Optimizing Record/Replay through Relaxed Total Ordering and Multi-Version eXecution

David Schwartz
University of Illinois Chicago, USA

Luís Pina
University of Illinois Chicago, USA

Abstract

Record/Replay (RR) allows developers to record an execution and then replay it exactly as it was recorded. RR enables deterministic replay of non-deterministic behaviors in a different environment than the one used for the recording, which can capture complex bugs in production and find the root cause during development. Unfortunately, support for RR still introduces a non-negligible amount of performance overhead, which limits its applicability. Two main sources of such overhead in state-of-the-art RR systems are: multi-threading, and I/O bound workloads. To ensure high-fidelity when replaying multi-threading execution, recordings either capture the total order of events, which the replayer then enforces, or capture a partial order that requires further processing before replay. Recording also effectively doubles the I/O performed as the recorder needs to perform the original I/O and then record it to a log. Such increased I/O severely limits the performance of I/O dominated workloads.

In this paper, we present two complimentary techniques to reduce the overhead of RR. First, we introduce Relaxed Total Order (RTO), an online-computable weakening of total order that preserves the cross-thread constraints needed for replay while avoiding unnecessary serialization. We design RTO to be compatible with Multi-Version eXecution (MVX), enabling online deterministic replay without pre-processing the recording log or heavyweight coordination. We formalize RTO's strictness and correctness, showing that it is a novel point between partial- and total-order. Our prototype implementation on top of an existing state-of-the-art RR system reduces recording overhead from **21.0%** to **15.3%** and halves replay overhead from **67.5%** to **31.7%**.

Second, we combine RR with Multi-Version eXecution (MVX) to eliminate RR's poor performance on I/O-bound workloads. Our hybrid design uses a follower variant to absorb the extra I/O needed for logging, and to backfill as much I/O as possible from the same underlying system, keeping the user-facing leader off the critical path. Our prototype reduces the overhead required to record I/O bound programs from **192.5%** to just **25.5%**, without penalizing other more common workloads. Together, RTO and hybrid MVX/RR substantially narrow the gap between today's RR systems and practical, low-overhead, always-on deployment.

2012 ACM Subject Classification Software and its engineering → Software post-development issues; Software and its engineering → Software reliability; Software and its engineering → Software maintenance tools; Software and its engineering → Software testing and debugging

Keywords and phrases Record Replay, Multi-version Execution, Java Virtual Machine

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.8

Funding This work was funded in part by NSF CCF-2227183.

Acknowledgements We would like to thank: William Mansky for his review of early versions of our formal model, John Fike for IT support during development and testing, and the anonymous reviewers for their insightful comments.

1 Introduction

Record-Replay (RR) is a technique that captures the execution of a program into a log, which can be saved to persistent storage and used later for replaying the same program execution. Typical RR systems capture all the non-determinism accessed during a program's execution (*e.g.,* data read from files and/or network) to ensure a high-fidelity replay. One common use-case is to replicate rare bugs that are hard to trigger: recording the bug once ensures consistent replay every time using the recorded log.

Multi-Version eXecution (MVX) is a technique very similar to RR that runs both recorder and replayer at the same time (sometimes referred to as *"online RR"*), using many processes called *variants*. Modern MVX architectures [14, 23, 27, 28, 34, 35] use one variant as the recorder — dubbed the *leader* — and many other variants as replayers on the same log — dubbed *followers*. By exploiting diversity among followers, MVX has applications in the broad areas of: software security [34, 35] (*e.g.,* variants have stacks growing in different directions to survive different attacks, and all vote on security-sensitive operations), software reliability [13, 14, 28] (variants run different releases of the same program, if one crashes due to a newly introduced bug the remaining survive without causing the whole program to stop, i.e., the survivors keep providing service), software analysis [23, 36] (each variant uses a different and incompatible dynamic analysis), and software availability [24, 27] (the follower performs a software update while the leader keeps providing service). Recent work explores the synergy between RR and MVX to update internet browsers without any source code modifcations nor user disruption [27].

Despite their promise, RR and MVX systems still suffer from a high performance penalty for some applications (*e.g.,* multi-threaded applications) and workloads (*e.g.,* I/O bound workloads are a known pathological case).

