DoubleChecker: Efficient Sound and Precise Atomicity Checking

Swarnendu Biswas Jipeng Huang Aritra Sengupta Michael D. Bond

Department of Computer Science and Engineering
Ohio State University
{biswass,huangjip,sengupta,mikebond}@cse.ohio-state.edu

Ohio State CSE Technical Report #OSU-CISRC-11/13-TR24, November 2013

Abstract

Atomicity is a key correctness property that allows programmers to reason about code regions in isolation. However, programs often fail to enforce atomicity correctly, leading to atomicity violations that are difficult to detect. *Dynamic program analyses* can detect atomicity violations soundly and precisely based on an atomicity specification, but existing approaches slow programs substantially.

This paper presents DoubleChecker, a novel sound and precise atomicity checker whose key insight lies in its use of two new cooperating dynamic analyses. Its *imprecise* analysis tracks inter-thread dependences soundly but imprecisely with significantly better performance than a fully precise analysis, so it detects all atomicity violations but also has false positives. Its *precise* analysis is more expensive but only needs to process parts of execution identified as potentially involved in atomicity violations. If DoubleChecker operates in *single-run* mode, the two analyses execute in the same program run, which guarantees soundness but requires logging program accesses to pass from the imprecise to the precise analysis. In contrast, in *multi-run* mode, the first program run executes only the imprecise analysis, and a second run executes both analyses. Multi-run mode trades accuracy for performance, potentially missing some violations but outperforming the single-run mode.

We have implemented DoubleChecker and an existing state-of-the-art atomicity checker called Velodrome in a high-performance Java virtual machine. DoubleChecker's single-run mode significantly outperforms Velodrome, while still providing full soundness and precision. DoubleChecker's multi-run mode improves performance further, without significantly impacting soundness in practice. These results suggest that DoubleChecker's approach is a promising direction for improving the performance of dynamic atomicity checking over prior work.

1. Introduction

Modern multicore hardware trends have made concurrency necessary for performance, but developing correct, high-performance parallel programs is notoriously challenging. Concurrency bugs have caused several high-profile failures, a testament to the fact that concurrency errors are present even in well-tested code [14, 31]. According to a study of real-world bugs, 65% of concurrency errors are due to atomicity violations [20].

Atomicity is a fundamental non-interference property that eases reasoning about program behavior in multithreaded programs. For programming language semantics, atomicity is synonymous with *serializability*: program execution must be equivalent to some serial execution of atomic regions and non-atomic instructions. That is, the code block's execution *appears* not to be interleaved with statements from other concurrently executing threads. Programmers can thus reason about atomic regions without considering effects of other threads. However, modern general-purpose languages provide crude support for enforcing atomicity—programmers are basically stuck using locks to control how threads' shared memory accesses can interleave. Programmers try to maximize scalability by minimizing synchronization, often mistakenly writing code that does not correctly enforce needed atomicity properties.

An *atomicity specification* denotes particular code regions that are expected to execute atomically. Program analysis can check atomicity by checking whether program execution(s) conform to the atomicity specification. A *violation* indicates that the program or specification is wrong (or both). Writing an atomicity specification may seem burdensome, but prior work shows that specifications can be derived mostly automatically [10, 13].

Static analysis can check atomicity but is inherently imprecise, and type-based approaches rely on annotations [8, 9, 12, 14]. Existing dynamic analyses are precise but slow programs by up to an order of magnitude or more [10, 13, 21, 33–35]. Dynamic approaches incur high costs to track cross-thread dependences soundly and precisely, which is especially expensive because it requires inserting intrusive synchronization in order to track dependences correctly.

The state-of-the-art Velodrome algorithm soundly and precisely checks *conflict serializability*, a sufficient condition for serializability [13]. During program execution, it builds a graph of *dependence edges* between *transactions*, and checks for occurrences of cycles. A transaction is a dynamically executing atomic region. Each non-transactional access conceptually executes in its own *unary transaction*. Dependences include intra-thread program order dependences, cross-thread data dependences (write–read, read–write, write-write), and release–acquire synchronization dependences.

Velodrome slows programs by 12.7X (prior work [13]) or 8.7X (our implementation and experiments) largely due to the cost of tracking cross-thread dependences soundly and precisely, which involves maintaining the last transaction to write and each thread's last transaction to read each variable. To preserve soundness in the face of potentially racy program accesses, the analysis must use atomic operations, which further slow execution by serializing in-flight instructions and by triggering remote cache misses for otherwise read-shared access patterns.

Atomicity violations are common but serious errors that are sensitive to inputs, environments, and thread interleavings, so violations manifest unexpectedly and only in certain settings. *Low-overhead* checking is needed in order to use it liberally to find bugs during in-house, alpha, and beta testing, and perhaps even some production settings. Our goal is to reduce the cost of sound and precise atomicity checking significantly in order to increase its practicality for various use cases.

Our Approach

This paper presents a sound and precise dynamic conflict serializability checker called DoubleChecker that significantly reduces overhead compared with existing state-of-the-art detectors. The key insight of DoubleChecker lies in its dual-analysis approach that avoids the high costs of precisely tracking cross-thread dependences and performing synchronized metadata updates, by overapproximating transactional dependences and then recovering precision only for transactions that might be involved in a violation.

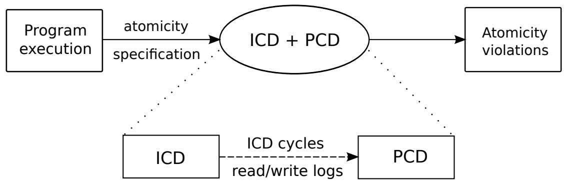
DoubleChecker achieves low overhead by staging work between two new analyses, one imprecise and the other precise. The *imprecise* analysis constructs a graph that soundly but imprecisely captures dependences among transactions. The imprecise analysis (1) detects cross-thread dependences by extending an existing concurrency control mechanism [3]; (2) computes dependence edges that soundly imply the true cross-thread dependences; (3) detects cycles in the graph, which indicate potential atomicity violations and are a superset of the true (precise) cycles; and (4) (when the precise analysis executes in the same run as the imprecise analysis) captures enough information about program accesses to allow reconstruction of precise dependences. The *precise* analysis computes precise cross-thread dependences and checks for cycles in a precise dependence graph. However,the precise analysis processes *only those transactions that the imprecise analysis identified* as being involved in a cycle. In practice, these transactions are a reasonable overapproximation of the precise cycles, eliminating most of the work that would otherwise normally be performed by *any* sound and precise analysis.

DoubleChecker supports two execution modes. In *single-run* mode, the imprecise and precise analyses operate on the same program execution. Single-run mode requires the imprecise analysis to record all program accesses so that the precise analysis can determine the precise dependences among the transactions identified by the imprecise analysis. The imprecise analysis is thus fully sound: it detects all atomicity violations in an execution. In *multi-run* mode, the first and second runs operate on different program runs. The *first run* executes only the imprecise analysis, while the *second run* executes both the imprecise and precise analyses (similar to single-run mode). The first run is thus imprecise whereas the second run is precise. The first run passes *static* program information about imprecise cycles to the second run to help reduce the instrumentation introduced by the second run. The multi-run mode is unsound since the first and second runs operate on different program executions, which could differ due to different program inputs and thread interleavings. The multi-run mode can thus miss atomicity violations that occur in either program run.

