Davida
A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations
Ma, Xiaoxue; Ashraf, Imran; Chan, W. K.

Published in:
IEEE Transactions on Reliability

Published: 01/06/2023

Document Version:
Post-print, also known as Accepted Author Manuscript, Peer-reviewed or Author Final version

Publication record in CityU Scholars:
Go to record

Published version (DOI):
10.1109/TR.2022.3176680

Publication details:

Citing this paper
Please note that where the full-text provided on CityU Scholars is the Post-print version (also known as Accepted Author Manuscript, Peer-reviewed or Author Final version), it may differ from the Final Published version. When citing, ensure that you check and use the publisher’s definitive version for pagination and other details.

General rights
Copyright for the publications made accessible via the CityU Scholars portal is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights. Users may not further distribute the material or use it for any profit-making activity or commercial gain.

Publisher permission
Permission for previously published items are in accordance with publisher’s copyright policies sourced from the SHERPA RoMEO database. Links to full text versions (either Published or Post-print) are only available if corresponding publishers allow open access.

Take down policy
Contact lbscholars@cityu.edu.hk if you believe that this document breaches copyright and provide us with details. We will remove access to the work immediately and investigate your claim.
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 \((e_a, e_b)\). 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 \(tx_a \rightarrow tx_b\) from one transaction \(tx_a\) to another transaction \(tx_b\) is due to the pair of conflicting events of a variable-level dependency executed by \(tx_a\) and \(tx_b\), respectively [3][16]. In a dependency sequence \(\theta\), 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 \(\theta\) 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: {xiaoxuem2-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).
stance is the active transaction of only ignore these possibleency sequence with respect to formed before a longer one; otherwise, \( \theta \) is a cyclic sequence involving two transactions only. It 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 two insights. (Insight II) A dependency sequence \( \theta \) can become a witness for an atomicity violation on a transaction \( tx \) only if a shorter prefix of \( \theta \) is formed before a longer one; otherwise, \( \theta \) will not be increasing with respect to \( tx \). Moreover, suppose \( \theta \) is an increasing dependency sequence starting from \( tx \) and \( t \) is a dependency. In this case, if \( \theta' (\tau) \) is non-increasing, then \( \tau \) should be irrelevant to any possible atomicity violation on \( tx \). As such, Davida can safely ignore these dependencies without losing the precision. (Insight I) An atomicity violation on \( tx \) triggered by its event \( e_0 \) only when \( tx \) generates \( e_0 \). Thus, \( tx \) should be an active transaction when generating \( e_0 \). 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 \( HBT_i \) be \( \delta_i \). Suppose \( \tau = tx_i \Rightarrow tx_j \) is a dependency, where transactions \( tx_i \) and \( tx_j \) are executed by threads \( u \) and \( v \), respectively. Davida works as follows:

- \( HBT_i \) checks whether \( tx_i \) is reachable from its root. If this is the case, it reports an atomicity violation on \( tx_i \) with the sequence of edges from the root \( \delta_i \) to \( tx_i \), appended with \( \tau \) as the witness.
- Each other HBT instance, including \( HBT_u \) and other HBT instances, makes its own decision. It will append \( \tau \) 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 \( m_3 \),
Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations

<table>
<thead>
<tr>
<th>Thread $t_1$</th>
<th>Thread $t_2$</th>
<th>Thread $t_3$</th>
<th>Thread $t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1 {}$</td>
<td>$m_2 {}$</td>
<td>$m_3 {}$</td>
<td>$m_4 {}$</td>
</tr>
<tr>
<td>$p = 1$</td>
<td>$x = 1$</td>
<td>$z = 1$</td>
<td>$s = 0$</td>
</tr>
<tr>
<td>$q = 0$</td>
<td></td>
<td></td>
<td>$x = s$</td>
</tr>
<tr>
<td></td>
<td>$y = 0$</td>
<td>$t = y + z$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p = t + 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q = t + 2$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. An exemplified program $p_1$ $m_3$ followed by $m_5$, and $m_4$, respectively.

B. Preliminaries and Illustrations

Each event has an operation type: (1) $rt(t, x)$ 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 $\alpha$ is a sequence of events: $\alpha = e_1, e_2, ..., e_n$. Each event has an operation type: (1) $acq$ or $release$ the same lock. (2) $begin$ and $end$: marking the begin and end of an atomic region executed by a thread. A trace $\alpha$ is well-formed [34]: all lock acquire and release events are well-matched, a lock cannot be acquired or released by more than one thread at any time, and all begin and end events are well-matched.

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

A (regular) transaction $tx = (tx.begin, ..., e_i, ..., tx.end)$ 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 $(tx_1, ..., tx_i, tx_{i+1}, ..., tx_n)$ 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 $T(e)$ and $T(tx)$, respectively. A containment relation is denoted by $e_i \in e_j$. For two transactions $tx_i$ and $tx_j$ of the same thread $t$, if $tx_i$ is executed before $tx_j$, then $tx_i$ follows $tx_j$ in the program order.

Illustration. In $\alpha_1$, 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 $tx_5$ and $tx_3$ and $tx_6$ follow the program order. Transactions $tx_1$ to $tx_4$ are execution instances of atomic regions $m_1$ to $m_6$. To simplify the presentation, we only illustrate the accesses on shared variables in $\alpha_1$.

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 $\delta_t$.

Illustration. We assume that right after event $e_1$ has been executed, transaction $tx_2$ is completed, and $tx_3$ is completed before the execution of $e_4$. After executing $\alpha_1$’s prefix $(e_1, e_2, e_3, e_4, e_5, e_6)$, transactions $tx_1$, $tx_4$, $tx_3$ and $tx_5$ are still active.

For two events $e$ and $e'$ in a trace $\alpha$, $e$ 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 $e \triangleright e'$. HB relation (e.g., $e$ happens before $e'$) is the transitive closure of HB dependencies in $\alpha$. Each HB dependency $e_i \triangleright e_j$ corresponds to a dependency $tx_i \triangleright tx_j$ at the transaction level where $tx_i$ and $tx_j$ are transactions, $e_i \in tx_i$ and $e_j \in tx_j$ and $T(tx_i) \neq T(tx_j)$. $tx_i \triangleright tx_j$ can be formed by different HB dependencies.

Illustration. In trace $\alpha_1$ of Fig. 2, events $e_1$, $e_3$, $e_4$, $e_5$, $e_7$, $e_8$ and $e_{10}$ conflict with events $e_2$, $e_5$, $e_8$, $e_9$, $e_{11}$, $e_6$ and $e_{12}$, respectively, which produces seven cross-thread happens-before dependencies (i.e., $e_1 \triangleright e_2$, $e_3 \triangleright e_4$, $e_4 \triangleright e_5$, $e_5 \triangleright e_6$, $e_6 \triangleright e_7$, $e_7 \triangleright e_11$, $e_8 \triangleright e_9$, and $e_{10} \triangleright e_{12}$). These variable-level dependencies correspond to seven cross-thread dependencies at the transaction-level: $t_{11} = tx_4 \triangleright tx_2$, $t_{12} = tx_6 \triangleright tx_5$, $t_{13} = tx_2 \triangleright tx_5$, $t_{14} = tx_1 \triangleright tx_6$, $t_{15} = tx_6 \triangleright tx_5$, $t_{16} = tx_5 \triangleright tx_4$ and $t_{17} = tx_5 \triangleright tx_1$. 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 $t_{17}$) for dependency $t_{17}$ is executed.