Recording multi-threaded orderings. Figure 1 shows an original execution with 2 threads and 3 locks controlling access to critical sections. Locks L_1 and L_2 are used exclusively by threads T_1 and T_2 respectively; observe in Figure 1 that different execution ordering strategies (explained below) handle the relationship between L_1 and L_2 differently. Systems that do not support multi-threading [6] cannot replay such an execution reliably, as a replay allows (incorrectly) T_2 to acquire L_3 first. Supporting even race-free programs requires capturing, at least, the order in which threads issue synchronization operations, such as acquiring and releasing locks. Systems such as the popular `rr` [22] follow a straightforward option: to enforce a **uniprocessor execution** of all threads when recording, effectively executing only one thread at a time. Unfortunately, as shown in Figure 1, doing so slows down the original execution dramatically. Another option is to record a **Total Order (TO)** of all synchronization events in the original execution, and enforce the same order on the replayer/followers [14, 23, 28, 34]. Despite its obvious correctness, TO's strictness results in long wait times when replaying unrelated operations. In Figure 1, Events e_4 and e_6 are completely unrelated: they take place in different threads (T_1 and T_2, respectively), and operate on different resources (L_2 and L_1, respectively). Yet, a TO replay enforces the order observed during the original execution: TO_3. Alternatively, recording a **Partial Order (PO)** results in less strict orderings by capturing the order of operations on the same resource, and enforcing it among different threads when replaying. Figure 1 shows that the only wait when using partial order enforces that Event e_8 takes place before Event e_9 as observed in the recording (PO_3).

Unfortunately, using PO in RR results in complex implementations with subtle trade-offs. For instance, Octet [8, 9] can replay data-races by capturing individual writes to shared

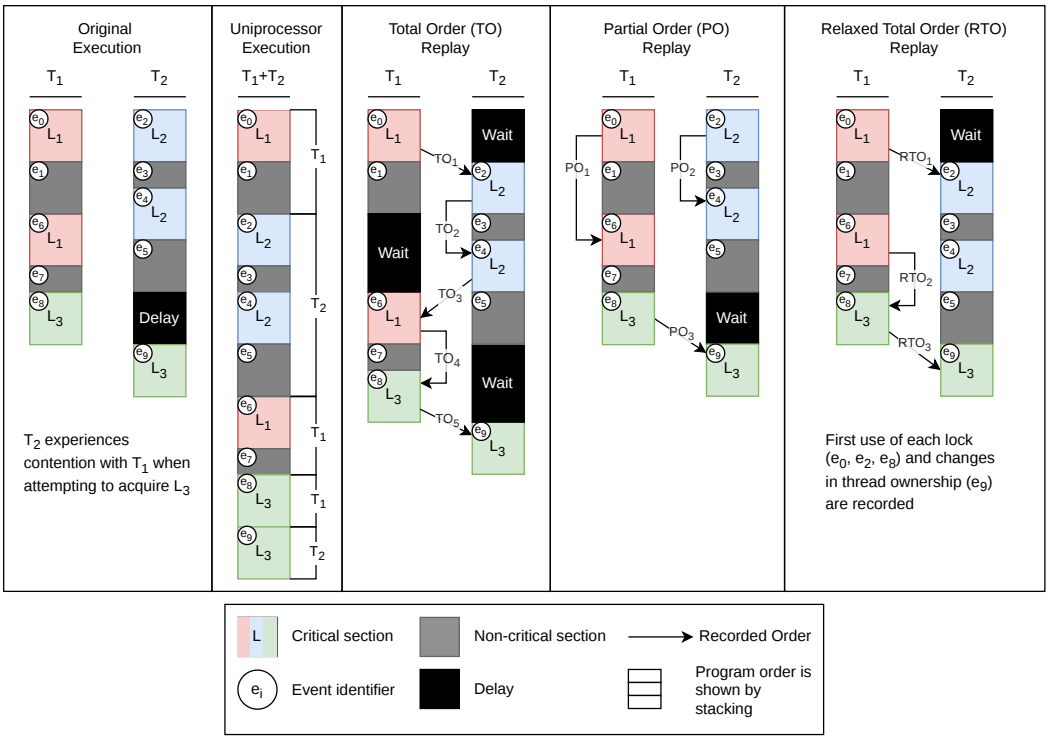

Figure 1 Sample execution and possible replay orders. The original execution shows two threads T_1 and T_2 executing noncritical and critical regions protected by locks L_1, L_2, and L_3. Regions are uniquely labeled and numbered according to their order in the uniprocessor execution. Possible scheduling strategies for replaying the execution are: Uniprocessor, Total Order (TO), Partial Order (PO), and Relaxed Total Order (RTO), which differ in how they order critical regions for L_1 and L_2.