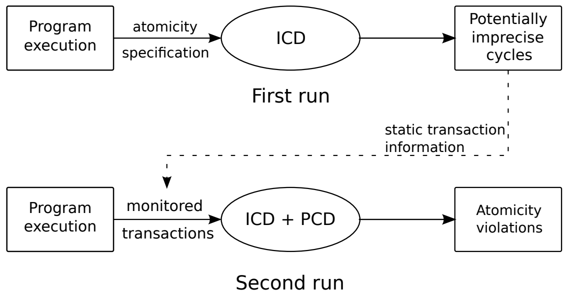
We have implemented DoubleChecker and prior work's Velodrome, in a high-performance Java virtual machine. We evaluate correctness, performance, and other characteristics of DoubleChecker on large, real-world multithreaded programs, and compare it with Velodrome. In single-run mode, DoubleChecker is a fully sound and precise analysis that slows programs by 3.5X on average, a significant improvement over Velodrome's 8.7X slowdown. DoubleChecker's multi-run mode does not guarantee soundness for either run, although we show it can find a high fraction of atomicity violations in practice. Its first and second runs slow programs by 2.2X each. As such, the *overhead* added by DoubleChecker in its single- and multi- run modes is 3.1 and 6.4 times *less* than Velodrome's, respectively. These results suggest that DoubleChecker's novel approach is a promising direction for providing significantly better performance for dynamic atomicity checking.

Contributions. This paper makes the following contributions:

- a novel sound and precise dynamic atomicity checker based on using two new, cooperating analyses:
    - an imprecise analysis that shows it can be cheaper to overapproximate dependence edges rather than compute them precisely, and thus detect cycles whose transactions are a superset of the true (precise) cycles, and
    - a precise analysis that replays an execution history of only those transactions involved in potential cycles;
- two modes of execution that provide two choices for balancing soundness and performance;



(a) Single-run mode



(b) Multi-run mode

Figure 1. An overview of DoubleChecker's two execution modes.

- implementations of DoubleChecker and Velodrome that we will make publicly available; and
- an evaluation that shows DoubleChecker outperforms Velodrome significantly, with multi-run mode providing better performance without sacrificing much soundness in practice.

2. Design of DoubleChecker

This section describes DoubleChecker, which uses two cooperating dynamic analyses to check atomicity without incurring the full costs of tracking cross-thread dependences soundly and precisely for all program accesses. First we overview DoubleChecker's analyses and execution modes. Then Section 2.2 describes the imprecise analysis, and Section 2.3 describes the precise analysis.

2.1 Overview

DoubleChecker's imprecise analysis, called *imprecise cycle detection* (ICD), monitors all program accesses to track cross-thread dependences soundly but imprecisely, i.e., the dependences imply that execution's actual dependences as well as other dependences. ICD is inherently imprecise because it identifies dependence edges by tracking shared memory "ownership"; a transfer of ownership indicates a possible dependence, but does not guarantee a dependence nor identify the source of the dependence edge. ICD constructs a transactional dependence graph whose nodes are transactions and whose edges correspond to the cross-thread dependences that ICD detects. ICD checks for cycles in this graph.

The second analysis, *precise cycle detection* (PCD), is a sound and precise analysis that limits its monitoring to a *subset* of all transactions. This subset of transactions are the transactions identified by ICD as being involved in potential cycles—which is correct because every precise cycle's transactions will always be a subset of some (potentially imprecise) cycle identified by ICD. It is important to note that PCD is *not* a standalone analysis: it performs its analysis on an execution's access log, provided by ICD.

DoubleChecker can operate in either of two modes. Figure 1 overviews the two modes of DoubleChecker.

Single-run mode. In *single-run* mode, ICD and PCD run on the same program execution. ICD logs all program reads and writes and ordering dependences between them, so PCD can reproduce the exact execution and identify precise cycles. A key cost of single-run mode is logging all program accesses.

Multi-run mode. In testing and deployment situations, programs are run multiple times with various inputs. DoubleChecker's *multi-run* mode takes advantage of this situation by splitting work across multiple program runs.[1] One run can identify transactions that might be involved in a dependence cycle, and another run can focus on this set of transactions. In contrast to single-run mode, multi-run mode avoids logging all accesses during the first run by instead performing precise checking during a second run of the program. The *first* run of multi-run mode uses only ICD. This run identifies all regular (non-unary) transactions that are involved in imprecise cycles during that execution according to their static starting locations (e.g., method signatures). Since it would be expensive to be able to identify *unary* transactions involved in cycles, the first run simply identifies whether *any* unary transactions were involved in a cycle.

The *second run* takes this static transaction information (set of regular transactions plus a boolean about unary transactions) as input and limits its analysis to the identified regular transactions and instruments *all* unary transactions if and only the unary transaction boolean is true. This approximation is acceptable in practice if most accesses are *not* unary, i.e., if they occur inside regular transactions. In our experiments, the second run uses both ICD and PCD—similar to the single-run mode—for the best performance, but the second run can use a different precise checker such as Velodrome.

In multi-run mode, DoubleChecker guarantees soundness if the two program runs execute identically. In practice, executions will differ due to different inputs and thread interleavings. The set of (static) transactions identified by the first run may not be involved in a cycle in the second run; similarly, the second run may execute transactional cycles not present in the first run. To increase efficacy, the second run can take as input the transactions identified across multiple executions of the first run. The multi-run mode can still be effective in practice if the same regions tend to be involved in cycles across multiple runs.

2.2 Imprecise Cycle Detection

Imprecise cycle detection (ICD) is a dynamic analysis that analyzes all program execution in order to detect cycles among transactions. ICD constructs a sound but imprecise graph called the *imprecise dependence graph* (IDG) to model dependences among the transactions in a multithreaded program. The nodes in an IDG are regular transactions (which correspond to atomic regions) or unary transactions (which correspond to single accesses outside of atomic regions). A *cross-thread* edge between two nodes in different threads indicates a (potentially imprecise) cross-thread dependence between the transactions. Two consecutive nodes (i.e., transactions) in the same thread are connected by an intra-thread edge that effectively captures any intra-thread dependences.

This section first describes an existing concurrency control mechanism that ICD uses to help detect cross-thread dependences but that makes detection inherently imprecise. We then describe how ICD builds the IDG and how it detects cycles.

2.2.1 Background: Concurrency Control

This section describes Octet, a recently developed software-based concurrency control mechanism [3] that ICD uses to help detect cross-thread dependences. Octet ensures that *happens-before* relationships [18] exist between all dependences in an execution.

Octet is a dynamic analysis that maintains a locality state for each object[2] that can be one of the following: WrEx_T (write exclusive for thread T), RdEx_T (read exclusive for thread T), or RdSh_c (read shared; we explain the counter c later). These states are analogous to the states in the MESI cache coherence protocol [24]. Table 1 shows the possible state transitions based on an access and the object's current state. To maintain each object's state at run time, the compiler inserts read and write barriers[3] before every write:

```
if (obj.state != WrEx_T) {
  /* slow path: change obj.state */
}
obj.f = ... ;   // program write
```

and before every read:

```
if (obj.state != WrEx_T && obj.state != RdEx_T &&
    !(obj.state == RdSh_c && T.rdShCnt >= c)) {
  /* slow path: change obj.state */
}
... = obj.f;   // program read
```

The state check, called the *fast path*, checks whether the state needs to change (the *Same state* rows in Table 1). The key to Octet's performance is that the fast path is simple and performs no writes or synchronization. If the state needs to change, the *slow path* executes in order to change the state.

Trans. type	Old state	Access	New state	Cross-thread dependence?
Same state	WrEx_T	R or W by T	Same	No
	RdEx_T	R by T	Same	
	RdSh_c	R by T *	Same	
Upgrading	RdEx_T	W by T	WrEx_T	No
	$\mathsf{RdEx}_\mathsf{T1}$	R by T2	$\mathsf{RdSh}_\mathsf{gRdShCnt}$	Possibly
Fence	RdSh_c	R by T *	Same *	Possibly
Conflicting	$\mathsf{WrEx}_\mathsf{T1}$	W by T2	$\mathsf{WrEx}_\mathsf{T2}$	Possibly
	$\mathsf{WrEx}_\mathsf{T1}$	R by T2	$\mathsf{RdEx}_\mathsf{T2}$	
	$\mathsf{RdEx}_\mathsf{T1}$	W by T2	$\mathsf{WrEx}_\mathsf{T2}$	
	RdSh_c	W by T	WrEx_T	

Table 1. Octet state transitions. *A read to a RdSh_c object by T triggers a fence transition if and only if per-thread counter $\mathsf{T.rdShCnt} < \mathsf{c}$. Fence transitions update T.rdShCnt to c.