A dependency sequence $\theta = tx_1 \triangleright tx_n$ is a non-empty sequence of dependencies at the transaction level and is formed by an underlying event sequence $S$. For all $i$, $0[i]$ refers to the $i$-th dependency in $\theta$. $0[i].source$ and $0[i].sink$ refer to the transactions at the head and tail positions of the dependency $\theta[i] = tx_{source} \triangleright tx_{sink}$, respectively. For each sequence $\theta$, $0[i].sink$ and $0[i+1].source$ should be executed by the same thread. The sequence $\theta$ is increasing, denoted as $tx_{sink} \triangleright tx_{source}$, if for all $j < k$, $tx_{sink} \triangleright tx_{source}$.
the event $S[j]$ either happens before the event $S[k]$ or is equal to that event. If $\theta$ contains exactly one dependency, it is increasing, $\theta$ is **non-increasing** if it is not increasing. Each dependency $0[i]$ as well as $0[i].source$ and $0[i].sink$ along the sequence $0$ is **reachable** from $tx_i$ in the sequence.

A cyclic dependency sequence $tx_1 \rightarrow\rightarrow tx_i$ is an increasing dependency sequence $tx_1 \rightarrow\rightarrow tx_i$ appended with $tx_i \rightarrow\rightarrow tx_1$.

When a cross-thread dependency $tx_i \rightarrow\rightarrow tx_i$ is generated by the HB dependency $e_c \rightarrow e_m$, where $e_c \in tx_i$ and $e_m \in tx_i$, an **atomicity violation** (or AV for short) [16][33] on transaction $tx_1$ is triggered at event $e_m$ if the cyclic dependency sequence $tx_1 \rightarrow\rightarrow tx_i$ is formed. We refer to $tx_1 \rightarrow\rightarrow tx_i$ 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 $tx_i$ and $tx_i \rightarrow\rightarrow tx_1$ 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: $c_1 = (t_{i1}, t_{i2}, t_{i3}), c_2 = (t_{i1}, t_{i3}, t_{i4}), c_3 = (t_{i1}, t_{i5}, t_{i6}), c_4 = (t_{i4}, t_{i2}, t_{i2}), c_5 = (t_{i4}, t_{i5}, t_{i7})$.

The event sequence $s_1$ underlying cycle $c_1$ is $(e_1, e_2, e_3, e_4, e_5, e_7, e_1)$, 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 $tx_1$. So, event $e_1$ triggers an atomicity violation on transaction $tx_1$. Cycle $c_1$ is a witness for the atomicity violation on $tx_1$. Specifically, in Fig. 1, witness $c_1$ indicates the execution instances of the atomic regions in the sequence $(m_{a1}@t_1, m_{a2}@t_2, m_{a3}@t_3, m_{a4}@t_4)$ are inter-dependent to trigger the atomicity violation on the execution instance of $m_a$.

Cycle $c_2$ is another witness for the atomicity violation on $tx_1$. It represents $(m_{b1}@t_4, m_{b2}@t_3, m_{b3}@t_2, m_{b4}@t_1)$.

Cycle $c_3$ is also increasing and is a witness for the atomicity violation on transaction $tx_1$, indicating $(m_{a1}@t_2, m_{a2}@t_4, m_{a3}@t_1, m_{a4}@t_1)$.

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 $(t_{i5}, t_{i6})$ and $(t_{i4}, t_{i2})$ are non-increasing.) So, $c_3$ and $c_4$ do not indicate atomicity violations on $tx_1$ 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 $tx.begin \rightarrow e_c$.

### 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 $a_1$, Velodrome and DoubleChecker construct a graph as shown in Fig. 2 but with $tx_1$ removed, because they only keep one dependency between the same two transactions. Specifically, on processing a dependency $\tau = tx_1 \rightarrow tx_2$ if the graph already contains a dependency from $tx_1$ to $tx_2$, Velodrome and DoubleChecker will not add $\tau$ 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., $t_{i1} = (tx_4, 1) \rightarrow (tx_3, 2)$). (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 $tx_6$ in $a_1$, Velodrome and DoubleChecker both find $tx_4$ in the graph already containing a dependency from $tx_4$ to some transactions. Starting from $t_{i1}$ 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 $t_{i1}$ is smaller than that at the head position of $t_{i3}$. Velodrome also reports $tx_4$ incurring an atomicity violation and $c_2$ being a witness.

Then, on processing $t_{i7}$, Velodrome and DoubleChecker find the graph containing a dependency from $tx_1$ to some other transaction. Starting from $t_{i4}$ in the graph, they locate the dependency sequence $c_4$. As DoubleChecker detects cycle $c_4$, it directly reports $tx_4$ that completes the cycle incurring an atomicity violation and $c_4$ being a witness. However, since $c_4$ is non-increase-
Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations

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 $t_12$ is increasing with respect to $tx_4$ (via $t_11$) but non-increasing with respect to $tx_1$ (via $t_14$). Determining $t_12$ as either increasing or non-increasing in a path-insensitive manner is infeasible. (Similarly, with respect to $tx_4$, $t_16$ is increasing following one path (via $t_12$) but non-increasing following another path (via $t_13$)). 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 $a_1$, 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 $tx_{16}$ 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 $t_16$ in $a_1$, after locating dependency $t_{11}$, it finds that $tx_2$ involves in some dependency with transactions of $t_3$, and then locates $t_{13}$. 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 $c_1$ includes execution instances of atomic regions $m_4$, $m_5$ and $m_6$, and witness $c_2$ includes execution instances of $m_4$, $m_5$, $m_6$ and $m_7$. 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 $t_{13}$ 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 $t_{12}$ and $t_{13}$ 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
view when debugging an atomicity violation.

Since RegionTrack discards all dependencies after its processing, it can only report the dependencies \( t_{16} \) and \( t_{17} \) 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 \( c_3 \)) 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 \( HBT \), for thread \( t \) is a rooted and directed tree \( (V, E) \), 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 \( HBT \) as the active transaction \( dt \) of thread \( t \) at the moment of \( HBT \) assigning with a new root node. Each path from the root \( dt \) to a node \( tx_n \) via an edge \( \tau = tx_a \rightarrow tx_b \) of the tree represents an increasing dependency sequence in the trace (i.e., \( \tilde{\tau} \rightarrow \tau \)).

We simply say that \( dt \) reaches \( tx_b \) and \( \tau \), or \( tx_a \in HBT \), and \( \tau \in HBT \), respectively, for short. (Note that both \( tx_b \) and \( \tau \) are contained in \( HBT_\tau \).) On the contrary, if a transaction \( tx_c \) is not a node in \( HBT \), we say that both \( HBT \), and \( dt \) do not reach \( tx_c \), denoted by \( tx_c \notin HBT \).

