$ are generated in a.*

The full proofs for Theorems 1 and 2-3 have been presented in the paragraphs marked by "■" in Sections III.B.2, III.B.3 and III.C, respectively. Theorems 2-3 indicate that tracking increasing dependency sequences for active transactions is sufficient to both detect all atomicity violations and localize a witness for every atomicity violation and will not affect the coverage and soundness of all active and completed transactions.

When each of Velodrome and Davida reports an atomicity violation with a witness triggered by the same event, Theorem 4 indicates that the witnesses reported by Velodrome on the same thread must be a subsequence of that reported by Davida.

Theorem 4. *Let Velodrome and Davida both report an atomicity violation on transaction tx_j of thread v in trace α triggered by an event $e_b \in tx_j$. Suppose that a cross-thread dependency $tx_i \rightsquigarrow tx_j$ triggered by the variable-level dependency $e_a \rightarrowtail e_b$ is generated, where e_a is an event of tx_i. Moreover, let the witnesses reported by Velodrome and Davida for the atomicity violation are c_V and c_D, respectively. For any thread w such that $w \neq v$, suppose that the projected transaction sequences of the witnesses c_V and c_D on thread w are $cw_V = \langle tx_A, ..., tx_B\rangle$ and $cw_D = \langle tx_C, ..., tx_E\rangle$, respectively, where cw_V and cw_D are both not empty. Note that all transactions along cw_V and cw_D performed by w are ordered according to the program order. Then, we should have (1) $tx_C \rightsquigarrow^+ tx_A$ or $tx_A = tx_C$, and (2)if $tx_A \rightsquigarrow^+ tx_E$,* *then $tx_B \rightsquigarrow^+ tx_C$ or $tx_B = tx_C$. In other words, cw_V is a subsequence of cw_D (i.e., $tx_C \rightsquigarrow^+ tx_A \rightsquigarrow^+ tx_B \rightsquigarrow^+ tx_E$).*

Proof. For the first condition, since the entry $M_v(w)$ must have been mapped to tx_C and tx_C has the smallest timestamp among the transactions of the thread w captured in HBT_v, tx_A is either tx_C or a transaction following tx_C by the program order. Suppose that tx_C follows tx_A by the program order (i.e., $tx_A \rightsquigarrow^+ tx_C$). Then, the increasing dependency sequence $tx_j \rightsquigarrow^+ tx_A$ should have been constructed before $tx_j \rightsquigarrow^+ tx_C$. $M_v(w)$ should be mapped to tx_A with a smaller timestamp than tx_C in HBT_v. The increasing dependency sequence $tx_j \rightsquigarrow^+ tx_C$ should be formed by appending $tx_A \rightsquigarrow^+ tx_C$ after $tx_j \rightsquigarrow^+ tx_A$ using HBT_v. There is a contradiction. So, we must have either $tx_C \rightsquigarrow^+ tx_A$ or $tx_C = tx_A$.

For the second condition, as tx_B is the last transaction in cw_V, tx_B is the only transaction in cw_V that can form the increasing dependency sequence $tx_j \rightsquigarrow^+ tx_i$. On the other hand, tx_D is the transaction captured by HBT_v that has a dependency to transactions of a thread other than w and forms $tx_j \rightsquigarrow^+ tx_i$. Note that the increasing dependency sequence $tx_j \rightsquigarrow^+ tx_i$ via tx_D is constructed before that via tx_B. Suppose that tx_B follows tx_D by the program order (i.e., $tx_D \rightsquigarrow^+ tx_B$). Then, when traversing from the first transaction of the sequence to transaction tx_D, Velodrome should follow the dependency sequence starting from tx_D to form $tx_j \rightsquigarrow^+ tx_i$ rather than reporting the dependency sequence starting from tx_B, resulting in a contradiction. As a result, if $tx_A \rightsquigarrow^+ tx_D$, then either $tx_B \rightsquigarrow^+ tx_D$ or $tx_B = tx_D$. □

Discussion. To debug data races and deadlocks, developers need the racy pair of events and the sequence of lock dependency involving the data races [15][1] and deadlocks [6][49], respectively. Similarly, to debug an atomicity violation, developers should be informed with a witness for each detected atomicity violation. Recall that Velodrome [16] can detect some but not all atomicity violations. Thus, to debug against some invoked methods, tracking and diagnosing the dependencies based on the output of Velodrome is infeasible. Moreover, the witnesses reported by Velodrome may only provide a special case (rather than the general case) of the involved transactions leading to the detected violations. This may pose a difficulty for developers to diagnose the full picture of the root cause of an atomicity violation. In contrast, Davida can support developers for such program tracing and provide a more comprehensive view of the dependencies involved in the detected atomicity violations.

IV. Evaluation

A. Implementation

In order to conduct a fair comparison with other techniques, we must ensure all the techniques analyze the same execution trace. However, the dynamic behavior of a multithreaded program can vary significantly across different executions, even with the same input. Besides, due to the heap size limitation of JikesRVM, we cannot implement all the techniques simultaneously and let them run against the same dynamic execution.

TABLE III: Descriptive Statistics of Benchmarks

Subject	# of Threads	# of Transactions in 10^3	# of Dependencies
$eclipse_6$	18	33,000	354,168
$hsqldb_6$	43	8,670	17,176
$lusearch_6$	6	10,400	633,644
$xalan_6$	10	14,500	346,365
$avrora_9$	5	35,400	2,555,595
$lusearch_9$	4	10,200	187
$sunflow_9$	6	13,000	45,913
$xalan_9$	4	14,600	332,989
crypt	3	0.065	22
series	3	1,910	11
sor	3	0.036	13
sparsematmult	3	0.037	19
montecarlo	3	613	52,024
elevator	2	4,9	15,257
tsp	8	11	56
	Total	**143,000**	**4,353,439**

Thus, following [33], our experiment contained two parts: online and offline analyses. The online analysis was built on top of JikesRVM 3.1.3 [24] for performance evaluation (i.e., TABLE VII). JikesRVM [2] is an open-source Java virtual machine written in Java. We denote the instrumentation framework as Empty [33][3]. To conduct a direct and fair comparison, the offline analysis lets each technique analyze the same trace for non-serializable trace, atomicity violation detection, and witness localization (i.e., TABLE III-TABLE IV). We implemented Davida (referred to as DV) on top of the existing framework of RegionTrack (referred to as RT) [33][29] based on JikesRVM and Java for online and offline analyses, respectively. Besides, we made the implementation of Velodrome [3][16] (referred to as Velo) to report the located witnesses at the statement the implementation detects an atomicity violation. RT also provided dependencies to DV as their generation is not the core contribution of DV.