Conflicting transitions. The last four rows of Table 1 show *conflicting* state transitions, which indicate a conflicting access and require a coordination protocol to perform the state change. For example, in Figure 2, before thread T2 can change an object o's state from $\mathsf{WrEx}_\mathsf{T1}$ to $\mathsf{RdEx}_\mathsf{T2}$, T2 must *coordinate* with T1 to ensure that T1 does not continue accessing o racily without synchronization. As part of this coordination protocol, T1 does not respond to T2's request until it reaches a *safe point*: a program point definitely *not* between an Octet barrier and its corresponding program access.

The roundtrip coordination protocol for conflicting transitions first puts o into an *intermediate* state, which helps simplify the protocol by ensuring that only one thread at a time tries to change an object's state. For example, if T2 wants to read an object that is in the $\mathsf{WrEx}_\mathsf{T1}$ state, T2 first puts the object into the $\mathsf{RdEx}^\mathsf{Int}_\mathsf{T2}$ state. The coordination protocol is then performed in one of two ways:

- The threads perform the *explicit* protocol if T1 is executing code normally. T2 sends a request to T1, and T1 responds to the request when it reaches a safe point. When T2 observes the response, a roundtrip happens-before relationship has been established, so T2 can change the state to $\mathsf{RdEx}_\mathsf{T2}$ and proceed.
- Otherwise, thread T1 is "blocking," e.g., waiting for synchronization or I/O. Rather than waiting for T1, T2 *implicitly* coordinates with T1 by atomically setting a flag that T1 will observe when it unblocks. This protocol establishes a happens-before relationship, so T2 can change the state to $\mathsf{RdEx}_\mathsf{T2}$ and proceed.

Upgrading and fence transitions. Upgrading and fence transitions do not require coordination since it is safe for other threads to continue accessing the object under the old state. In Figure 2, T3 atomically *upgrades* an object's state from $\mathsf{RdEx}_\mathsf{T2}$ to RdSh_c. The

[1] Prior bug detection work has split work across runs using sampling (e.g., [19]), which is complementary to our work.

[2] We use the term "object" to refer to any unit of shared memory.

[3] A barrier is instrumentation added to every program read and write [36].

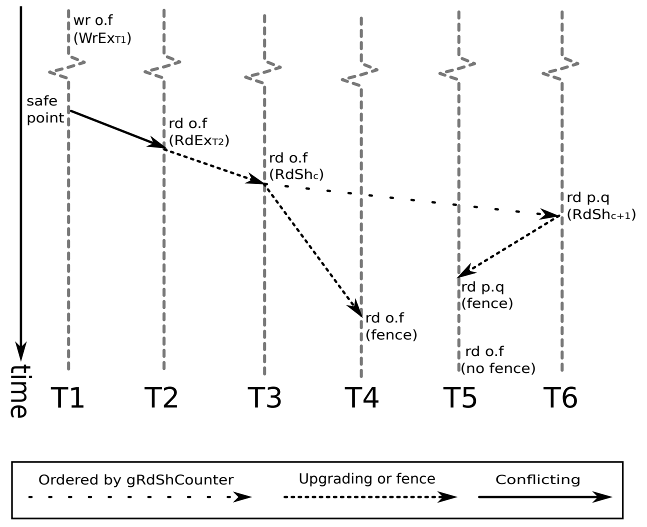


Figure 2. A possible interleaving of six concurrent threads accessing shared objects o and p, and the corresponding Octet state transitions (with new states shown) they trigger.

value c is the new value of a global "read-shared counter" gRdShCnt that gets incremented atomically every time an object transitions to RdSh, establishing a global order of all transitions to RdSh. This state change establishes a happens-before relationship from the read on T2 to the current program point on T3, ensuring a transitive happens-before relationship from T1's write to T3's read.

In Figure 2, T4 reads o in the RdSh_c state. To ensure a happens-before relationship from the last write to o (by T1) to this read in T4, a *fence* transition is triggered. The fence transition is triggered when a thread's local counter T.rdShCnt is not up-to-date with the counter c in RdSh_c. T4 issues a memory fence to ensure a happens-before relationship from T3's transition to RdSh_c to T4's read, and T4 updates T4.rdShCnt to c.

In Figure 2, T5 reads o but does *not* trigger a fence transition because T5 has already read an object (p) in the RdSh_{c+1} state. There is a *transitive* happens-before relationship from T1's write to T5's read of o because there is a happens-before relationship from o's state transition to RdSh_c in T3 to p's transition to RdSh_{c+1} in T6 (since both transitions update gRdShCnt atomically).

Octet thus establishes happens-before edges that transitively imply all cross-thread dependences [3]. ICD can piggyback on Octet's state transitions to identify potential cross-thread dependences. However, a key challenge is actually identifying the dependence edges that ICD should add to the IDG.

2.2.2 Identifying Cross-Thread Dependences

ICD uses Octet to help detect cross-thread dependences. While Octet establishes happens-before relationships that soundly imply all cross-thread dependences, it does not exactly identify the exact points in execution with which happens-before relationships are established. ICD addresses the challenge of how to identify these program points and add cross-thread edges to the IDG that soundly imply all cross-thread dependences, so that any true dependence cycle will lead to a cycle in the IDG. In this way, ICD detects atomicity violations soundly but imprecisely with substantially lower overhead than a fully precise approach.

The challenge of identifying each cross-thread edge is in identifying its *source*; the *sink* is obvious since it is the current execution point on the thread triggering the state change. At a high level, ICD keeps track of a few "last transaction to do X" facts, to help identify sources of cross-thread edges later:

T.lastRdEx – Per-thread variable that stores the last transaction of thread T to transition an object into the RdEx_T state.

gLastRdSh – Global variable that stores the last transaction among all threads to transition an object into the RdSh state.

We also define the following helper function:

currTX(T) – Returns the transaction currently executing on T.

Creating cross-thread edges for conflicting transitions. A conflicting transition (last four rows of Table 1) involves one requesting thread reqT, which coordinates with each responding thread respT. ICD piggybacks on each invocation of coordination, using the procedure handleConflictingTransition() in Algorithm 1, in order to add an edge to the IDG.

Algorithm 1 ICD procedures called from Octet state transitions.

```
procedure handleConflictingTransition(respT, reqT, oldState, newState)
    IDG.addEdge(currTX(respT) $\rightarrow$ currTX(reqT))
    if newState = $\text{RdEx}_{\text{reqT}}$ then
        reqT.lastRdEx := currTX(reqT)
    end if
end procedure

procedure handleUpgradingTransition(T, oldState, newState)
    Let rdExThread be the thread T such that oldState = $\text{RdEx}_T$
    IDG.addEdge(rdExThread.lastRdEx $\rightarrow$ currTX(T))
    IDG.addEdge(gLastRdSh $\rightarrow$ currTX(T))
    gLastRdSh := currTX(T)
end procedure

procedure handleFenceTransition(T)
    IDG.addEdge(gLastRdSh $\rightarrow$ currTX(T))
end procedure
```

Either reqT or respT will invoke the procedure as part of the coordination protocol, depending on whether the explicit or implicit protocol is used. For the explicit protocol, respT invokes the procedure before it responds, which is safe since both threads are stopped at that point. For the implicit protocol, reqT invokes the procedure since respT is blocked; reqT first temporarily atomically places a "hold" on respT so respT will not unblock until reqT is done.

Figure 3 shows a possible thread interleaving among seven concurrent threads executing transactions. The edges among transactions are IDG edges that ICD adds. The access rd o.g in Tx_{2j} conflicts with the first write to object o in transaction Tx_{1i}. The handleConflictingTransition() procedure creates a cross-thread edge in the IDG from Tx_{1i} (the transaction executing the responding safe point) to Tx_{2j} (the transaction triggering the conflicting transition).