Suppose \( tx_a, tx_b \in HBT \), and \( tx_c \notin HBT \); are two transactions such that \( tx_a \) happens before \( tx_b \) by the program order. In this case, we have: \( dt \) happens before \( tx_a \) and \( tx_a \) happens before \( tx_c \), indicating that \( dt \) happens before \( tx_c \). We say that \( tx_c \) is reachable from \( HBT \), denoted by \( HBT \models tx_c \). Similarly, if \( tx_a \notin HBT \) and there is no such \( tx_a \in HBT \) such that \( tx_a \) happens before \( tx_c \), \( tx_c \) is said to be unreachable from both \( HBT \), and \( dt \), denoted by \( HBT \models tx_c \).

To simplify our subsequent presentation further, for brevity, if \( tx_c \notin HBT \), we also say \( tx_c \) is reachable from \( HBT \). Moreover, in case we need to be specific to \( tx_c \) containing in \( HBT \), we will use the notation \( tx_c \in HBT \), and state it as \( dt \) reaching \( tx_c \).

Davida maintains a transactional happens-before forest \( F = \{(HBT_t, M_t) | t \in Tid\} \), 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 \( dt \). 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, \( M_t(t') \notin HBT_t \).

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 \( dt \) yet. We simply refer to the entry \( M_t(t') \) with an assigned value as defined, otherwise undefined.

To efficiently determine whether or not a transaction \( tx_c \) of thread \( t' \) is reachable from \( dt \), 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 \( dt \) is still unable to reach any transaction of \( t' \) yet, and so, \( tx_c \) is unreachable from \( dt \).

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 \( du \), \( dv \), and \( dw \) of threads \( u \), \( v \), and \( w \). Threads \( u \) and \( v \) execute \( tx_u \) and \( tx_v \), respectively. On processing an event \( e_0 \in tx_u \), a cross-thread dependency \( \tau = tx_u \rightarrow tx_v \) (denoted as \( \tau \) triggered by the happens-before dependency \( e_0 \Rightarrow e_0 \) is generated, as depicted in Fig. 3(c), where event \( e_0 \in tx_u \) appears earlier than \( e_0 \) in the trace. For ease of presentation, we refer to \( tx_u \) and \( tx_v \) as the source and sink transactions of \( \tau \), denoted as \( t.source = tx_u \) and \( t.sink = tx_v \), 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 \( a_1 \) in Fig. 2 to maintain the increasing dependency sequences from each transaction while it is active. Davida processes dependency sequence \( \{t_{11} = tx_4 \rightarrow tx_2, t_{12} = tx_6 \rightarrow tx_3, t_{13} = tx_2 \rightarrow tx_5, t_{14} = tx_1 \rightarrow tx_6, t_{15} = tx_6 \rightarrow tx_3, t_{16} = tx_5 \rightarrow tx_4 \text{ and } t_{17} = tx_3 \rightarrow tx_4\} \) in turn. On processing the beginning event of \( tx_4 \), the HBT instance \( HBT_{a_1} \) is set to empty. The root \( dt_{a_1} \) of \( HBT_{a_1} \) is set
to \(tx_4\). On processing \(t_{11}\), Davida adds \(t_{11}.sink\) (i.e., \(tx_2\)) to \(HBT_{tx}\) as a child of \(\delta_4\) and updates \(M_{s}(t_{4})\) to \(tx_2\). The parent-child edge from \(\delta_4\) to \(tx_2\) represents \(t_{11}\). Next, Davida processes \(t_{12}\). At this moment, \(M_{s}(t_{2})\) is mapped to \(tx_2\) and \(M_{s}(t_{4})\) is undefined. Since \(t_{12}.source\) (i.e., \(tx_6\)) follows \(tx_2\) by the program order and \(\{t_{11}, t_{12}\}\) increases, Davida adds \(tx_6\) to \(HBT_{tx}\) as a child of \(tx_2\) and adds \(tx_5\) to \(HBT_{tx}\) as a child of \(tx_6\). Davida also updates \(M_{s}(t_{2})\) to \(tx_5\). Then, on processing \(t_{13}\), since \(M_{s}(t_{3})\) is mapped to \(tx_5\) and \(M_{s}(t_{4})\) is undefined, \(t_{14}\) is irrelevant to any possible atomicity violation that might occur on \(\delta_4\). Thus, Davida does not capture \(t_{14}\) into \(HBT_{tx}\). Then, on processing \(t_{15}\), because \(M_{s}(t_{3})\) is already mapped to \(tx_5\) indicating that \(HBT_{tx}\) reaches \(t_{15}.sink\) (i.e., \(tx_5\)) already. So, Davida does not capture \(t_{15}\) into \(HBT_{tx}\). When handling \(tx_6\), Davida does not capture \(tx_6\) into \(HBT_{tx}\) because \(t_{16}.sink\) is \(\delta_4\). On processing \(t_{17}\), Davida captures \(t_{17}\) into \(HBT_{tx}\) by appending it after \(t_{12}\).

Similarly, Davida captures dependencies \(t_{14}\) and \(t_{15}\) into \(HBT_{tx}\). 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 \(t_{12}\) is generated, \(tx_2\) has been completed. Thus, Davida captures no dependency into both \(HBT_{tx}\) and \(HBT_{tx}\) during the executions of \(tx_2\) and \(tx_3\). After processing \(t_{11}\), two transactions \(tx_2\) and \(tx_5\) start. Davida clears \(HBT_{tx}\) and \(HBT_{tx}\). Then, \(\delta_2\) and \(\delta_3\) are updated to \(tx_6\) and \(tx_5\), respectively. Davida captures dependencies \(t_{12}, t_{16}\) and \(t_{17}\) in \(HBT_{tx}\). Similarly, Davida also captures dependencies \(t_{16}\) and \(t_{17}\) in \(HBT_{tx}\). Fig. 5 shows the forest maintained by Davida in \(\alpha_{tx}\) when \(e_{11}\) executes.

2) Our Design: Algorithm 1 shows the main algorithm of Davida. It consists of three components: initialization, maintenance of \(HBT_{tx}\), and maintenance of every \(HBT_{tx}\) where \(w \neq u \lor v\) when processing dependency \(\tau = tx_{j} \Rightarrow tx_{i}\).

Initialization: Whenever a thread, say \(t\), starts an active transaction \(\delta\), Davida calls initialization at line 2. Line 3 clears the HBT instance \(HBT_{tx}\) 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 \(\delta_i\) as the root of \(HBT_{tx}\) at line 4. It clears the map \(M_{s}\) and sets \(M_{s}(t)\) to \(\delta_i\) at lines 5-6.