In the implementation, each transaction is a pair of integers (i.e., epoch [15]): one for thread identifier and another for the transaction timestamp. A dependency is an epoch pair. Each HBT instance was implemented as an array of epoch pairs. Each *RVMThread* object maintained an HBT instance to capture the increasing dependency sequences from the active transactions. Our implementations had been tested on a few small programs.

B. Benchmarks and Experimental Setup

We adopted the set of subjects used in [33][34]. Specifically, we evaluated Velo, RT and DV using the DaCapo [5], Java Grande Forum [42] benchmark and microbenchmark [54] suites. The following 15 programs were used in the experiment: $eclipse_6$, $hsqldb_6$, $lusearch_6$, and $xalan_6$ from DaCapo 2006-10-MR2; $avrora_9$, $lusearch_9$, $sunflow_9$, and $xalan_9$ from DaCapo 9.12-bach; *crypt*, *series*, *sor*, *sparsematmult* and *montecarlo* from Java Grande Forum; and *elevator* and *tsp* from microbenchmark. All programs are single-process multithreaded. These programs were correctly executed successfully on Jikes RVM 3.1.3 in our environment. These programs from DaCapo and Java Grande Forum benchmark suites contained atomicity violations based on atomicity specifications [3]. We did not present the results of the remaining benchmarks in [33] because no atomicity violation was found on them [33]. We, however, have run DV on them to have confirmed that DV did not report any atomicity violation or witness for them. We have also run all programs (*elevator*, *hedc*, *philo*, *sor* and *tsp*) in microbenchmark suites. However, only *elevator* and *tsp* contained atomicity violations. Thus, we did not present the results of the remaining programs.

TABLE III shows the descriptive statistics of the subjects. We followed the experiments of [3][16][33][34] to use the small input size of the DaCapo subjects and the input size A of the Java Grande Forum subjects. We use the same input configurations in [34][55] for microbenchmark. All dynamic data were collected on Empty. Following [3][16][33], the presented results are the mean results of 10 trials. The first four columns show the subject names, the numbers of threads, (regular and unary) transaction nodes and cross-thread dependencies processed in the execution traces, respectively.

For online analysis, our experiments ran on an Ubuntu Linux 12.04 x86_64 virtual machine built on a server with two 2.20GHz Intel Xeon E7-4850 v3 processors. The virtual machine was configured with two logical processors (2 cores), 16GB memory, and OpenJDK 1.6. We followed [3][33] to compile the frameworks using the production configuration in JikesRVM, which was closer to the production environment. The JikesRVM was configured with 2GB memory. JikesRVM has a limitation on the version of JDK. Building JikesRVM with JDK 9 or later is currently not supported [56]. Similar to [3] [29], in our environment, JikesRVM 3.1.3 only ran successfully on Ubuntu Linux 12.04. For the offline analysis, our experiments ran on an Ubuntu Linux 18.04 x86_64 virtual machine built on a server with two 2.20GHz Intel Xeon E7-4850 v3 processors. The virtual machine was configured with two logical processors (2 cores), 128GB memory, and OpenJDK 11. The system for online analysis is an old version of Ubuntu virtual machine, we thus chose to use a recent version of the virtual machine for offline analysis. Since the offline analysis was used to let each technique analyze the same recorded execution traces and compare their detection results against the same trace, a different system compared with the online analysis would not affect the detection results. We collected 100 traces for each subject over the same input on Empty.

Atomicity specifications for the subjects were used in our experiment. For the offline analysis, we strictly followed [3][33] [34] to use the initial atomicity specification provided by Biswas et al. [3] and adopt the iterative refinement methodology [3][33] to analyze traces. The atomicity specification produced by the iterative refinement methodology was as follows: First, all methods are assumed to be atomic except those methods that are not in the initial specification [3]. These methods are intended to run non-atomically which include top-level methods (e.g., main() and Thread.run()), methods that contain interrupting calls (e.g., wait() and notify()) and DaCapo benchmarks' driver thread. Then, if any instance of a method is detected for an atomicity violation in the current round of analysis, this method will be removed from the specification for the next round of analysis, and the instances of this method will still be analyzed in the current round of analysis. If no new atomicity violation was reported after two successive rounds, the iterative refinement process ends [3]. We repeated the experiment over the 100 collected traces for each subject and made the tool and data available at [22].

TABLE IV. Average Numbers of Detected AVs and Transactions with AVs for a trace

Subject	# of detected AVs			# of transactions with AVs		
	Velo	**RT**	**DV**	**Velo**	**RT**	**DV**
eclipse6	19,677	251,500	251,500	1,261	1,718	1,718
hsqldb6	762	8,267	8,267	221	227	227
lusearch6	3	643,325	643,325	2	5	5
xalan6	13,250	1,024,043	1,024,043	7,465	8,469	8,469
avrora9	838,981	8,360,455	8,360,455	602,730	656,752	656,752
lusearch9	18	109	109	18	31	31
sunflow9	33	63	63	29	34	34
xalan9	1,964	288,625	288,625	1,288	1,904	1,904
crypt	1	1	1	1	1	1
series	1	1	1	1	1	1
sor	1	1	1	1	1	1
sparsematmult	1	1	1	1	1	1
montecarlo	1,874	2,154	2,154	1,874	2,154	2,154
elevator	6	58	58	6	11	11
tsp	1	3	3	1	1	1
Total	**876,573**	**10,578,606**	**10,578,606**	**614,899**	**671,310**	**671,310**

Following the iterative refinement methodology, the offline analysis first allocates transactions for the outermost method. When the outermost method is detected for atomicity violations in the current round of analysis and removed from the atomicity specification, the analysis allocates transactions for inner method in the next round of analysis to handle nested methods. The same procedure was used by [3][33].

C. Research Questions

We aim to answer the following research questions.

RQ1: Compared to RT and Velo, does DV detect all atomicity violations and localize a witness for every atomicity violation?

RQ2: Compared to Velo, is DV efficient in maintaining the dependencies to localize the witnesses?

RQ3: Compared to RT and Velo, is DV both time- and memory-efficient?