To help upgrading transitions (explained next), handleConflictingTransition() updates the per-thread variable T.lastRdEx, the last transaction to put an object into RdEx_T. In Figure 3, this procedure updates T2.lastRdEx to Tx_{2j} (not shown).

Creating cross-thread edges for upgrading transitions. To see why ICD needs to add cross-thread edges for upgrading transitions (and not just for conflicting transitions), consider the upgrading transition from RdEx_{T2} to RdSh_{c+1} in Figure 3. Creating a cross-thread edge is necessary to capture the dependence from T1's write to o to T3's read of o *transitively*. To create this edge, T3 invokes the procedure handleUpgradingTransition() in Algorithm 1.

This procedure also creates a second edge: from the last transaction to transfer an object to the RdSh state, referenced by gLastRdSh, to the current transaction. This edge orders all transitions to RdSh state, and is needed in order to capture dependences related to fence transitions, discussed next. For rd o.f in Tx_{3k}, the procedure creates an edge from gLastRdSh, which is Tx_{5m}, to the current

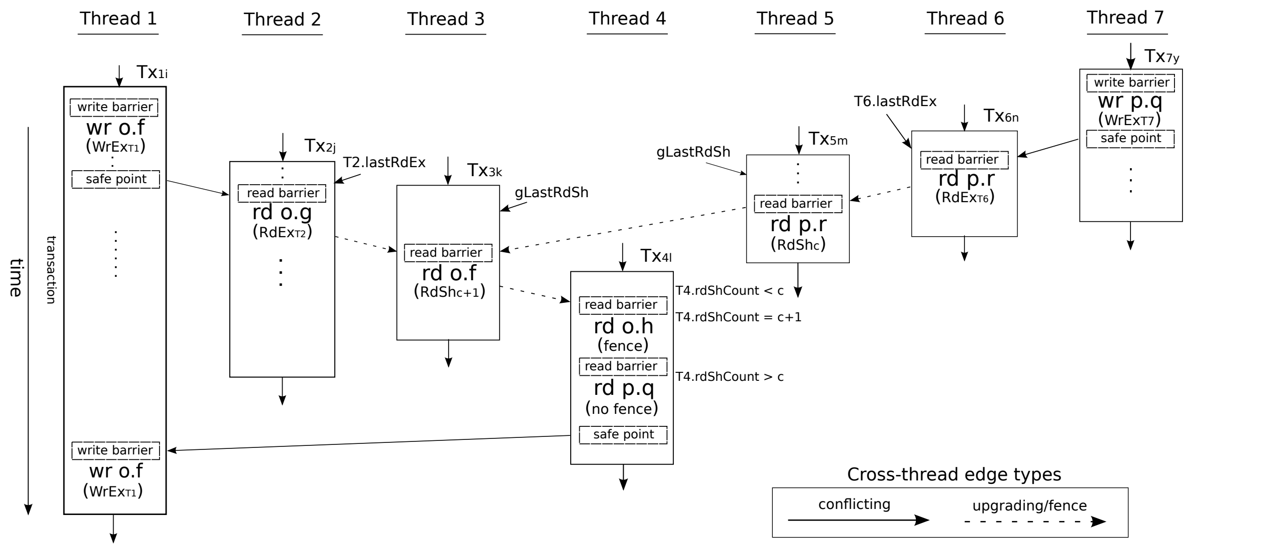


Figure 3. An example interleaving of threads executing atomic regions of code as transactions. The figure shows the Octet states after each access and the IDG edges added by ICD.

transaction. Finally, the procedure updates gLastRdSh to point to the current transaction, Tx_{3k}.

ICD safely ignores $RdEx_T \rightarrow WrEx_T$ upgrading transitions. Any new dependences created by this transition are already captured transitively by existing intra-thread and cross-thread edges.

Creating cross-thread edges for fence transitions. ICD also captures happens-before relationships transitively for fence transitions, in order to capture a possible write–read dependence for RdSh objects. Each fence transition calls handleFenceTransition() (Algorithm 1), which creates an edge from the last transaction to transition an object to RdSh (gLastRdSh) to the current transaction.

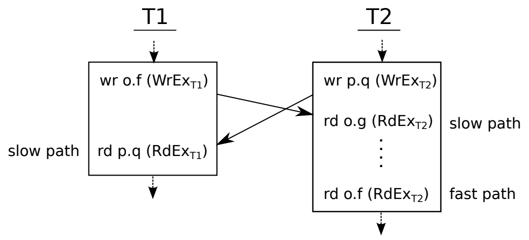
In Figure 3, T4's read of o.h triggers a fence transition and a call to handleFenceTransition(), which creates an edge from gLastRdSh (Tx_{3k}) to Tx_{4l}. This edge helps capture the possible dependence from T1's write to T4's read (in this case, no actual dependence exists since the fields accessed are different).

After T4 reads o.h, it reads p.q, which does not trigger a fence transition because T4 has already read an object (o) with a *more recent* RdSh counter (c+1) than p's RdSh counter (c). However, because $RdEx \rightarrow RdSh$ transitions create edges between all transactions that transition an object to RdSh (e.g., the edge from Tx_{5m} to Tx_{3k}), all write–read dependences are captured by IDG edges even if they do not trigger a fence transition. In the figure, the IDG edges added by the procedures transitively capture the dependence from T7's write to p.q to T4's read of p.q.

Handling synchronization operations. Like Velodrome [13], DoubleChecker captures dependences not only between reads and writes to program variables, but also between synchronization operations: lock release–acquire, notify–wait, and thread fork and join. ICD handles these operations by simply treating acquire-like operations as reads and release-like operations as writes, on the object being synchronized on.

2.2.3 Cycle detection

Rather than triggering cycle detection each time it creates a cross-thread edge (as Velodrome does [13]), ICD waits until a transaction ends to detect cycles. Consider the following example.



Even if T1 and T2 each can trigger cycle detection when they add cross-thread edges, no precise cycle yet exists until rd o.f executes. In single-run mode, to ensure that PCD sees the precise cycle, ICD should report the cycle only after the transaction finishes. By invoking cycle detection when transactions end, ICD is guaranteed to detect each cycle at least once. In the first run of multi-run mode, deferring cycle detection until transactions finish is not strictly necessary but leads to fewer invocations of cycle detection.

Detecting strongly connected components. A side effect of delayed cycle detection is that a transaction might be involved in multiple cycles. ICD therefore computes the maximal strongly connected component (SCC) [7] starting from the transaction that just ended, which identifies the set of all transactions that are part of a cycle. The SCC computation explores a transaction tx only if tx has finished. This does not affect correctness since if tx is indeed involved in cycles, an SCC computation launched when tx finishes will detect those cycles. Avoiding processing unfinished transactions helps prevent identifying the same cycles multiple times, and it avoids races with threads updating their own metadata (e.g., read-/write log updates by the fast path).

In Figure 3, ICD detects an SCC (in this case, a simple cycle) of size four when transaction Tx_{1i} ends. In single-run mode or the second run of multi-run mode, ICD passes these transactions to PCD for further processing. Note that PCD will detect a precise cycle between Tx_{1i} and Tx_{3k}. In contrast, if Tx_{3k} did not execute rd o.f, ICD would still detect an imprecise cycle, but PCD would not detect a precise cycle since none exists.

2.2.4 Maintaining Read/Write Logs

In single-run mode or the second run of multi-run mode, when ICD detects a cycle, it passes the set of transactions involved in the cycle to PCD. PCD also needs to know the exact accesses that have executed as well as cross-thread ordering between them. To provide this information, ICD records read/write logs [23] for every transaction: the exact memory accesses (e.g., object fields) read and written by the transaction. To accomplish this, ICD adds instrumentation before each program access but after Octet's instrumentation that records the access in the current thread-local read/write log. Synchronization operations are recorded as reads or writes to synchronization objects. ICD provides cross-thread ordering of accesses by recording, for each IDG edge, not only the source and sink transactions of the edge, but also the exact read/write log entries that correspond to the edge's source and sink.