Maintenance of \(HBT_{tx}\): When a dependency \(\tau = tx_{j} \Rightarrow tx_{i}\) is generated at the moment of an event \(e_{j} \in tx_{j}\) being generated, the HBT instance \(HBT_{tx}\) of thread \(u\) checks whether \(\tau\) should be added to \(HBT_{tx}\) at lines 9-12. \(HBT_{tx}\) first checks whether the following two conditions hold at line 9: (Condition C1) \(tx_{j}\) is the active transaction \(\delta_{t}\) of thread \(u\); and (Condition C2) \(HBT_{tx}\) does not reach any transaction of thread \(v\) which is checked by examining whether \(M_{s}(v)\) is defined. As shown in Fig. 4, if both conditions hold, \(HBT_{tx}\) adds \(tx_{j}\) as a child of \(tx_{i}\) at line 10 to capture \(\tau\). It also updates \(M_{s}(v)\) to \(tx_{j}\) at line 11 to indicate that \(tx_{j}\) is the transaction of thread \(v\) with the smallest timestamp kept in \(HBT_{tx}\). This entry \(M_{s}(v)\) indicates that \(\tau\) is “ready for extension” to form a longer increasingly dependency sequence with respect to the root.

The two conditions (C1) and (C2) are specially designed to eliminate cases where Davida can skip the dependency maintenance of \(HBT_{tx}\) 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 \(\tau\) is feasible only when \(tx_{j}\) is active. The status of \(tx_{j}\) at the moment of the event \(e_{j} \in tx_{j}\) 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 \(r\), this dependency \(\tau\) can be safely skipped by \(HBT_{tx}\). 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 \(HBT_{tx}\) is \(tx_{i}\)). Without loss of generality, suppose further that \(M_{s}(v)\) maps to a transaction \(tx_{j}\) of thread \(v\). (Note that the thread executing \(tx_{j}\) is \(v\) as well.) In this case, we have \(tx_{j} \in HBT_{tx}\), as illustrated in Fig. 6. There are two sub-cases to consider. First, suppose \(tx_{j} \neq tx_{i}\). In this case, \(tx_{j}\) happens before \(tx_{i}\) (i.e., \(tx_{j}\) is already reachable from \(HBT_{tx}\)). Second, suppose \(tx_{j} = tx_{i}\). In this case, \(tx_{j}\) should already be contained in \(HBT_{tx}\). So, in either case, there is no need to further maintain \(HBT_{tx}\) to capture \(\tau\). 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 \(\tau\) into \(HBT_{tx}\) by adding \(tx_{j}\) as a child of \(tx_{i}\) at line 10.

Maintenance of every \(HBT_{tx}\) where \(w \neq u \lor v\): 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 \(t_{15}\), \(t_{15}\) is appended to \(t_{14}\) that is already kept in \(HBT_{tx}\). To do so, Algorithm 1 calls checkOtherTrees at lines 13-14. The function checkOtherTrees extends the increasing
dependency sequences maintained in HBT\(_w\) at lines 16-31.

**TABLE II. Case Analysis of Capturing Dependency \(r\) into HBT\(_w\)**

<table>
<thead>
<tr>
<th>(M_u(u))</th>
<th>(M_v(v))</th>
<th>(DS_1)</th>
<th>(DS_2)</th>
<th>(DS_3)</th>
<th>(DS_4)</th>
<th>(DS_5)</th>
<th>(DS_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>defined</td>
<td>defined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>defined</td>
<td>undefined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>undefined</td>
<td>defined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>undefined</td>
<td>undefined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Case (1) to Case (8) | (9): skip | (10): capture | (11): infeasible |

\(DS = HBT_w \lor tx \); \(DS - HBT_w \lor tx\)

In **TABLE II**, there are **12 infeasible** cases as shown in Fig. 7(a)-(e): \(2 \cdots 4 \cdots 5 \cdots 7 \cdots 9 \cdots 11 \cdots 12 \cdots 13 \cdots 14 \cdots 15\). Each such case satisfies at least one pair of self-contradicting conditions: (1) \(M_u(u)\) is defined and HBT\(_w\) \(\not\supseteq tx\), (2) \(M_u(u)\) is undefined and HBT\(_w\) \(\supseteq tx\), (3) \(M_v(v)\) is defined and HBT\(_w\) \(\not\supseteq tx\), (4) \(M_u(u)\) is undefined and HBT\(_w\) \(\supseteq tx\). Each of cases (1) and (3) is self-contradictory because when \(M_u(u)\) and \(M_v(v)\) are defined, we must have \(M_u(u) \in HBT_w\) and \(M_v(v) \in HBT_w\), respectively. Since \(M_u(u)\) and \(M_v(v)\) must have timestamps not larger than \(tv\) and \(tv\) respectively, so, \(tv\) and \(tv\) must be reachable from \(\delta_w\) (i.e., HBT\(_w\) \(\supseteq tv\) and HBT\(_w\) \(\supseteq tv\) must be held), respectively, contradicting to HBT\(_w\) \(\not\supseteq tv\) and HBT\(_w\) \(\not\supseteq tv\), respectively. Each of cases (2) and (4) is self-contradictory because when \(M_u(u)\) and \(M_v(v)\) are undefined, \(\delta_w\) could not reach any transaction of threads \(u\) and \(v\), respectively. It is infeasible for \(tv\) and \(tv\) reachable from \(\delta_w\), contradicting to the given conditions of HBT\(_w\) \(\supseteq tv\) and HBT\(_w\) \(\supseteq tv\), respectively.

In **TABLE II**, there are three cases to skip \(r = tx \supseteq tx\) from being captured into HBT\(_w\): \(1 \cdots 11 \cdots 16\). In cases \(1 \cdots 11\), the condition HBT\(_w\) \(\supseteq tx\) holds already. Specifically, \(M_u(u)\) is defined and \(tx\) is reachable from HBT\(_w\). Note that \(M_u(u)\) can be defined (i.e., \(1\)) or undefined (i.e., \(11\)). There are two sub-cases to consider. First, suppose that \(M_u(u)\) maps to a transaction \(tx\) of thread \(v\). In this case, we should have \(tx\) \(\in HBT_w\) and \(tx\) happens before \(tv\) due to the program order of thread \(v\). Thus, the dependence sequence \(\delta_v \supseteq tx\) is captured in HBT\(_w\) and \(\delta_v \supseteq tx\) is formed by appending \(tx \supseteq tx\) to \(\delta_v \supseteq tx\). Second, suppose that \(M_v(v)\) maps to \(tx\) such that \(tx \in HBT_w\). The dependence sequence \(\delta_v \supseteq tx\) has already been kept in HBT\(_w\). In either sub-case, HBT\(_w\) needs not to further capture \(r\) to form HBT\(_w\) \(\supseteq tx\) as illustrated in Fig. 7(f).

In case \(16\), all the four conditions are not satisfied. \(r\) is irrelevant to \(\delta_w\) because HBT\(_w\) \(\not\supseteq tx\) holds and \(tx\) is not reachable from any sequence captured in HBT\(_w\). So, \(r\) cannot be appended to any sequence captured in HBT\(_w\) to form \(\delta_w \supseteq tx\). In other words, \(\delta_w \supseteq tx\) cannot be formed even with the presence of \(r\). So, as illustrated in Fig. 7(g), \(r\) should not be captured into HBT\(_w\).