D. Results and Data Analysis

1) Atomicity Violation Detection and Witness Localization

TABLE IV shows the main results for Velo, RT and DV. Columns 2-4 show the mean numbers of detected atomicity violations by Velo, RT and DV over 100 collected traces. RT is a sound and complete atomicity violation checker. As expected, DV and RT detected the same numbers of atomicity violations. Since Velo added at most one edge between any two transactions, it only detected 8.3% of atomicity violations reported by RT and DV. Columns 5-7 show the mean numbers of transactions with atomicity violations reported by Velo, RT and DV over 100 collected traces. Each transaction represented at least one instance of atomicity violation on it. DV and RT detected the same numbers of transactions with atomicity violations. Velo missed detecting some transactions detected by DV and RT. All of Velo, RT and DV identified non-serializable traces in a sound and complete manner.

TABLE V shows the overall effectiveness results of Velo, RT and DV. Columns 2-4 show the mean numbers of witnesses for atomicity violations localized by the three techniques, Velo, RT and DV, over 100 collected traces. Since RT discarded all dependencies during the analysis, it could not locate any witness for atomicity violations over all subjects. Compared to DV, Velo missed locating many atomicity violation instances with witnesses.

Columns 5-7 show the mean numbers of distinct witnesses reported by Velo, RT and DV over 100 collected traces. These columns group the reported witnesses sharing the same sequence of transactions and count each such witness as one. Each distinct witness represents a unique scenario that triggers an atomicity violation on a transaction at the transaction level. Overall, RT cannot locate any witnesses. DV detected more distinct witnesses than that of Velo.

Columns 11-13 are marked with "✓" if at least one witness is detected for every transaction with atomicity violation by Velo, RT and DV over 100 traces. For all subjects, RT could not report any witness for every transaction with atomicity violation. On most subjects, Velo missed reporting witnesses for transactions with atomicity violation due to its path-insensitive graph-based approach. In contrast, DV detected at least one witness for every transaction with atomicity violation over all subjects.

Answer to RQ1: The experimental results are consistent with the theory of DV that DV should detect all atomicity violations. Velo missed reporting many atomicity violations. Compared TABLE IV with TABLE V, RT cannot locate any witnesses. Velo missed reporting many witnesses and atomicity violations. In contrast, DV located a witness for every atomicity violation.

2) Dependency Maintenance for Witness Localization

Since RT does not keep any dependencies and thus cannot locate any witnesses, we only present the relative results for Velo and DV in TABLE VI. Columns 2 and 3 show the mean numbers of visited transactions to localize witnesses by Velo and DV over 100 traces. During the localization of witnesses along the trace, DV and Velo visited many nodes in their HBT instances and dependency graph, respectively. Each visited occurrence of any node is counted as one. Recall that DV directly localized the witness along the increasing dependency sequence maintained by the HBT instances, and Velo searched its graph to locate witnesses and contained those failed acyclic paths.

On *hsqldb*$_6$ and *xalan*$_9$, the numbers of transactions visited by Velo were 2.22x and 1.81x more than that visited by DV. On *xalan*$_6$ and *montecarlo*, the numbers of transactions visited by DV to localize witnesses were 15.72x and 37.27x less than that of Velo. On *sunflow*$_9$, *eclipse*$_6$ and *lusearch*$_9$, the numbers of transactions visited by DV to localize witnesses were 642x to 13338x less than that of Velo. On *elevator* and *tsp*, Velo visited 1456x and 8.2x more transactions nodes than DV to locate witnesses. Note that DV reported more witnesses than Velo but visited fewer transactions for the above subjects. On *crypt*, *series*, *sor* and *sparsematmult*, DV and Velo reported the same number of witnesses. These witnesses involved only 2 or 3 transactions that were directly localized by DV. However, Velo needed to visit 3.00x to 6.50x more transactions to localize these witnesses. On *lusearch*$_6$ and *avrora*$_9$, DV visited more transactions than Velo. However, the underlying reason is that DV localized significantly more witnesses than Velo (see columns 5 and 7 in TABLE V). Across all subjects, the number of transactions visited by DV to localize witnesses was 26.39x less than that of Velo.

Columns 5 and 6 show the mean numbers of dependencies kept by Velo and DV over 100 collected traces. Each dependency kept represented that a cross-thread dependency is kept in the dependency graph of Velo or is kept in an HBT instance of DV. On average, Velo kept more dependencies than DV by 7.20x. Compared to columns 2 and 3, the reduced number of visited transactions by DV is primarily due to precisely maintaining the increasing dependency sequences and the ability to

TABLE V: Average Numbers of Witness and Distinct Witness for a trace

Subject	# of witnesses			# of distinct witnesses			One witness for every transaction with AV?		
	Velo	RT	DV	Velo	RT	DV	Velo	RT	DV
eclipse$_6$	19,677	0	251,500	19,677	0	22,088	×	×	✓
hsqldb$_6$	762	0	8,267	762	0	1,015	×	×	✓
lusearch$_6$	3	0	643,325	3	0	12	×	×	✓
xalan$_6$	13,250	0	1,024,043	13,250	0	17,349	×	×	✓
avrora$_9$	838,981	0	8,360,455	838,981	0	1,061,435	×	×	✓
lusearch$_9$	18	0	109	18	0	31	×	×	✓
sunflow$_9$	33	0	63	33	0	39	×	×	✓
xalan$_9$	1,964	0	288,625	1,964	0	2,779	×	×	✓
crypt	1	0	1	1	0	1	✓	×	✓
series	1	0	1	1	0	1	✓	×	✓
sor	1	0	1	1	0	1	✓	×	✓
sparsematmult	1	0	1	1	0	1	✓	×	✓
montecarlo	1,874	0	2,154	1,874	0	2,154	×	×	✓
elevator	6	0	58	6	0	11	×	×	✓
tsp	1	0	3	1	0	2	×	×	✓
	876,573	**0**	**10,578,606**	**876,573**	**0**	**1,106,919**	**-**	**-**	**-**

TABLE VI. Average Numbers of Visited Transactions, Dependencies Ever Kept and Dependencies Ever Skipped, Maximum Forest Size and Maximum Number of Dependencies Skipped in Forest for a trace