2.2.5 Soundness Argument

We now argue that ICD is a sound first-pass filter. In particular, we show that for any actual (precise) cycle of dependences, there exists an (imprecise) IDG cycle that is a superset of the precise cycle.

Let C be any set of executed nodes $tx_1, tx_2, \ldots, tx_n$ whose (sound and precise) dependence edges form a (sound and precise) cycle $tx_1 \rightarrow tx_2 \rightarrow \ldots \rightarrow tx_n \rightarrow tx_1$.

Let $tx_i \rightarrow tx_j$ be any dependence edge in C. Since ICD adds edges to the IDG that imply all dependences soundly, there must exist a path of edges from tx_i to tx_j in the IDG.

Thus there exists a path $tx_1 \rightarrow tx_2 \rightarrow \ldots \rightarrow tx_n \rightarrow tx_1$ in the IDG. ICD will detect this as a cycle $C' \supseteq C$ and pass C' to PCD. Since C' contains all nodes in C, and PCD computes all dependences between nodes in C', PCD will compute the dependences $tx_1 \rightarrow tx_2 \rightarrow \ldots \rightarrow tx_n \rightarrow tx_1$, and it will thus detect the cycle C.

2.3 Precise Cycle Detection

Precise cycle detection (PCD) is a sound and precise analysis that identifies cycles of dependences on a set of transactions provided as input. DoubleChecker invokes PCD with the following input from ICD: (1) a set of transactions identified by ICD as being involved in an SCC, (2) the read/write logs of the transactions, and (3) the cross-thread edges added by ICD recorded relative to read/write log entries (to order conflicting accesses). PCD processes each SCC identified by ICD separately. Using these inputs, PCD essentially "replays" the subset of execution corresponding to the transactions in the IDG cycle, and performs a sound and precise analysis similar to Velodrome [13]. PCD uses the read/write ordering information to replay accesses in an order that reflects the actual execution order. As PCD simulates replaying execution from logs, it tracks the last access(es) to each field:

- $\mathcal{W}(\mathsf{f})$ is the last transaction to write field f.
- $\mathcal{R}(\mathsf{T},\mathsf{f})$ is the last transaction of thread T to read field f.

PCD constructs a *precise dependence graph* (PDG) based on the last-access information. A helper function thread(tx) returns the thread that executes transaction tx. At each read or write of a field f, the analysis (1) adds a cross-thread edge to the PDG (if a dependence exists) and (2) updates the last-access information of f, as shown in Algorithm 2.

PCD performs cycle detection on the PDG after adding each cross-thread edge. A detected cycle indicates a precise atomicity violation. As part of the error log, PCD reports all the transactions and edges involved in the precise PDG cycle. For example, in Figure 3, PCD processes an IDG cycle of size four, computes the PDG, and identifies a precise cycle with just two transactions, Tx_{1i} and Tx_{3k}.

PCD aids debugging by performing *blame assignment* [13], which "blames" a transaction for an atomicity violation if its outgoing edge is created *earlier* than its incoming edge, implying that the transaction *completes* a cycle. In Figure 3, PCD blames Tx_{1i}.

Algorithm 2 Rules to update last-access information for a read and write of field f by a transaction tx.

```
procedure READ(f, tx)
    if $\mathcal{W}(f) \neq$ null and thread(tx) $\neq$ thread($\mathcal{W}(f)$) then
        Add PDG edge: $\mathcal{W}(f) \rightarrow$ tx
    end if
    $\mathcal{R}(T,f) :=$ tx                                ▷ Update last read for T
end procedure

procedure WRITE(f, tx)
    if $\mathcal{W}(f) \neq$ null and thread(tx) $\neq$ thread($\mathcal{W}(f)$) then
        Add PDG edge: $\mathcal{W}(f) \rightarrow$ tx
    end if
    for all T, $\mathcal{R}(T,f) \neq$ null do
        if thread($\mathcal{R}(T,f)$) $\neq$ thread(tx) then
            Add PDG edge: $\mathcal{R}(T,f) \rightarrow$ tx
        end if
    end for
    $\mathcal{W}(f) :=$ tx                                    ▷ Update last write
    $\forall$ T, $\mathcal{R}(T,f) :=$ null                      ▷ Clear all reads
end procedure
```

3. Implementation

We have implemented a prototype of DoubleChecker in Jikes RVM 3.1.3 [1], a high-performance Java virtual machine (JVM) that has performance competitive with commercial JVMs.[4] Our implementation uses the publicly available Octet implementation [3].[5] For comparison purposes, we have also implemented Velodrome [13] in Jikes RVM since Flanagan et al.'s implementation [13] is not available (and in any case, it is implemented on top of the JVM-independent RoadRunner framework [11], so its performance characteristics could differ significantly). We will make our implementations of DoubleChecker and Velodrome publicly available.

Specifying atomic regions. DoubleChecker takes as input an atomicity specification, which is currently specified as a list of methods to be *excluded* from the specification; all other methods are part of the specification, i.e., they are expected to execute atomically. DoubleChecker extends Jikes RVM's dynamic compilers so each compiled method can statically be either *transactional* or *non-transactional*. Methods specified as atomic are always transactional, and non-atomic methods are compiled as transactional or non-transactional depending on their caller. The compilers compile separate versions of non-atomic methods called from both transactional and non-transactional contexts.

Constructing the IDG. The dynamic compilers insert instrumentation in each *atomic* method called from a *non-transactional* context. At method start, instrumentation creates a new regular transaction. At method end, it creates a new unary transaction.

Each access outside of a transaction conceptually executes in its own unary transaction. Following prior work's optimization [13], the instrumentation merges consecutive unary transactions not interrupted by an incoming or outgoing cross-thread edge.

Each transaction maintains (1) a list of its outgoing edges to other transactions and, (2) for single-run mode or the second run of multi-run mode, a read/write log that is an ordered list of precise memory access entries. Each read/write log entry corresponds to a program access; it records the base object reference, field address, and read versus write flag. The read/write log has special entries that correspond to incoming and outgoing cross-thread edges, since PCD needs to know access order with respect to cross-thread edges.

Transactions (and their read/write logs) are regular Java objects in our implementation, so garbage collection (GC) naturally collects them as they become transitively unreachable from each thread's current transaction reference. The implementation treats read/write log entries as weak references[6] to avoid memory leaks. When a reference in a read/write log entry dies, our modified GC

[4] `http://dacapo.anu.edu.au/regression/perf/9.12-bach.html`

[5] `http://www.jikesrvm.org/Research+Archive`

[6] `http://www.ibm.com/developerworks/java/library/j-jtp11225/`

replaces the reference with the old field address and the current GC number, distinguishing the field precisely.

Instrumenting program accesses. The compilers add read and write barriers at (object and static) field accesses in application methods. They do not currently instrument array accesses, nor any accesses in Java library methods, in order to imitate the closest related prior work [10, 13]. They instrument program synchronization by treating acquire operations like object reads, and release operations like object writes. ICD extends Octet's slow path to perform the procedures in Algorithm 1 (Section 2.2.2).

If PCD executes in the same run, ICD adds instrumentation to record read/write logs. Although logs are ordered, duplicate entries with no incoming or outgoing edges between them can be merged to save space. ICD tracks the last entry information using per-field metadata for WrEx and RdEx objects and per-thread hash tables for RdSh objects, in order to elide duplicate entries on the fly.

Velodrome implementation. Our DoubleChecker and Velodrome implementations share features as much as possible: they instrument the same accesses, demarcate transactions the same way, and represent transactional dependence graphs the same way. The Velodrome implementation does not use Octet. It adds two words for each object and static field: one references the transaction to write the field, and the other references the last transaction(s) (up to one per thread) to read the field since the last write. To capture release–acquire dependences, each object has an extra header word to track the last transaction to release the object's lock.

At each access, instrumentation detects cross-thread dependences, adds them to the transactional dependence graph, detects cycles (and reports a precise atomicity violation for each cycle), and updates the field's last-access information. To provide check–access atomicity and thus track cross-thread dependences accurately, the instrumentation executes a small critical section around each check and access that "locks" the field's last-writer metadata using an atomic operation (e.g., compare-and-swap instruction).