memory, but imposes prohibitive overhead for production use (31% overhead recording and 49% replaying). Respec [19, 32] is difficult to adopt as it requires custom OS support to roll-back program execution when recording/replaying. Similarly, Castor [21] uses special instructions only available on certain Intel CPUs to capture the PO efficiently and, therefore, cannot provide efficient support for all hardware (*e.g.,* ARM processors). LEAP [15] requires a conservative static analysis to identify all synchronization resources before recording, which is not feasible for programs in managed languages (like Java) that can create locks dynamically. In addition, and both Castor and LEAP have to pre-process/sort the resulting log before replay. Such extra pre-processing represents a hidden cost with a duration that is workload dependent. Using PO in MVX systems is even more complex: unlike an RR replayer, MVX followers cannot pre-process the log (the log is shared as it is constructed *"online"* from the leader), and supporting PO results in overall poor scalability due to increased inter-thread communication [33].

I/O bound workloads are pathological. RR and MVX typically operate under the assumption that recorded operations are sparse, which holds for CPU-bound workloads with buffered I/O. Unfortunately, performance degrades dramatically when such an assumption fails, resulting in RR recorders effectively doubling the amount of I/O performed: once for the log, and once for the recorded program. We measured the performance overhead to record such an I/O-bound workload as high as 4.6x–8.3x for state-of-the-art RR systems (Section 5.5).

In this paper, we propose novel solutions to support efficient RR of: (i) multi-threaded programs, such that it is compatible with MVX, and (ii) I/O-bound workloads. **First**, we propose a new ordering for capturing relevant synchronization operations: **Relaxed Total Ordering (RTO)**. The main insight behind RTO is based on the time locality of synchronization resources [10, 31]: when a thread t acquires a lock l, it is likely that the same t will acquire l again in the near future. Time locality is workload dependant, and our empirical results suggest it is prevalent in common workloads: the high proportion of Relaxed events in Table 5 is an indicator that the benchmark programs often experience such time locality. We show that RTO can be implemented efficiently in systems that already capture TO simply by tracking the last owner of each synchronization resource and ordering operations only when the owner changes (Section 3.1). For instance, in Figure 1, RTO only captures the event ordering when TO ownership changes: e_0 for L_1, e_2 for L_2, and e_8 and e_9 for L_3. We formalize RTO (Section 2) and show that is is a new point in the strictness spectrum between PO and TO: RTO is stricter than PO but less strict than TO. We also describe a prototype implementation of RTO on a TO-based RR/MVX system for Java [28] (Section 4.1) that dramatically reduces overhead on the: recorder, from **21.0%** to **15.3%**, replayer, from **67.5%** to **31.7%**, leader, from **25.9%** to **23.3%**, and follower overhead, from **73.6%** to **41.9%**. However, RTO requires that the program is *data race free (DRF)* [20], which is a common requirement of prior systems [14, 21, 28, 34, 35]. **Second,** we propose a novel combination of MVX and RR to address the well-known pathological case of I/O bound workloads. Building on the synergy between RR and MVX [27], we propose a new hybrid recorder that uses MVX internally to split the burden of recording between one leader and one follower (Section 3.2). Based on the assumption that both leader and follower execute on the same machine, our proposed hybrid leader only captures non-determinism not available to the hybrid follower, such as network I/O and multi-threaded ordering. Our proposed hybrid follower executes the same program being recorded, capturing the majority of the same events as the leader (*e.g.,* file-system I/O) without any inter-variant communication, reducing the pathological performance overhead without increasing the overhead in common workloads, and generating logs that can be used by the original replayer.

In summary, our paper makes the following contributions:

- A novel multi-threaded event ordering — Relaxed Total Ordering (RTO) — that can be efficiently implemented in both RR and MVX for managed languages and dramatically reduces the performance overhead, together with a formalization that places RTO in the strictness spectrum between PO and TO.
- A novel recording technique that uses MVX internally to reduce the overhead on a well-known pathological case of RR — I/O bound workloads — without increasing the overhead in common workloads, and generating logs that are compatible with the unmodified replayer.
- The design, implementation, and evaluation of the proposed techniques on top of a state-of-the-art MVX/RR system.
- A replication package [**?**] that includes our implementation [**?**], all the scripts that automate the experiments we describe, and the data we obtained.