Subject	# of visited transactions			# of dependencies ever kept			# of dependencies ever skipped			Max forest size	Max # of dep. skipped
	Velo (A)	DV (B)	A/B	Velo (C)	DV (D)	C/D	Velo (E)	DV (F)	F/E	DV	DV
eclipse$_6$	629,642,542	635,874	990.20	446,137	957	466.18	281,109	726,289	2.58	34	338,796
hsqldb$_6$	90,614	40,734	2.22	4,573	453	10.09	13,224	17,345	1.31	81	11,030
lusearch$_6$	273,525	1,378,152	0.20	139,938	8	17492.25	493,706	633,636	1.28	2	6,393
xalan$_6$	59,935,702	3,812,909	15.72	348,804	2,213	157.62	29,822	318,543	10.68	38	94,577
avrora$_9$	16,567,784	20,344,284	0.81	1,213,829	349,291	3.48	1,663,040	2,527,578	1.52	5	4,281
lusearch$_9$	2,614,224	196	13337.88	286,730	12	23894.17	415,388	702,107	1.69	2	4,593
sunflow$_9$	92,477	144	642.20	13,334	9	1481.56	61,958	75,284	1.22	2	75,088
xalan$_9$	1,289,589	711,635	1.81	40,255	281	143.26	68,389	108,365	1.58	2	2,095
crypt	16	3	5.33	9	3	3.00	12	18	1.50	3	25
series	13	2	6.50	9	4	2.25	2	8	4.00	2	11
sor	9	2	4.50	5	2	2.50	8	12	1.50	2	18
sparsematmult	9	3	3.00	4	1	4.00	15	18	1.20	1	27
montecarlo	192,442	5,163	37.27	104,104	8,028	12.97	88,555	184,632	2.08	5	478
elevator	2,912	2	1456	3,809	71		11,448	15,186	1.33	4	72
tsp	41	5	8.20	36	15		20	41	2.05	17	73
Total	**710701,899**	**26,929,108**	**-**	**2,601,576**	**361,348**	**-**	**3,126,696**	**5,309,062**	**-**	**200**	**537,557**
	Total A / Total B		**26.39**	**Total C / Total D**		**7.20**	**Total F / Total E**		**1.70**	**-**	**-**

not keep those non-reachable, non-increasing or skippable dependencies from active transactions.

Columns 8 and 9 show the mean numbers of dependencies skipped by Velo and DV over 100 traces. Each dependency skipped by Velo represented that a cross-thread dependency between the same two transactions is already kept in the dependency graph. Each dependency skipped by DV means that it needs not be kept by ***all*** HBT instances in the forest of DV when processing the dependency. On average, DV precisely and safely skipped more dependencies than Velo by 1.70x.

Column 11 presents the mean size of the forest (in terms of number of dependencies) at the moment of keeping the max number of dependencies in a trace during the analysis over 100 traces. Each dependency may be captured into one or more HBT instances in the forest of DV, and each such capturing of a dependency into an HBT instance is counted as one. For all subjects, DV captured at most 1 to 81 residual dependencies. Column 12 shows the number of dependencies skipped among all HBT instances in the forest in a trace during the analysis (i.e., max # of dep skipped), averaging over 100 traces. Each dependency skipped by one HBT instance is counted as one. On average, DV skipped up to 11 to 338,796 dependencies in the forest during the analysis. The results show that DV is effective in keeping the number of dependencies small.

We investigate five subjects (*eclipse*$_6$, *hsqldb*$_6$, *lusearch*$_6$, *xalan*$_6$ and *avrora*$_9$) to collect their mean numbers of dependencies such that each is reachable and increasing from one active transaction but non-increasing from another active transaction over 100 collected traces. These subjects include more than two threads and contained 14979, 992, 5086, 61862 and 188548 such dependencies, respectively.

We recalled that in the offline analysis, we did not compare DV with DoubleChecker [3] for the following reasons: According to the experiments presented in [33], DoubleChecker falsely reported many instances of atomicity violations (e.g., up to 70% on *eclipse*$_6$), which required follow-up confirmation analyses. Besides, DoubleChecker adopted the graph construction approach of Velo but without the timestamp information. DoubleChecker utilized a depth-first traversal algorithm for cycle localization over the graph as well. Usually, there were more acyclic paths than cyclic paths in a graph [33]. Before a cycle was detected, such an algorithm might find acyclic paths first. In the experiment reported in [33], on subject *avrora*$_9$, DoubleChecker ran out of memory on JikesRVM.

> **Answer to RQ2:** During the witness localization, DV reduced over 26.39x visited transactions accessed by Velo to localize these witnesses on average. Besides, the HBT instances in DV precisely captured fewer dependencies than the centralized graph in Velo by 7.19x for atomicity violation detection and witness localization.

4) Time and Memory Overhead

TABLE VII presents the memory overheads of each technique. Base means the results of the subject executing on the un-instrumented virtual machine. Memory consumptions are collected via Linux time command. The second column shows the memory consumed by Base. The memory overhead is the ratio of the memory used by a technique to the memory used by Base (i.e., memory overhead = technique's memory consumption ÷ Base's memory consumption [15]). The third to fifth columns present the memory overhead incurred by Velo, RT and DV.

In TABLE VII, Velo incurred the heaviest memory overhead on 12 out of 15 subjects than RT and DV. On average, Velo incurred 1.57x memory overhead. On subject *avrora*$_9$, Velo consumed significantly more memory than RT and DV. It needed to maintain a large dependency graph and could not delete a transaction node if the node was reachable from active transactions, making it incur a larger memory overhead than RT and DV. In general, RT and DV incurred similar memory overhead: 1.27x and 1.28x, respectively. The results confirm that tracking dependencies in DV is memory efficient and DV only incurs small additional overheads than RT.

TABLE VII also presents the slowdown on each subject by using each technique. We collected the CPU time via the Linux time command. The slowdown incurred by each technique is reported as the technique's time spent ÷ Base's time spent [15], and the results incurred by Velo, RT and DV are shown in the seventh to ninth columns. The sixth column presents the time spent by Base.

From TABLE VII, Velo incurred the heaviest slowdown on 13 out of 15 subjects than RT and DV. On the rest two subjects, Velo, RT and DV incurred the same slowdowns. On the three relatively large subjects (*eclipse*$_6$, *xalan*$_6$ and *avrora*$_9$), DV was faster than Velo by at least 1.4x. On average, Velo incurred 5.08x slowdown while RT and DV incurred 4.17x and 4.43x slowdown, respectively.