4. Evaluation

This section evaluates the correctness and performance of our prototype implementation of DoubleChecker in both single- and multi-run modes and compares with Velodrome.

4.1 Methodology

Benchmarks. Our experiments run the following programs: the multithreaded DaCapo Benchmarks that Jikes RVM 3.1.3 executes [2]: eclipse6, hsqldb6, lusearch6, xalan6, avrora9, jython9, luindex9, lusearch9,[7] pmd9, sunflow9, and xalan9 (suffixes '6' and '9' indicate benchmarks from versions 2006-10-MR2 and 9.12-bach, respectively). We also execute the following programs since they have been used in prior work [10, 13]: the microbenchmarks elevator, hedc, philo, sor, and tsp [32]; and moldyn, montecarlo, and raytracer from the Java Grande benchmark suite [28].

Experimental setup. We build a high-performance configuration of the VM (FastAdaptive) that adaptively optimizes the application and uses a high-performance generational garbage collector. We let the VM adjust the heap size automatically. Our experiments use the *small* workload size of the DaCapo benchmarks, since DoubleChecker's single-run mode runs out of memory with larger workload sizes for a few benchmarks. For benchmarks other than DaCapo, we use the same input configurations as described in prior work [10, 13]. Since the VM targets the IA-32 platform, programs are limited to a heap of roughly 1.5–2 GB; a 64-bit implementation could avoid these out-of-memory errors.

For DoubleChecker's multi-run mode, we execute 10 trials of the first run, take the union of the transactions reported as part of ICD cycles, and use it as input for the second run. This methodology represents a point in the accuracy–performance tradeoff that we anticipate would be used in practice: combining information from multiple first runs should help a second run find more atomicity violations but also increase its overhead.

[7] We use a version of lusearch9 that fixes a serious memory leak [37].

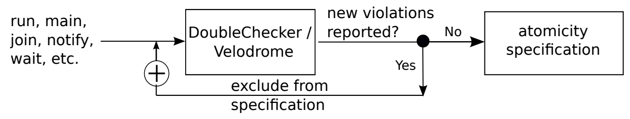

run, main, join, notify, wait, etc.
DoubleChecker / Velodrome
new violations reported?
No
Yes
atomicity specification
exclude from specification

Figure 4. Iterative refinement methodology to generate an atomicity specification for a benchmark.

Platform. The experiments execute on a workstation with a 3.30 GHz 4-core Intel i5 processor, with 4 GB memory running 64-bit RedHat Enterprise Linux 6.4, kernel 2.6.32.

Deriving atomicity specifications. Atomicity specifications for the benchmarks either have not been determined by prior work (DaCapo) or are not publicly available (non-DaCapo). We derive specifications for all the programs using an *iterative refinement* methodology used successfully by prior work [10, 12, 13, 33]. Figure 4 illustrates iterative refinement. First, all methods are assumed to be atomic with a few exceptions: top-level methods (e.g., main() and Thread.run()) and methods that contain interrupting calls (e.g., to wait() or notify()).[8] Iterative refinement repeatedly removes methods from the specification when they are "blamed" for detected atomicity violations. We terminate iterative refinement only when no new atomicity violations are reported after 10 trials, which simulates the case of well-tested software with an accurate atomicity specification and few known atomicity violations.

We use iterative refinement in two ways. First, we use it to evaluate the soundness of DoubleChecker's single- and multi-run modes by comparing the set of atomicity violations reported by Velodrome and DoubleChecker's single- and multi-run modes (Section 4.2). For each of the three configurations, we perform iterative refinement to completion (Figure 4) and collect all methods blamed as non-atomic along the way.

Second, we use iterative refinement to determine specifications that lead to few or no atomicity violations, in order to evaluate performance (Section 4.3). We take the intersection of the finalized specifications (no more violations reported) for both Velodrome and DoubleChecker (single-run mode, since it is fully sound by design), in order to eliminate any bias toward one approach.

We adjust the specifications in a few cases because of implementation issues. raytracer and sunflow9 have one and two long-running transactions, respectively, that execute atomically and that ICD passes to PCD, causing PCD to run out of memory, so we exclude the corresponding methods from the specifications. On the flip side, refining the specifications of xalan6 and avrora9 leads to so many transactions being created that DoubleChecker and Velodrome,[9] respectively, run out of memory, so we use intermediate (not fully refined) specifications for these programs.

4.2 Soundness

DoubleChecker's *single-run* mode is sound and precise by design. By comparing it to Velodrome, we sanity-check our implementations while also measuring the effect of timing differences caused by the two algorithms. *Multi-run* mode is not fully sound, so by comparing it to Velodrome and single-run mode, we evaluate how sound it is in practice. A caveat of this comparison is that the first and second runs use the same program inputs, thus representing an upper bound on soundness guarantees.

[8] For simplicity, we also exclude the DaCapo benchmarks' driver thread (which launches worker threads that actually run the benchmark program) from the atomicity specification, since we know it executes non-atomically.

[9] Velodrome may run out of memory if per-field metadata keeps otherwise-dead transactions alive. We will investigate using weak references.

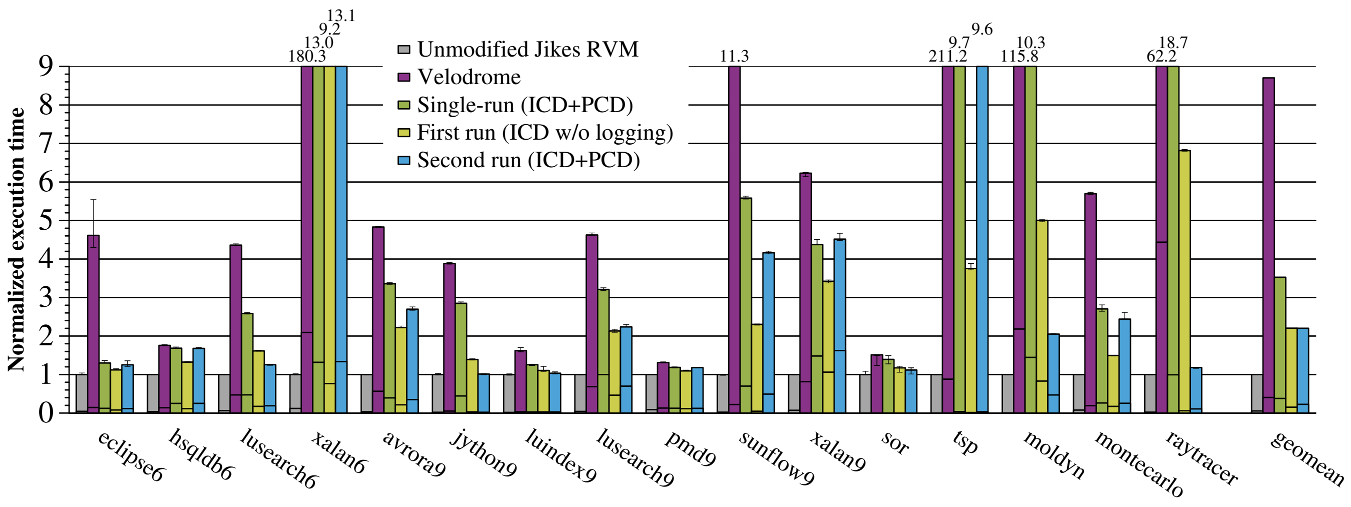


Figure 5. Run-time performance comparisons of Velodrome, DoubleChecker in the single-run mode, and the first and second runs of DoubleChecker in the multi-run mode. The sub-bars show GC time.