In **TABLE II**, there is **only one** case that HBT\(_w\) cannot exclude \(r\) from capturing for the detection of atomicity violation on \(\delta_w\) and its witness. Specifically, in case \(6\), \(M_u(u)\) is defined, \(M_v(v)\) is undefined, and we have HBT\(_w\) \(\supseteq tv\) and HBT\(_w\) \(\not\supseteq tv\). The function checkOtherTrees is specially designed to precisely handle this case at line 17 in Algorithm 1. Let \(tx\) be a transaction and \(tv\) \(\in HBT_w\) so that \(tx\) and \(tv\) are executed by the same thread \(u\). Also, let \(e_v \in tv\) be the last event in the event sequence underlying the dependence sequence \(\delta_u \supseteq tx\) at line 19, and \(e_u \in \epsilon_u\) be the event of \(tx\) underlying the dependence \(\tau\). There are two-subcases to consider, depending on whether \(e_v\) happens before \(e_u\) due to program order of thread \(u\). Suppose \(e_v\) happens before \(e_u\). (This scenario is depicted in Fig. 7(i)). Since \(tx\) is reachable from HBT\(_w\), thus, appending \(r\) to \(\delta_u \supseteq tx\) is also increasing, which is ensured by line 20 in the algorithm. So, HBT\(_w\) appends \(r\) the node \(tx\) to capture \(\delta_u \supseteq tx\) at lines 22 and 25-26 to make \(tx\) reachable from HBT\(_w\). Then the algorithm updates \(M_u(u)\) from undefined to \(tx\) at lines 23 and 27 to reflect that HBT\(_w\) can reach some transaction (i.e., \(tx\)) of thread \(v\). (Note that after the maintenance, we have HBT\(_w\) \(\supseteq tx\).) Next, suppose \(e_u\) does not happen before \(e_v\). As depicted in Fig. 7(h), since \(e_v\), \(e_u\) does not follow their temporal order in the trace, the dependence sequence \(\delta_u \supseteq tx\) appended with \(\langle r\rangle\) is non-increasing. Thus, HBT\(_w\) will not capture \(r\) because \(\delta_u \supseteq tx\) appended with \(\langle r\rangle\) will never trigger an atomicity violation on \(\delta_w\).
Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations

As a result, \( HBT_w \) soundly captures a reduced set of increasing dependency sequences reachable from \( \delta_w \), which can infer any increasing dependency sequences reachable from \( \delta_w \).

B. Detecting Atomicity Violations and Localizing Witnesses

1) Illustration: In trace \( \gamma \) in Fig. 2, when dependency \( t_{6} \) appears, \( M_d(t_6) \) has been mapped to \( t_{6}\text{.source} \) (i.e., \( t_{6} \)), and the beginning event \( t_{6}\text{.begin} \) happens before \( e_{7} \). An atomicity violation on \( t_{4} \) is detected. At this moment, \( HBT_d \) is \( \{(t_{4}, t_{2}, t_{6}, t_{3}), (t_{1}, t_{2})\} \) and the root of \( HBT_d \) is \( t_{4} \). To localize the witness, \( HBT_d \) attempts to reproduce a path \( p_1 \) for the increasing dependency sequence \( t_{4} \iff t_{5} \) appended with \( t_{6} \), it starts from adding the sink transaction of \( t_{6} \) (i.e., \( t_{6}\text{.sink} \) to the path and retrieving the transaction at the entry \( M_d(t_6) \) (i.e., \( t_{3} \)). \( t_{4} \) and \( t_{5} \) are added to path \( p_1 = \{t_{6}, t_{5}, t_{4}\} \). \( t_{5} \) is set to be the currently visiting node. Then, Davida finds the parent node of \( t_{5} \) which is \( t_{6} \), adds it to path \( p_1 = \{t_{6}, t_{5}, t_{4}\} \) and sets it as the currently visiting node. Similarly, Davida adds \( t_{2} \) and \( t_{4} \) to \( p_1 \) and \( t_{2} = \{t_{4}, t_{2}, t_{6}, t_{5}, t_{4}\} \) in turn. Since \( t_{5} \) is the root node of \( HBT_d \), the witness localization ends. Davida reproduces a cyclic dependency sequence as the corresponding witness \( p_1 = \{t_{4}, t_{2}, t_{6}, t_{5}, t_{4}\} \), 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 \( r = t_{8}\text{.omega} \iff t_{7}\text{.omega} \) produced by the underlying HB dependency \( e_{6} \iff e_{7} \). Recall that \( t_{7} \) is the active transaction of \( v \), which is \( \delta_v \). \( HBT_d \) determines whether the two conditions hold: (Condition D1) \( t_{6} \) is reachable from \( \delta_v \) (i.e., the increasing dependence sequence \( t_{4} \iff t_{5} \) can be formed by \( HBT_d \)); (Condition D2) \( t_{6} \iff t_{5} \) appended with \( t_{6} \) forms a cyclic dependency sequence \( t_{6} \iff t_{5} \), representing an atomicity violation on \( t_{6} \).

There are four possible combinations of the two conditions (D1) and (D2). However, (D2) is satisfiable only when (D1) is satisfied because if \( t_{6} \iff t_{5} \) does not exist in \( HBT_d \), there is no need to append \( r \) after it. Therefore, Algorithm 2 first checks whether \( M(u) \) maps to any transaction of thread \( u \) at line 5.