TABLE VII. Memory Overhead and Slowdown

Subject	Base (MB)	Memory Overhead			Base (sec.)	Slowdown		
		Velo	RT	DV		Velo	RT	DV
eclipse$_6$	828	1.53	1.45	1.44	16.69	2.11	1.50	1.47
hsqldb$_6$	468	1.17	1.15	1.16	2.17	1.92	1.86	1.62
lusearch$_6$	396	1.37	1.49	1.48	1.91	4.55	3.53	3.62
xalan$_6$	554	1.80	1.39	1.50	2.29	6.24	3.52	4.19
avrora$_9$	521	3.94	1.52	1.49	7.15	13.67	7.77	9.76
lusearch$_9$	415	1.70	1.36	1.36	2.03	4.86	3.81	3.67
sunflow$_9$	407	1.43	1.42	1.42	1.89	9.70	9.06	9.41
xalan$_9$	499	2.16	1.35	1.39	2.45	3.17	2.96	3.16
crypt	326	1.04	1.00	1.00	0.95	2.35	2.18	2.27
series	228	1.08	1.00	1.00	3.15	1.00	1.00	1.00
sor	320	1.45	1.14	1.14	2.13	1.00	1.00	1.00
sparsematmult	296	1.35	1.46	1.45	1.06	17.49	16.76	17.25
montecarlo	670	1.11	1.10	1.10	3.58	2.98	2.67	2.89
elevator	158	1.17	1.00	1.01	23.64	1.02	1.00	1.00
tsp	234	1.27	1.27	1.27	0.25	4.09	3.95	4.08
Mean	**-**	**1.57**	**1.27**	**1.28**	**-**	**5.08**	**4.17**	**4.43**

Answer to RQ3: On average, Velo incurred the heaviest memory overhead, 1.57x, and the heaviest slowdown among the three techniques. DV incurred slightly higher memory overhead and slowdown than RT. Since RT cannot locate any witnesses, DV is time- and memory-efficient to both detect all atomicity violations and locate a witness for every atomicity violation.

E. Case Study

In our experiments, Davida precisely detected all atomicity violations and localized a witness for every atomicity violation. Fig. 9 shows one increasing cyclic dependency sequence and one non-increasing cyclic dependency sequence on *PriceStock* in *MonteCarlo* of Java Grande Forum benchmark suite. A happens-before dependency between the head and the tail statements is presented by the arrow line. The interleaving is shown in Fig. 9. Threads 1, 2 and 3 all executed an instance of *PriceStock*. After the statement S_2 of thread 3 has executed, the instance of *PriceStock* of thread 1 established a dependency with the instance of *PriceStock* of thread 3 (i.e., τ_1) due to writing to the same memory location. Then, after the statement S_3 has executed, the instance of *PriceStock* of thread 3 established a dependency with the instance of *PriceStock* of thread 2 (i.e., τ_2). After the statement S_4 of thread 1 has executed, the instance of *PriceStock* of thread 2 established a dependency with the instance of *PriceStock* of thread 1 (i.e., τ_3). The dependency sequence ⟨τ_1, τ_2, τ_3⟩ forms a non-increasing cyclic dependency sequence. Then, the instance of *PriceStock* of thread 2 established another dependency with the instance of *PriceStock* of thread 1 (i.e., τ_4). The dependency sequence ⟨τ_1, τ_2, τ_4⟩ forms an increasing cyclic dependency sequence, representing an atomicity violation on the instance of *PriceStock* of thread 1 with the sequence as the witness.

Davida detected the atomicity violation on the instance of *PriceStock* and reported the sequence ⟨τ_1, τ_2, τ_4⟩ as the witness. However, since Velodrome only adds at most one edge between the same two transactions, dependency τ_4 was not added to the graph maintained by Velodrome. So, Velodrome only located the non-increasing sequence ⟨τ_1, τ_2, τ_3⟩ and missed reporting the atomicity violation on the instance of *PriceStock* of thread 1 and its witness.

Fig. 10 shows two increasing cyclic dependency sequences on *ElemNumber$getCountString* (or *getCountString* for short) in *xalan*$_6$ of DaCapo benchmark suite, representing two scenarios triggering the atomicity violation on the instance of *getCountString*. Threads 1, 2 and 3 executed an instance of *ElemNumber$getCountString*, an instance of *StringBufferPool$free* (or *free* for short), and an instance of *StringBufferPool$get* (or *get* for short) followed by another instance of *StringBufferPool$free*. After the statement S_2 of thread 2 has executed, the instance of *getCountString* of thread 1 established a dependency with the instance of *free* of thread 2 (i.e., τ_5) due to writing to the same memory location. Then, after the statement S_3 has executed, the instance of *free* of thread 2 established a dependency with the instance of *get* of thread 2 (i.e., τ_6) due to accessing the same lock. After the statement S_6 of thread 3 has executed, the instance of *free* of thread 2 established a dependency with the instance of *free* of thread 1 (i.e., τ_7). Then, after the statement S_7 of thread 1 has executed, the instance of

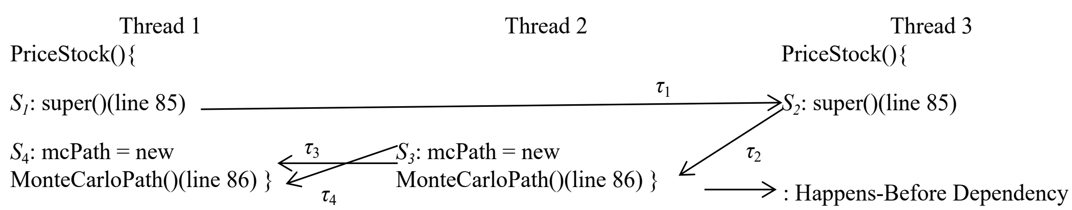

Fig. 9. Localized Witness for Atomicity Violation on *PriceStock*() in *MonteCarlo* by Davida