	Velodrome		DoubleChecker		
	Total	(Unique)	Single-run	Multi-run	(Unique)
eclipse6	230	(8)	244	190	(8)
hsqldb6	10	(0)	56	56	(0)
lusearch6	1	(0)	1	1	(0)
xalan6	57	(0)	69	54	(0)
avrora9	22	(0)	25	18	(0)
jython9	0	(0)	0	0	(0)
luindex9	0	(0)	0	0	(0)
lusearch9	41	(1)	40	38	(0)
pmd9	0	(0)	0	0	(0)
sunflow9	13	(1)	13	13	(0)
xalan9	78	(0)	82	69	(0)
elevator	2	(0)	2	2	(0)
hedc	3	(1)	3	2	(0)
philo	0	(0)	0	0	(0)
sor	0	(0)	0	0	(0)
tsp	7	(0)	7	7	(0)
moldyn	0	(0)	0	0	(0)
montecarlo	2	(0)	2	2	(0)
raytracer	0	(0)	0	0	(0)
Total	466	(11)	544	452	(8)

Table 2. Static atomicity violations reported by our implementations of Velodrome and DoubleChecker. For Velodrome and the multi-run mode, *Unique* counts how many violations were not reported by single-run mode.

Table 2 shows the number of violations reported across all trials and all steps of iterative refinement, for each atomicity checker. Each violation is a method identified by blame assignment at least once. Overall, the violations reported by Velodrome and DoubleChecker's single-run mode match closely. In theory both implementations are sound and precise, hence the violations reported should match exactly, but timing effects can lead to different interleavings. The *Unique* value in parentheses counts violations reported by Velodrome but not by single-run mode; it is nonzero for just four programs, indicating single-run mode finds nearly all violations found by Velodrome. Similarly, single-run mode finds several more violations than Velodrome in a few cases. We investigated the program with the greatest discrepancy, hsqldb6. By inserting random timing delays in Velodrome, we were able to reproduce six violations reported by DoubleChecker, suggesting differences may be due to timing effects.

Not surprisingly, multi-run mode does not quite detect as many violations as the sound single-run mode. Overall, multi-run modes detects 83% of all violations detected by single-run mode. Normalizing the detection rate across programs with at least one violation, multi-run mode detects 90% of a program's violations on average. which may be worthwhile in exchange for multi-run mode's lower run-time overhead (discussed next). Multi-run mode finds violations not found by single-run mode only for eclipse6; some but not all of these are the same violations found by Velodrome but not single-run mode.

4.3 Performance

This section compares the performance of Velodrome, DoubleChecker's single-run mode, and the first and second runs of DoubleChecker's multi-run mode. We use refined specifications that lead to no atomicity violation reports (Section 4.1). We exclude elevator, philo, and hedc since they are not compute bound [13].

Figure 5 shows the execution time of Jikes RVM running various configurations of the Velodrome and DoubleChecker implementations. Execution times are normalized to the first configuration, *Unmodified Jikes RVM*. Each bar is the median of 25 trials to minimize effects of any machine noise. We also show the mean as the center of 95% confidence intervals. Sub-bars show the fraction of time taken by stop-the-world GC.

Our implementation of *Velodrome* slows programs by 8.7X on average. This result corresponds with the 12.7X slowdown reported in prior work [13], although they are hard to compare since we implement Velodrome in a JVM and use different benchmarks. Comparing results for the benchmarks evaluated by prior work, we find that our implementation adds substantially more overhead in several cases. In particular, the Velodrome paper reports 71.7X, 4.5X, and 9.2X slowdowns for tsp, moldyn, and raytracer, respectively [13]. It is somewhat surprising that our implementation in a JVM would add more overhead than the dynamic bytecode instrumentation approach by the Velodrome authors [11, 13]. By running various partial configurations, we find that more than half (54%) of this overhead comes from synchronization required to provide analysis–access atomicity, which is unsurprising since atomic operations can lead to remote cache misses on otherwise mostly-read-shared accesses. We have learned from the Velodrome authors that their implementation eschews synchronization when metadata does not actually need to change (i.e., the current transaction is already the last writer or reader). This optimization is unsound and can miss dependences in the presence of data races. To check the benefits of this unsound optimization, we have also implemented an unsound configuration of Velodrome, which slows programs by 5X on aver-

	Single-run mode (or first run of multi-run mode)					Second run of multi-run mode				
	# Instrumented					# Instrumented				
Name	Regular transactions	Regular accesses	Unary accesses	# IDG edges	# ICD SCCs	Regular transactions	Regular accesses	Unary accesses	# IDG edges	# ICD SCCs
eclipse6	2,000,000	165,000,000	6,050,000	235,000	155	1,720,000	62,200,000	5,400,000	112,000	101
hsqldb6	86,300	13,400,000	147,000	26,100	80	85,700	10,100,000	147,000	25,700	76
lusearch6	96,400	144,000,000	124,000	17	0	0	0	0	0	0
xalan6	4,280,000	85,300,000	2,440,000	210,000	18,700	4,270,000	85,200,000	2,450,000	213,000	18,900
avrora9	1,770,000	623,000,000	3,770,000	624,000	159,000	1,110,000	271,000,000	3,770,000	504,000	134,000
jython9	8	53,200,000	29	0	0	0	0	0	0	0
luindex9	7	8,550,000	25	0	0	0	0	0	0	0
lusearch9	1,120,000	142,000,000	731,000	141	6	65,700	13,700,000	36,540	138	6
pmd9	7	2,650,000	25	0	0	0	0	0	0	0
sunflow9	35,200	265,000,000	129,000	1033	29	10,600	202,000,000	129,000	986	29
xalan9	4,760,000	77,500,000	3,760,000	75,130	667	4,720,000	77,600,000	3,690,000	74,900	702
elevator	3203	17,000	5,590	431	49	3,180	16,100	5,590	408	32
hedc	79	38,400	114	87	4	25	38,400	114	85	3
philo	6	16	459	140	0	0	0	0	0	0
sor	2	16	18,700	303	0	0	0	0	0	0
tsp	22,200	408,000	691,000,000	13,800	1	6,520	29,000	692,000,000	11,700	1
moldyn	1,360,000	1,070,000,000	211,000,000	37	0	0	0	0	0	0
montecarlo	170,000	272,000,000	230,000	31,100	3,910	160,000	252,000,000	230,000	31,100	3,880
raytracer	90,000	3,100,000,000	422,000,000	278	1	4	113	0	9	1

Table 3. Run-time characteristics of DoubleChecker for the single-run and the second run in the multi-run mode. Each average is rounded to a whole number with at most three significant digits, with ranges representing small integers that vary from run to run.

age, providing the most help to the most afflicted programs. DoubleChecker still outperforms this unsound variant of Velodrome.

The remaining configurations in Figure 5 are for DoubleChecker. All DoubleChecker configurations run ICD, which uses Octet. Octet alone adds 38% overhead on average (not shown).

Single-run (ICD+PCD) shows the execution time of running ICD and PCD in the same execution. This configuration slows programs by 3.5X (250% overhead) on average. Using partial configurations, we find that around half (126%) of this overhead comes from Octet, building the IDG using Octet concurrency control protocol, and detecting IDG cycles. (This partial configuration is similar to the first run of multi-run mode, presented next.) Logging read and write accesses as part of ICD adds another 114% overhead. Figure 5 shows that a substantial amount of the slowdown (around 38%) comes from GC for several programs, largely because of the memory footprint of long-lived read/write logs. In all, ICD adds 240% overhead. Just 10% overhead on average comes from PCD, since ICD filters out most transactions. Overall DoubleChecker's single-run mode avoids much of Velodrome's synchronization costs and adds 3.1 times less overhead than Velodrome.

First run (ICD w/o logging) executes only ICD, without logging of read and write accesses. Its functionality is similar to a subset of single-run mode that we evaluated above, and its overhead is unsurprising: it slows programs by 2.2X (120% overhead) on average. The first run of multi-run mode is significantly faster than ICD in the single-run mode primarily because it avoids logging.