Consider the case where \( M(u) \) is undefined. In this case, \( t_{6} \iff t_{5} \) will not exist in \( HBT_d \). Thus, the cyclic dependency sequence \( t_{6} \iff t_{5} \) cannot be formed, indicating no atomicity violation on \( t_{6} \). Next, consider the case where \( M(u) \) maps to a transaction of \( u \), denoted as \( t_{7} \). First, suppose that thread \( u \) executes \( t_{7} \) before \( t_{8} \) (i.e., \( t_{7} \) follows \( t_{8} \) by program order). In this case, \( t_{7} \) is not reachable from \( HBT_a \) (recall that \( \delta_v = t_{7} \)). So,
neither $tx_j \xrightarrow{+} tx_i$ can be formed, nor $tx_j \xrightarrow{-} tx_i$ can be formed, indicating no atomicity violation on $\delta_v$. Next, suppose that thread $u$ executes $tx_y$ before $tx_i$ (i.e., $tx_y \xrightarrow{+} tx_i$). In this case, $tx_i$ is reachable from $tx_y$ by appending $tx_y \xrightarrow{-} tx_i$ with $tx_y \xrightarrow{+} tx_i$. So, $\tau$ can be appended after $tx_y \xrightarrow{+} tx_i$ to form the cyclic sequence $tx_y \xrightarrow{-} tx_i$, indicating an atomicity violation on $tx_i$. Algorithm 2 handles this case at line 6 (i.e., conditions (D1) and (D2) are both satisfied). An atomicity violation on $tx_i$ is reported at line 7, and the witness $tx_y \xrightarrow{+} tx_i$ is localized at line 8. Last, suppose $tx_y = tx_i$. In this case, we have $tx_y \in HBT_v$. Let $e_p \in tx_y$ be the last event in the event sequence underlying $tx_y \xrightarrow{+} tx_i$. Since $tx_i$ is the first transaction in every increasing dependency sequence in $HBT_v$, the event $tx_i.begin$ must happen before $e_p$ and $(tx_i.begin, e_p)$ is increasing. If thread $u$ executes $e_o$ before $e_p$ and thus $(e_p, e_o)$ is non-increasing, then $(tx_i.begin, e_p, e_o)$ is also non-increasing and $tx_i.begin$ must not happen before $e_p$. So, $e_p$ and $e_o$ cannot be appended after the event sequence underlying $tx_y \xrightarrow{+} tx_i$ to form $tx_y \xrightarrow{-} tx_i$, indicating no atomicity violation on $tx_i$. On the other hand, if thread $u$ executes $e_p$ before $e_o$ (i.e., $(e_p, e_o)$ is increasing as depicted in Fig. 8), then $(tx_i.begin, e_p, e_o)$ is increasing and $tx_i.begin$ will also happen before $e_p$. So, $e_p$ and $e_o$ can be appended after the event sequence underlying $tx_y \xrightarrow{+} tx_i$ to form $tx_y \xrightarrow{-} tx_i$, indicating an atomicity violation on $tx_i$. Algorithm 2 handles this case at line 9 (i.e., conditions (D1) and (D2) are both satisfied). An atomicity violation on $tx_i$ is reported at line 10, and the witness $tx_i \xrightarrow{-} tx_i$ 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 $tx_o$ is the root of $HBT_v$. Suppose further that a dependency $\delta$ is increasing with respect to a dependency $\delta'$ already kept in $HBT_v$, where the thread executing $\tau.sink$ and $\tau'.source$ are the same. Then $\tau'$ will not involve in any witness that leads to atomicity violations on $tx_i$ in trace $\alpha$.

**Theorem 2.** Davida reports an atomicity violation on transaction $tx_y$ of thread $v$ in trace $\alpha$ if and only if an event $e_o \in tx_y$ and a cross-thread dependency $tx_o \xrightarrow{v} tx_y$ triggered by $e_o \xrightarrow{v} e_o$ are generated, where $e_o \in tx_i$, such that $tx_y \xrightarrow{+} tx_i$ formed by $HBT$, and appended with $tx_i \xrightarrow{-} tx_i$ construct a cyclic dependency sequence $tx_i \xrightarrow{+} tx_y$.

---

**Algorithm 2. Atomicity Violation Detection and Witness Localization**

1. **Input:** $\tau = tx_i \xrightarrow{v} tx_j \xrightarrow{-} tx_j \xrightarrow{+} tx_i$, a new cross-thread dependency by $e_o \xrightarrow{v} e_o$ where $\tau' = \tau_i \rightarrow \tau_j \rightarrow \tau_j \rightarrow \tau_i$ is currently visiting node
2. **Output:** $\text{witness} \leftarrow$ a cyclic dependency sequence on $\tau.sink$
3. **function** $\text{AVetation}(\tau, e_o)$
4. \begin{align*}
    & \text{let } v = \tau.sink, T(\tau) \\
    & tx_i = M(\mu) \\
    & \text{if } \text{isProgramOrder}(tx_y, tx_y) \text{ then} \\
    & \quad \text{report atomicity violation on } tx_i \\
    & \quad \text{witnessLocalization}(tx_i, \tau) \\
    & \text{else if } \text{isEqual}(tx_y, tx_y) \text{ and } tx_j.begin \rightarrow e_o, \text{then} \\
    & \quad \text{report atomicity violation on } tx_j \\
    & \quad \text{witnessLocalization}(tx_j, \tau) \\
    & \text{end if}
\end{align*}
5. **function** $\text{witnessLocalization}(tx_y, \tau)$
6. \begin{align*}
    & \text{witness.append}(tx_j) \\
    & \text{if } tx_i = \tau' \text{ then} \\
    & \quad \text{witness.append}(tx_i) \\
    & \text{else} \\
    & \quad \text{witness.append}(tx_j) \\
    & \quad \text{witness.append}(tx_i) \\
    & \text{end if}
\end{align*}
7. **while** true do
8. \begin{align*}
    & \text{current} = \text{HBT.v.getParent(current)} \\
    & \text{witness.append(current)} \\
    & \text{if } \text{current} = \tau \text{ then} \\
    & \quad \text{//find the root node, loop ends}
\end{align*}
9. \begin{align*}
    & \text{break}
\end{align*}
10. **end while

**Theorem 3.** Davida localizes a witness for an atomicity violation on transaction $tx_j$ of thread $v$ through $HBT_v$, in trace $\alpha$ if and only if an atomicity violation is reported on $tx_j$ when an event $e_o \in tx_j$ and a cross-thread dependency $tx_o \xrightarrow{v} tx_j$ triggered by $e_o \xrightarrow{v} e_o$ are generated in $\alpha$.

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 $\alpha$ triggered by an event $e_o \in tx_j$. Suppose that a cross-thread dependency $tx_o \xrightarrow{v} tx_j$ triggered by the variable-level dependency $e_o \xrightarrow{v} e_o$ is generated, where $e_o$ is an event of $tx_o$. Moreover, let the witnesses reported by Velodrome and Davida for the atomicity violation are $\tau_v$ and $\tau_d$, respectively. For any thread $w$ such that $w \neq v$, suppose that the projected transaction sequences of the witnesses $\tau_v$ and $\tau_d$ on thread $w$ are $cw_w = \{tx_{a}, ..., tx_{b}\}$ and $cwd_w = \{tx_{c}, ..., tx_{d}\}$, respectively, where $cw_v$ and $cwd_v$ are both non-empty. Note that all transactions along $cw_v$ and $cwd_v$ performed by $w$ are ordered according to the program order. Then, we should have (1) $tx_{C} \rightarrow tx_{A}$ or $tx_{A} = tx_{C}$, and (2) if $tx_{A} \rightarrow tx_{E}$,
Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicty Violations

then \(tx_b \Rightarrow^+ tx_c\) or \(tx_b = tx_c\). In other words, \(cvw\) is a subsequence of \(cwd\) (i.e., \(tx_c \Rightarrow^+ tx_a \Rightarrow^+ tx_b \Rightarrow^+ tx_c\)).

Proof. For the first condition, since the entry \(M(w)\) must have been mapped to \(tx_c\) and \(tx_c\) has the smallest timestamp among the transactions of the thread \(w\) captured in \(HBT_i\), \(tx_i\) is either \(tx_c\) or a transaction following \(tx_c\) by the program order. Suppose that \(tx_c\) follows \(tx_i\) by the program order (i.e., \(tx_i \Rightarrow^+ tx_c\)). Then, the increasing dependency sequence \(tx_i \Rightarrow^+ tx_a\) should have been constructed before \(tx_j \Rightarrow^+ tx_c\); \(M(w)\) should be mapped to \(tx_i\) with a smaller timestamp than \(tx_c\) in \(HBT_i\). The increasing dependency sequence \(tx_j \Rightarrow^+ tx_c\) should be formed by appending \(tx_i \Rightarrow^+ tx_c\) after \(tx_j \Rightarrow^+ tx_a\) using \(HBT_i\).