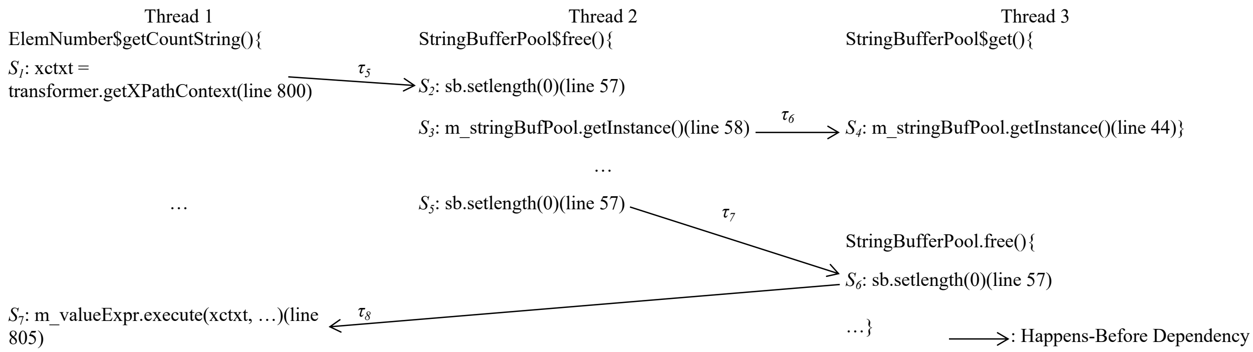

Fig. 10. Localized Witness for Atomicity Violation on *ElemNumber$getCountString*() in *xalan*$_6$ by Davida and Velodrome

free of thread 3 established a dependency with the instance of *getCountString* of thread 1 (i.e., τ_8), triggering the atomicity violation on the instance of *getCountString* of thread 1. The two increasing dependency sequences $\langle \tau_5, \tau_6, \tau_8 \rangle$ and $\langle \tau_5, \tau_7, \tau_8 \rangle$ both witnessed the atomicity violation on the instance of *getCountString*.

Velodrome located the cyclic sequence $\langle \tau_5, \tau_7, \tau_8 \rangle$ and found the sequence is increasing. Thus, Velodrome detected an atomicity violation on the instance of *getCountString* and reported the sequence $\langle \tau_5, \tau_7, \tau_8 \rangle$ as a witness. On the other hand, Davida also detected an atomicity violation on the instance of *getCountString*. However, it located the sequence $\langle \tau_5, \tau_6, \tau_8 \rangle$ as a witness. Although Velodrome and Davida each detects the violation and reports a witness for the violation, the witness reported by Velodrome does not include the instance of *get*. When developers manually inspect the code of the atomic regions in the witness, they are able to construct the dependency between different instances of *free*. However, even if they resolve such dependency, the atomicity violation on the instance of *getCountString* still exists because the developers are not informed to consider the dependency between instances *free* and *get*. On the other hand, the witness reported by Davida includes the instance of *get*, providing a more comprehensive scope for developers to inspect and resolve the violation on the instance of *getCountString*.

F. Threats to Validity

Our experiment was conducted on a total of 24 subjects from three benchmark suites: two versions of DaCapo benchmark suites [5], Java Grande Forum benchmark suite [42] and microbenchmark suite [54]. The experimental results showed that 15 subjects incurred atomicity violations while the remaining 9 subjects were not reported for any atomicity violations. Using subjects of other benchmark suites may provide different results and can also improve the generalization of the experimental results. JikesRVM has a limitation [25] of the benchmark suites that can successfully execute on it. Thus, using a different framework may obtain more experimental results. In our benchmark suites, all programs are single-process multithreaded. Thus, utilizing multi-process multithreaded programs may have different results and can be used to verify the generalization of the proposed technique.

Atomicity specifications are needed in our experiment. However, the atomicity specifications of the subjects in our experiment are either not annotated or not publicly available. So, following [3][33][34], we used the initial atomicity specifications provided by Biswas et al. [3]. The initial specifications assume that all methods are atomic except some methods, such as main() and Thread.run(), are non-atomic on purpose. In practice, such assumptions work fine as previous works [13][14][16][50] used the same assumptions, and their experiments have confirmed that atomicity is a desirable property for concurrency. Moreover, running experiments against such assumptions provide a baseline.

It is hard to reuse the original implementation of Velodrome for a direct and fair comparison. The main reason is that the original work uses a different dynamic bytecode instrumentation framework, RoadRunner [12]. From [12][16], it alone slows the programs on average by roughly 4-5x. For subject *montecarlo*, the execution time of *montecarlo* by Velodrome on RoadRunner is 21s. Following [3][33], our implementation uses a different instrumentation framework, JikesRVM. In the experiment of RegionTrack [33], the execution time of *montecarlo* by its implemented Velodrome is 9.29s. In our experiment, our implemented Velodrome ran *montecarlo* for 10.67s. The absolute time between these two implementations is consistent. However, in the experiment of DoubleChecker, *montecarlo* executed by their Velodrome incurred 4.5x slowdown. The same subject running on the same framework but on a different system environment also incurred different slowdowns. Valor [4] compares the differences between RoadRunner and JikesRVM. It states that the implementation inside a JVM (e.g., JikesRVM) substantially outperforms the implementation outside a JVM on top of a general dynamic bytecode instrumentation framework (e.g., RoadRunner). Thus, we believe our experiment results are correct and different framework and system environment both influence the runtime of the same subject.

In our experiment, we conducted an offline analysis. The purpose of the offline analysis is to ensure that all the three techniques (Velo, RT and DV) analyze the same execution trace against the subjects for a fair comparison. The underlying reason is that the dynamic behavior of a multithreaded program can vary significantly across different executions, even with the same input. We could not control the executions when dynamically running each technique independently against the same subject with the same input. An alternative approach is to run the three techniques simultaneously against the same dynamic execution. This is infeasible because the underlying framework JikesRVM targets the IA-32 32bit platform and is limited to a heap size of approximately 1.5-2GB. A 64-bit implementation of JikesRVM or running the experiment on a different framework could resolve this problem.

In the offline analysis, we conducted our experiment over 100 collected traces, which only occupied a tiny fraction of the interleaving space. However, the statistics of the collected traces did not vary drastically across different collected traces. Running the experiment on more execution traces could provide more generalized results.

The current implementations could successfully handle field and array accesses as well as lock operations. These implementations could not handle volatile variable accesses and synchronization idioms such as barriers, wait-notify, etc., which we leave as future work. Despite that, the experiment has been conducted successfully on the current implementations, and the empirical results have already shown the effectiveness and efficiency of our technique.

V. Related Work

A. Static Analysis and Model Checking

Static analysis techniques [9][18][44][39] check atomicity without executing the programs by examining the code for all inputs. They are imprecise and need a follow-up confirmation. Model checking methods are also proposed to check atomicity by exploring all possible executions. Due to the state space explosion problems, different reduction methods [17][20] are leveraged to reduce the search space. However, these methods cannot scale well for large-scale multithreaded programs.