Second run (ICD+PCD) executes both ICD and PCD (similar to single-run mode), except it only instruments transactions statically identified by the first run, and it instruments non-transactional accesses if and only if the first run identified any non-transactional accesses involved in cycles. The second run slows programs by 2.2X (120% overhead) on average (coincidentally the same as for the first run; the first and second runs' speedups over single-run mode come from entirely different sources).

Since the first run detects few imprecise cycles, one might expect the second run would have little work to do. However, the first run identifies transactions *statically* by method signature, leading to many more instrumented accesses in the second run than the total number of accesses identified as involved in cycles in the first run. The filter for unary accesses is even coarser; the second run must instrument all unary accesses in many cases. For this reason, DoubleChecker's ICD and PCD analyses perform better than using Velodrome for the second run, i.e., ICD is still effective as a dynamic transaction filter in the second run. Using Velodrome for the second run (i.e., instrumenting only the transaction statically identified by the first run) slows programs by 3.2X.

A promising direction for future work is to devise an effective way for the first run to more precisely communicate potentially imprecise cycles to the second run.

Overall, DoubleChecker substantially outperforms prior art. The single-run mode is a fully sound and precise atomicity checker that adds 3.1 times less overhead than Velodrome. Multi-run mode does not guarantee soundness, since atomicity checking is split between two runs, but it avoids the need for logging all program accesses (which the single-run mode needs in order to perform a precise analysis). The first and second runs of the multi-run mode each add 6.4 times less overhead than Velodrome, providing a performance–accuracy tradeoff that is likely worthwhile in practice for providing more acceptable overhead for checking atomicity. DoubleChecker's significant performance benefits justify our design's key insights (Sections 1 and 2). Our experimental results show that it is indeed cheaper to track cross-thread dependences imprecisely in order to filter most of a program's execution from processing by a precise analysis.

4.4 Run-Time Characteristics

Table 3 shows execution characteristics of ICD in single-run mode (the first run of multi-run mode provides the same results) and the second run of multi-run mode. Each value is the mean of 10 trials of a special statistics-gathering configuration of DoubleChecker; otherwise methodology is the same as Figure 5. For each of the two configurations, the table reports the number of transactions and (regular and unary) accesses instrumented, and the number of edges and SCCs in the IDG. Single-run mode instruments everything, while the second run instruments a subset of transactions. For several programs, the second run avoids instrumenting anything because the first run reports no SCCs. For one program (raytracer), the second run avoids instrumenting any unary accesses since no first-run SCC contained a unary transaction, but other programs instrument all unary accesses. For programs where the second run instruments (nearly) all transactions and accesses, there is little benefit from multi-run mode's optimization. Even when they should be the same, the counts sometimes differ across modes due to run-to-run nondeterminism.

Compared to how many memory accesses execute, there are few ICD edges, justifying ICD's approach that optimistically assumes accesses are not involved in cross-thread dependences. There are few ICD SCCs in most cases, justifying DoubleChecker's dual-analysis approach and explaining why PCD adds low overhead.

5. Related Work

This section details other static and dynamic analyses besides the most closely related work, Velodrome [13].

Statically checking atomicity. Static approaches can check all inputs soundly, but they are imprecise, and in practice they do not scale well to large programs nor to dynamic language features such as dynamic class loading. Type systems can help check atomicity but require a combination of type inference and programmer annotations [12, 14]. Model checking does not scale well to large programs because of state space explosion [8, 9, 16]. Static approaches are well suited to analyzing critical sections but not wait–notify synchronization.

Dynamically checking atomicity. Wang and Stoller propose two dynamic analyses for checking atomicity based on detecting unserializable patterns [34]. These approaches aim to find potential violations in other executions, but this process is inherently imprecise, so they may report false positives. The authors also propose "commit-node" algorithms, which are more precise and check conflict serializability and view serializability [33]. *Atomizer* is a dynamic atomicity checker that uses a variation of the lockset algorithm [27] to determine shared variables that can have racy accesses, and monitors those variables for potential atomicity violations. Atomizer reports false positives since it relies on the locket algorithm. Like Velodrome, these dynamic approaches slow programs by an order of magnitude or more.

Inferring atomicity specifications. Several approaches infer an atomicity specification automatically [6, 15, 21, 30, 35]. However, these approaches are inherently unsound and imprecise. Furthermore, many of these approaches add high overhead to track cross-thread dependences accurately, e.g., AVIO slows programs by more than an order of magnitude [21].

Alternatives. *HAVE* combines static and dynamic analysis to obtain benefits of both approaches [4]. Because HAVE speculates about untaken branches, it can report false positives. Prior work *exposes* atomicity violations [25, 26, 29], which is complementary to checking atomicity.

Transactional memory enforces programmer-specified atomicity annotations by speculatively executing atomic regions as *transactions*, which are rolled back if a region conflict occurs [17]. *Atom-Aid* relies on custom hardware to execute regions atomically and to detect some atomicity violations [22].

Atomicity can be achieved using static analysis that infers needed locks automatically from an atomicity specification, but it is inherently imprecise, leading to overly conservative locking [5].

In summary, most prior work gives up on soundness or precision or both, and dynamic approaches slow programs substantially. In contrast, DoubleChecker checks conflict serializability soundly and precisely and significantly outperforms other dynamic approaches.

6. Conclusion

This paper presents a new direction for dynamic sound and precise atomicity checking that divides work into two cooperating analyses: a lightweight analysis that detects cross-thread dependences, and thus atomicity violations, soundly but imprecisely, and a precise second analysis that focuses on potential cycles and rules out false positives. These cooperating analyses can execute in a single run, or the imprecise analysis can first run by itself, providing a performance–soundness tradeoff. DoubleChecker outperforms existing sound and precise atomicity checking, reducing overhead by several times, suggesting that its new direction can provide substantially more efficient atomicity checking than existing work.

Acknowledgments

We thank Harry Xu and Jaeheon Yi for feedback on the paper text; and Cormac Flanagan, Stephen Freund, and Jaeheon Yi for discussions and advice.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[3] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, pages 693–712, 2013.

[4] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *FASE*, pages 425–439, 2009.

[5] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring Locks for Atomic Sections. In *PLDI*, pages 304–315, 2008.

[6] L. Chew and D. Lie. Kivati: Fast Detection and Prevention of Atomicity Violations. In *EuroSys*, pages 307–320, 2010.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[8] A. Farzan and P. Madhusudan. Causal Atomicity. In *CAV*, pages 315–328, 2006.

[9] C. Flanagan. Verifying Commit-Atomicity Using Model-Checking. In *SPIN*, pages 252–266, 2004.

[10] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *SCP*, 71(2):89–109, 2008.

[11] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, pages 1–8, 2010.

[12] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for Atomicity: Static Checking and Inference for Java. *TOPLAS*, 30(4):20:1–20:53, 2008.

[13] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[14] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, pages 338–349, 2003.

[15] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *ICSE*, pages 231–240, 2008.

[16] J. Hatcliff, Robby, and M. B. Dwyer. Verifying Atomicity Specifications for Concurrent Object-Oriented Software using Model-Checking. In *VMCAI*, pages 175–190, 2004.

[17] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[18] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[19] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California at Berkeley, 2004.

[20] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[21] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ASPLOS*, pages 37–48, 2006.

[22] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, pages 277–288, 2008.

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[24] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, pages 348–354, 1984.

[25] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *FSE*, pages 135–145, 2008.

[26] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, pages 25–36, 2009.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[28] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 8–8, 2001.

[29] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *FSE*, pages 37–46, 2010.

[30] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting Atomic-Set Serializability Violations with Conflict Graphs. In *RV*, pages 161–176, 2012.

[31] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[32] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[33] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *PPoPP*, pages 137–146, 2006.

[34] L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multi-threaded Programs. *IEEE TSE*, 32:93–110, 2006.

[35] M. Xu, R. Bodík, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *PLDI*, pages 1–14, 2005.

[36] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers Reconsidered, Friendlier Still! In *ISMM*, pages 37–48, 2012.

[37] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why Nothing Matters: The Impact of Zeroing. In *OOPSLA*, pages 307–324, 2011.