There is a contradiction. So, we must have either \(tx_c \Rightarrow^+ tx_i\) or \(tx_c = tx_i\).

For the second condition, as \(tx_b\) is the last transaction in \(cvw\), \(tx_b\) is the only transaction in \(cvw\) that can form the increasing dependency sequence \(tx_j \Rightarrow^+ tx_i\). On the other hand, \(tx_0\) is the transaction captured by \(HBT_i\) that has a dependency to transactions of a thread other than \(w\) and forms \(tx_j \Rightarrow^+ tx_i\). Note that the increasing dependency sequence \(tx_j \Rightarrow^+ tx_i\) via \(tx_0\) is constructed before that via \(tx_b\). Suppose that \(tx_b\) follows \(tx_0\) by the program order (i.e., \(tx_0 \Rightarrow^+ tx_b\)). Then, when traversing from the first transaction of the sequence to transaction \(tx_0\), Velodrome should follow the dependency sequence starting from \(tx_0\) to form \(tx_j \Rightarrow^+ tx_i\) rather than reporting the dependency sequence starting from \(tx_b\), resulting in a contradiction. As a result, if \(tx_i \Rightarrow^+ tx_0\), then either \(tx_b \Rightarrow^+ tx_0\) or \(tx_b = tx_0\). \(\Box\)

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>
<thead>
<tr>
<th>TABLE III: Descriptive Statistics of Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>eclipse</td>
</tr>
<tr>
<td>hsqldb</td>
</tr>
<tr>
<td>lusearch</td>
</tr>
<tr>
<td>xalan</td>
</tr>
<tr>
<td>avorao</td>
</tr>
<tr>
<td>lusearch</td>
</tr>
<tr>
<td>sunflow</td>
</tr>
<tr>
<td>xalan</td>
</tr>
<tr>
<td>crypt</td>
</tr>
<tr>
<td>series</td>
</tr>
<tr>
<td>sor</td>
</tr>
<tr>
<td>sparsematmult</td>
</tr>
<tr>
<td>montecarlo</td>
</tr>
<tr>
<td>elevator</td>
</tr>
<tr>
<td>tsp</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

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 \(tx\)). To conduct a fair comparison, 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, hsqldb, lusearch, and xalan from DaCapo 2006-10-MR2; avorao, lusearch, sunflow, and xalan from DaCapo 9.12-bach; crypt, series, sor, sparsematmult and montecarlo from Java Grande Forum; and elevator and tsp from micro-benchmark. 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, hede, 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].

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 methods 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

<table>
<thead>
<tr>
<th>Subject</th>
<th># of detected AVs</th>
<th># of transactions with AVs</th>
</tr>
</thead>
<tbody>
<tr>
<td>eclipse</td>
<td>19,677</td>
<td>251,500</td>
</tr>
<tr>
<td>hsqldb</td>
<td>762</td>
<td>8,267</td>
</tr>
<tr>
<td>fusearch</td>
<td>3</td>
<td>643,325</td>
</tr>
<tr>
<td>xalan</td>
<td>13,250</td>
<td>1,024,043</td>
</tr>
<tr>
<td>avrorv</td>
<td>838,981</td>
<td>8,360,455</td>
</tr>
<tr>
<td>lusearch</td>
<td>18</td>
<td>109</td>
</tr>
<tr>
<td>sunflow</td>
<td>33</td>
<td>63</td>
</tr>
<tr>
<td>xalan</td>
<td>1,964</td>
<td>288,625</td>
</tr>
<tr>
<td>crypt</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>series</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sor</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mapreduce</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>montecarlo</td>
<td>1.874</td>
<td>2,154</td>
</tr>
<tr>
<td>elevator</td>
<td>6</td>
<td>58</td>
</tr>
<tr>
<td>tsp</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

| Total   | 876,573          | 10,578,606                 |

TABLE IV. Average Numbers of Detected AVs and Transactions with AVs for a trace
Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations

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 to 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* and *xalan*, the numbers of transactions visited by Velo were 2.22x and 1.81x more than that visited by DV. On *xalan*, and *montecarlo*, the numbers of transactions visited by DV to localize witnesses were 15.72x and 37.27x less than that of Velo. On *sflow*, *eclipse*, and *lusearch*, 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* and *avroras*, 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>
<thead>
<tr>
<th>Subject</th>
<th># of witnesses</th>
<th># of distinct witnesses</th>
<th>One witness for every transaction with AV?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Velo</td>
<td>RT</td>
<td>DV</td>
</tr>
<tr>
<td>eclipse</td>
<td>19,677</td>
<td>0</td>
<td>251,500</td>
</tr>
<tr>
<td>hsqldb</td>
<td>762</td>
<td>0</td>
<td>8,267</td>
</tr>
<tr>
<td>lusearch</td>
<td>3</td>
<td>0</td>
<td>643,325</td>
</tr>
<tr>
<td>xalan</td>
<td>13,250</td>
<td>0</td>
<td>1,024,043</td>
</tr>
<tr>
<td>avroras</td>
<td>838,981</td>
<td>0</td>
<td>8,360,455</td>
</tr>
<tr>
<td>lusearch</td>
<td>31</td>
<td>0</td>
<td>109</td>
</tr>
<tr>
<td>xalan</td>
<td>33</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>xalan</td>
<td>1,964</td>
<td>0</td>
<td>288,625</td>
</tr>
<tr>
<td>crypt</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>series</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sor</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sparsematmult</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>montecarlo</td>
<td>1,874</td>
<td>0</td>
<td>2,154</td>
</tr>
<tr>
<td>elevator</td>
<td>6</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>tsp</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