B. Dynamic Analysis

Atomizer [13][14] combines the lockset algorithm [38] used in data race detection and Lipton's reduction theory [28] to check atomicity. Velodrome [16] and DoubleChecker [3] both build a centralized dependency graph to detect atomicity violations with witnesses, while DoubleChecker utilizes a more efficient instrumentation framework Octet [36]. Farzan and Madhusudan [19] also build a centralized dependency graph to detect atomicity. They summarize the effect of completed transactions and absorb the event content into active transactions. However, their technique is offline and can only detect non-serializable traces. If using a similar strategy as Velodrome and further recording timestamps for each dependency, their technique can detect atomicity violations at the transaction level but suffers from the same problem of Velodrome as presented in Section II. AeroDrome [34] utilizes vector clocks to identify non-serializable traces. RegionTrack [33] soundly and completely detects atomicity violations and non-serializable traces, but it does not capture any dependencies and cannot locate any witness for atomicity violations.

C. Predictive Analysis

Some techniques [7][8][10][11][23][41][48] detect single-variable or multi-variable atomicity violations not only for the observed trace but also for other possible interleavings of the observed trace. These techniques log the observed trace first but may produce false negatives or false positives in detecting such predictive atomicity violations.

D. Two-Phase Strategy

As the atomicity specification may not always be defined or provided by developers, another kind of techniques [31][37][40][43][45][46] utilize a two-phase strategy to check atomicity. In the first phase, they predict suspicious instances of atomicity violations in the observed trace. In the second phase, these techniques schedule confirmation runs to examine the detected suspicious atomicity violation instances.

E. Other Related Work

There are other kinds of atomicity violations. Some techniques detect atomicity violations [31][32][37] that involve three accesses of two threads to the same variable. Other techniques detect atomic-set serializability [21][27][47]. Some techniques [57][58][59] detect linearizability violations. Davida efficiently and precisely detects conflict-serializability violations of atomic regions with witnesses.

VI. Conclusion

In this paper, we have presented a novel online checker, named Davida, to detect all atomicity violations each with a witness in a trace. Davida has addressed the challenge that a dependency could be increasing with respect to one transaction and non-increasing with respect to another transaction. It is also novel in its distributed design. Davida is sound, guaranteed by our theorems. The experiment has shown the effectiveness and efficiency of Davida. We recommend using RegionTrack followed by Davida on the same trace so that non-serializable trace, atomicity violation and witness can all be precisely detected.

References

[1] A. Pavlogiannis, "Fast, sound, and effectively complete dynamic race prediction", in *Proc. ACM Program. Lang.* 4, POPL, Article 17 (January 2020), 29 pages, 2019.

[2] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J.E.B. Moss, T. Ngo, V. Sarkar, and M. Trapp, "The Jikes research virtual machine project: Building an open-source research community." *IBM Systems Journal*, 44(2), pp. 399–417, 2005.

[3] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "DoubleChecker: efficient sound and precise atomicity checking", in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pp. 28–39, 2014.

[4] S. Biswas, M. Zhang, M.D. Bond, and B. Lucia, "Valor: efficient, software-only region conflict exceptions", In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*, pp. 241–259, 2015.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "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 (OOPSLA '06)*, pp. 169–190, 2006.

[6] Y. Cai, R. Meng, and J. Palsberg, "Low-overhead deadlock prediction", in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, pp. 1298–1309, 2020.

[7] Y. Cai, H. Yun, J. Wang, L. Qiao, and J. Palsberg, "Sound and efficient concurrency bug prediction", in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, pp. 255–267, 2021.

[8] F. Chen, T. F. Serbanuta, and G. Rosu, "JPredictor: a predictive runtime analysis tool for java", in *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pp. 221–230, 2008.

[9] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller, "HAVE: Detecting atomicity violations via integrated dynamic and static analysis", in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09)*, pp. 425–439, 2009.

[10] M. Eslamimehr and M. Lesani, "AtomChase: Directed search towards atomicity violations", in *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE '15)*, pp. 12–23, 2015.

[11] M. Eslamimehr, M. Lesani, and G. Edwards, "Efficient detection and validation of atomicity violations in concurrent programs", *Journal of Systems and Software*, 137(3), pp. 618–635, 2018.

[12] C. Flanagan and S.N. Freund, "The RoadRunner Dynamic Analysis Framework for Concurrent Programs", (*PASTE '10*), pp. 1-8, 2010.

[13] C. Flanagan and S.N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs", in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04)*, pp. 256–267, 2004.

[14] C. Flanagan and S.N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs", *Science of Computer Programming (SCP)*, 71(2), pp. 89–109, 2008.

[15] C. Flanagan and S.N. Freund, "FastTrack: efficient and precise dynamic race detection", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pp.121–133, 2009.

[16] C. Flanagan, S.N. Freund, and J. Yi, "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 (PLDI '08)*, pp. 293–303, 2008.

[17] C. Flanagan, "Verifying commit-atomicity using model checking", in *Proceedings of 11th International SPIN Work-shop on Model Checking of Software (SPIN '04)*, pp. 252–266, 2004.

[18] C. Flanagan and S. Qadeer, "A type and effect system for atomicity", in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)*, pp. 338–349, 2003.

[19] A. Farzan and P. Madhusudan, "Monitoring Atomicity in Concurrent Programs", in *Proceedings of the 20th international conference on Computer Aided Verification (CAV '08)*, pp. 52–65, 2008.

[20] J. Hatcliff, Robby, and M.B. Dwyer, "Verifying atomicity specifications for concurrent object-oriented software using model-checking", in *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, pp. 175–190, 2004.

[21] C. Hammer, J. Dolby, M.Vaziri and F.Tip, "Dynamic detection of atomic-set-serializability violations", in *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pp. 231–240, 2008.

[22] https://github.com/anonyau0321/David

[23] J. Huang, Q. Luo, and G. Rosu, "GPredict: generic predictive concurrency analysis", in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, pp. 847–857, 2015.

[24] Jikes RVM 3.1.3. http://www.jikesrvm.org/

[25] Jikes RVM Project Status. https://www.jikesrvm.org/ProjectStatus/

[26] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Commun. ACM* 21, 7 (July 1978), pp. 558–565, 1978.

[27] Z. Lai, S.C. Cheung, and W.K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*, pp. 235–244, 2010.

[28] R.J. Lipton, "Reduction: a method of proving properties of parallel programs", *Commun. ACM* 18, 12 (Dec. 1975), pp. 717–721, 1975.

[29] RegionTrack. https://github.com/LittleSnow321/RegionTrack-v

[30] S. Lu, S. Park, E. Seo, and Y. Zhou, "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 (ASPLOS '08)*, pp. 329–339, 2008.

[31] S. Lu, S. Park, E. Seo, and Y. Zhou, "Finding atomicity-violation bugs through unserializable interleaving testing", *IEEE Transactions on Software Engineering (TSE)*, 38(4), pp. 844–860, 2012.

[32] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "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 '06)*, pp. 37–48, 2006.

[33] X. Ma, S. Wu, E. Pobee, X. Mei, H. Zhang, B. Jiang, and W.K. Chan, "RegionTrack: A Trace-Based Sound and Complete Checker to Debug Transactional Atomicity Violations and Non-Serializable Traces", *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 7, 49 pages, 2021.

[34] U. Mathur and M. Viswanathan, "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)*, pp. 183–199, 2020.

[35] F. Mattern, "Virtual time and global states of distributed systems", in *Parallel and Distributed Algorithms*, pp. 215–226, 1988.

[36] M.D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. F. Salmi, S. Biswas, A. Sengupta, and J. Huang, "OCTET: capturing and controlling cross-thread dependences efficiently", in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA '13)*, pp. 693–712, 2013.

[37] S. Park, S. Lu, and Y. Zhou, "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* (*ASPLOS '09*), pp. 25–36, 2009.

[38] E. Pozniansky and A. Schuster, "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 (PPoPP '03)*, pp. 179–190, 2003.

[39] C.V. Praun and T. R. Gross, "Static detection of atomicity violations in object-oriented programs", *Journal of Object Technology*, 3(2), pp. 1–12, 2004.

[40] Q. Shi, J. Huang, Z. Chen and B. Xu, "Verifying synchronization for atomicity violation fixing", *IEEE Transactions on Software Engineering (TSE)*, 42(3), pp. 280–296, 2016.

[41] A. Sinha, S. Malik, C. Wang and A. Gupta, "Predictive analysis for detecting serializability violations through trace segmentation", in *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign (MEMO-CODE '11)*, pp. 99–108, 2011.

[42] L. A. Smith, J. M. Bull, and J. Obdrzálek, "A parallel java grande benchmark suite", in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC '01)*, pp. 8, 2001.

[43] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: weaving threads to expose atomicity violations", in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10)*, pp. 37–46, 2010.

[44] L. Wang and S. D. Stoller, "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 (PPoPP '05)*, pp. 61–71, 2005.

[45] S. Wu, C. Yang, and W.K. Chan, "ASR: Abstraction subspace reduction for exposing atomicity violation bugs in multithreaded programs", in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*, pp. 272–281, 2015.

[46] S. Wu, C. Yang, and W.K. Chan, "ASP: Abstraction subspace partitioning for detection of atomicity violations with an empirical study", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(3), pp. 724–734, 2016.

[47] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs", in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*, pp. 1–14, 2005.

[48] Z. Sun, R. Zeng and X. He, "A Method for Predicting Two-Variable Atomicity Violations", in Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS '18), pp. 103–110, 2018.

[49] J. Zhou, S. Silvestro, H. Liu, Y. Cai and T. Liu, "UNDEAD: Detecting and preventing deadlocks in production software", in *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE'17*), pp. 729-740, 2017.

[50] J. Yi, C. Sadowski, and C. Flanagan, "SideTrack: generalizing dynamic atomicity analysis", in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (*PADTAD '09*), pp. 1–10, 2009.

[51] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations", *in 2008 International Symposium on Computer Architecture*, pp. 277-288, 2008.
[52] C. S. Park, and K. Sen, "Randomized active atomicity violation detection in concurrent programs", *in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE'08)*, pp. 135-145, 2008.
[53] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language", *in Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*, pp. 334–345, 2006.
[54] C. v. Praun and T.R. Gross, 2003. "Static Conflict Analysisfor Multi-threaded Object-oriented Programs" in *Proceedings of the 2003 ACM SIGPLAN conference on Programming language design and implementation (PLDI '03)*, pp. 115–128.
[55] https://zenodo.org/record/3605759#.YjYaQRNBxQI
[56] https://github.com/JikesRvm/JikesRVM
[57] B. Çirisci, C. Enea, A. Farzan and SO Mutluergil, "Root Causing Linearizability Violations", in *Proceedings of International Conference on Computer Aided Verification (CAV'20)*, pp. 350-375, 2020.
[58] A. Katsarakis, V. Gavrielatos, M.R. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol", in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, pp. 201–217, 2020.
[59] C. Peterson, D. Dechev, "An Efficient Dynamic Analysis Tool for Checking Durable Linearizability", in *Proceedings of 2021 International Conference on Code Quality (ICCQ)*, pp. 27-38, 2021.

XIAOXUE MA received the B.Eng. degree in telecommunication engineering with honor from the College of Physical Science and Technology, Central China Normal University (CCNU), China, in 2017. She is currently working toward the Ph.D. degree with the Department of Computer Science, City University of Hong Kong. Her current research interest includes dynamic program analysis. Her work has been published in TOSEM.

IMRAN ASHRAF is a Ph.D. student of Computer Science at the City University of Hong Kong. He received his B.S. (Computer & Information Sciences) degree from Pakistan Institute of Engineering and Applied Sciences (PIEAS) in 2011 with a Gold Medal Award. Currently, his research involves program analysis and security vulnerability detection in decentralized platforms such as Ethereum blockchains. His primary interests in security vulnerability detection techniques are fuzz testing, static analysis and symbolic execution.

W.K. Chan received all his BEng in Computer Engineering, an MPhil degree, and a Ph.D. degree from The University of Hong Kong. He was in the industry to manage and develop software applications for many years before returning to The University of Hong Kong to complete his Ph.D. study. Chan is an associate editor of Software Testing, Verification and Reliability. He has served as a member in many program/review committees of many software engineering and web services conferences. He co-chaired AITest'21, QRS'20, COMPSAC'20-21, and was a symposium chair of SETA'15-21.