| Total | 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

<table>
<thead>
<tr>
<th>Subject</th>
<th># of visited transactions</th>
<th># of dependencies ever kept</th>
<th># of dependencies ever skipped</th>
<th>Max forest size</th>
<th>Max # of dep. skipped</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Velo (A)</td>
<td>DV (B)</td>
<td>A/B</td>
<td>Velo (C)</td>
<td>DV (D)</td>
</tr>
<tr>
<td>eclipse</td>
<td>629,642,542</td>
<td>635,874</td>
<td>0.98</td>
<td>446,137</td>
<td>957</td>
</tr>
<tr>
<td>hsqldb</td>
<td>90,614</td>
<td>40,234</td>
<td>2.22</td>
<td>4,573</td>
<td>453</td>
</tr>
<tr>
<td>lusearch</td>
<td>273,525</td>
<td>1,378,152</td>
<td>0.20</td>
<td>139,938</td>
<td>8</td>
</tr>
<tr>
<td>xalan</td>
<td>59,935,702</td>
<td>3,812,909</td>
<td>15.72</td>
<td>348,804</td>
<td>2,213</td>
</tr>
<tr>
<td>avroras</td>
<td>16,567,784</td>
<td>20,344,284</td>
<td>0.81</td>
<td>1,213,829</td>
<td>349,291</td>
</tr>
<tr>
<td>lusearch</td>
<td>2,614,224</td>
<td>196</td>
<td>13337.88</td>
<td>286,730</td>
<td>12</td>
</tr>
<tr>
<td>sunflow</td>
<td>92,477</td>
<td>144</td>
<td>642.20</td>
<td>13,334</td>
<td>9</td>
</tr>
<tr>
<td>xalan</td>
<td>1,289,589</td>
<td>711,635</td>
<td>1.81</td>
<td>40,255</td>
<td>281</td>
</tr>
<tr>
<td>crypt</td>
<td>16</td>
<td>5</td>
<td>3.33</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>series</td>
<td>13</td>
<td>2</td>
<td>6.50</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>sor</td>
<td>9</td>
<td>2</td>
<td>4.50</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>sparsematmult</td>
<td>9</td>
<td>3</td>
<td>3.00</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>montecarlo</td>
<td>192,442</td>
<td>5,163</td>
<td>37.27</td>
<td>104,104</td>
<td>8,028</td>
</tr>
<tr>
<td>elevator</td>
<td>2,912</td>
<td>2</td>
<td>1456</td>
<td>3,809</td>
<td>71</td>
</tr>
<tr>
<td>tsp</td>
<td>41</td>
<td>5</td>
<td>8.20</td>
<td>36</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>710,071,899</td>
<td>26,929,108</td>
<td>-</td>
<td>2,601,576</td>
<td>361,348</td>
</tr>
<tr>
<td>Total A / Total B</td>
<td>26.39</td>
<td>Total C / Total D</td>
<td>7.20</td>
<td>Total F / Total E</td>
<td>1.70</td>
</tr>
</tbody>
</table>

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 (eclipses, hsqldb, lusearch, xalan, and avroras) 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 eclipses), which required follow-up verification 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 avroras, 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 avroras, 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, xalan and avroras), 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.

**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 S2 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., τ1) due to writing to the same memory location. Then, after the statement S3 has executed, the instance of PriceStock of thread 3 established a dependency with the instance of PriceStock of thread 2 (i.e., τ2). After the statement S4 of thread 1 has executed, the instance of PriceStock of thread 2 established a dependency with the instance of PriceStock of thread 2 of PriceStock of thread 1 (i.e., τ3). The dependency sequence (τ1, τ2, τ3) 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 (τ1, τ2, τ3) as the witness. However, since Velodrome only adds at most one edge between the same two transactions, dependency τ4 was not added to the graph maintained by Velodrome. So, Velodrome only located the non-increasing sequence (τ1, τ2, τ3) 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 ElevatorNumber$GetCountString (or GetCountString for short) in xalan of DaCapo benchmark suite, representing two scenarios triggering the atomicity violation on the instance of GetCountString. Threads 1, 2 and 3 executed an instance of ElevatorNumber$GetCountString, an instance of StringBuffer-Pool$Free (or free for short), and an instance of StringBuffer-PoolGet (or get for short) followed by another instance of StringBufferPool$Free. After the statement S2 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., τ5) due to accessing the same memory location. Then, after the statement S3 has executed, the instance of free of thread 2 established a dependency with the instance of get of thread 2 (i.e., τ6) due to accessing the same memory location. After the statement S4 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., τ7). Then, after the statement S4 of thread 1 has executed, the instance of

<table>
<thead>
<tr>
<th>Subject</th>
<th>Base (MB)</th>
<th>Memory Overhead</th>
<th>Base (sec.)</th>
<th>Velo</th>
<th>Slowdown</th>
<th>RT</th>
<th>DV</th>
</tr>
</thead>
<tbody>
<tr>
<td>eclipse</td>
<td>828</td>
<td>1.53</td>
<td>1.45</td>
<td>1.64</td>
<td>2.11</td>
<td>1.50</td>
<td>1.47</td>
</tr>
<tr>
<td>hasqldb</td>
<td>468</td>
<td>1.17</td>
<td>1.15</td>
<td>1.16</td>
<td>2.17</td>
<td>1.92</td>
<td>1.86</td>
</tr>
<tr>
<td>lusearchxml</td>
<td>396</td>
<td>1.37</td>
<td>1.49</td>
<td>1.48</td>
<td>1.91</td>
<td>4.55</td>
<td>3.53</td>
</tr>
<tr>
<td>avroras</td>
<td>554</td>
<td>1.80</td>
<td>1.39</td>
<td>1.50</td>
<td>2.29</td>
<td>6.24</td>
<td>3.52</td>
</tr>
<tr>
<td>xalan</td>
<td>521</td>
<td>1.94</td>
<td>1.52</td>
<td>1.49</td>
<td>7.15</td>
<td>13.67</td>
<td>7.77</td>
</tr>
<tr>
<td>lusearchxml</td>
<td>415</td>
<td>1.70</td>
<td>1.36</td>
<td>1.36</td>
<td>2.03</td>
<td>4.86</td>
<td>3.61</td>
</tr>
<tr>
<td>sunflow</td>
<td>407</td>
<td>1.43</td>
<td>1.42</td>
<td>1.89</td>
<td>9.70</td>
<td>9.06</td>
<td>9.41</td>
</tr>
<tr>
<td>xalan</td>
<td>499</td>
<td>2.16</td>
<td>1.35</td>
<td>1.39</td>
<td>2.45</td>
<td>3.17</td>
<td>2.96</td>
</tr>
<tr>
<td>crypt</td>
<td>326</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>2.35</td>
<td>2.18</td>
<td>2.27</td>
</tr>
<tr>
<td>series</td>
<td>228</td>
<td>1.08</td>
<td>1.00</td>
<td>3.15</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sor</td>
<td>320</td>
<td>1.45</td>
<td>1.14</td>
<td>1.14</td>
<td>2.13</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sparsematmult</td>
<td>296</td>
<td>1.35</td>
<td>1.46</td>
<td>1.45</td>
<td>1.06</td>
<td>17.49</td>
<td>17.25</td>
</tr>
<tr>
<td>montecarlo</td>
<td>670</td>
<td>1.11</td>
<td>1.10</td>
<td>1.10</td>
<td>3.58</td>
<td>2.98</td>
<td>2.67</td>
</tr>
<tr>
<td>elevator</td>
<td>158</td>
<td>1.17</td>
<td>1.00</td>
<td>1.01</td>
<td>23.64</td>
<td>1.02</td>
<td>1.00</td>
</tr>
<tr>
<td>tap</td>
<td>234</td>
<td>1.27</td>
<td>1.27</td>
<td>1.27</td>
<td>4.09</td>
<td>3.25</td>
<td>3.95</td>
</tr>
</tbody>
</table>

**TABLE VII. Memory Overhead and Slowdown**
**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.

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 21x. 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
Davida: A Decentralization Approach to Localizing Atomicity Violations

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 execution. 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

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


[55] https://zenodo.org/record/3605759#.YjYaQRNBxQI
[56] https://github.com/JikesRvm/JikesRVM


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.