2 Formal Model for Relaxed Total Order

In this section, we define our proposed *Relaxed Total Order (RTO)* formally. Our formalism is inspired by previous work in RR [19, 21, 32], and proposes RTO as a novel event ordering that explore the spectrum defined by Total Order (TO) and Partial Order (PO) at each

end. We show that RTO is fundamentally different from PO and TO: when considering only events observed in the recording, RTO is less strict than Total Order (TO) and Partial Order (PO) is less strict than RTO ($\rightarrow_{\mathrm{PO}} \subseteq \rightarrow_{\mathrm{RTO}} \subseteq \rightarrow_{\mathrm{TO}}$) placing RTO between PO and TO. Following established literature [14, 21, 28, 34, 35], we assume that the underlying program is *data-race free (DRF)* [20], and that capturing all synchronization acquisition operations (*lock acquisitions*) is enough for an accurate replay of any recorded execution.

2.1 Event Model, Total Order, and Partial Order

▶ **Definition 1** (Recorded execution and events). *A recorded execution is a finite sequence* $E = \langle \bot, e_1, \ldots, e_n \rangle$ *of* synchronization acquisition events *(e.g., locking). Each event* $e \in E$ *is associated with: (i) a* thread *identifier* $t(e)$*, and (ii) a* synchronization resource *identifier* $r(e)$*. The first event* $\bot$ *is used as a marker for the start of the program.*

▶ **Definition 2** (Resource and thread set). *Given an execution* E*:*

1. *The set of resources used in* E *is* $R(E) = \{\, r' \mid \exists e \in E.\ r(e) = r' \,\}$.
2. *The set of threads used in* E *is* $\mathbb{T}(E) = \{\, t' \mid \exists e \in E.\ t(e) = t' \,\}$.
3. $r(\bot) = r_\bot$ *is a unique resource and* $t(\bot) = t_\bot$ *is a unique thread, neither are used in the rest of the log:* $\forall i \in \{1, \ldots, n\} : r(e_i) \neq r_\bot \wedge t(e_i) \neq t_\bot$.

▶ **Definition 3** (Program order (**C**ode **O**rder)). *For each thread identifier* τ*, let* $E_\tau = \{e \in E \mid t(e) = \tau\}$*. The* program order *relation* $\rightarrow_{\mathrm{CO}}$ *is the per-thread order induced by the trace: for* $e_1, e_2 \in E : e_1 \rightarrow_{\mathrm{CO}} e_2$ *iff* $t(e_1) = t(e_2) \ \wedge\ e_1$ *occurs before* e_2 *in* E.

▶ **Definition 4** (Per-resource recorded order). *The* per-resource order *relation* $\rightarrow_{\mathrm{res}}$ *captures the order in which the recorder observes acquisitions of the same resource. For* $e_1, e_2 \in E$ *:* $e_1 \rightarrow_{\mathrm{res}} e_2$ *iff* $r(e_1) = r(e_2) \ \wedge\ e_1$ *occurs before* e_2 *in* E.

▶ **Definition 5** (Adjacency in any given order). *Two events* $e_1, e_2 \in E$ *are* adjacent *for any given order* $\rightarrow_?$ *if* $e_1 \rightarrow_? e_2$ *and there is no event* $e \in E$ *such that* $e_1 \rightarrow_? e \rightarrow_? e_2$.

▶ **Definition 6** (Total Order (TO)). *A* total order *is a relation* $\rightarrow_{\mathrm{TO}} \subseteq E \times E$ *such that, for any two distinct events* $e_1, e_2 \in E$*:*

1. ***Totality.*** $e_1 \neq e_2 \implies (e_1 \rightarrow_{\mathrm{TO}} e_2 \ \vee\ e_2 \rightarrow_{\mathrm{TO}} e_1)$.
2. ***Consistent with program order.*** $e_1 \rightarrow_{\mathrm{CO}} e_2 \implies e_1 \rightarrow_{\mathrm{TO}} e_2$.
3. ***Consistency with per-resource order.*** $e_1 \rightarrow_{\mathrm{res}} e_2 \implies e_1 \rightarrow_{\mathrm{TO}} e_2$.
4. ***Order axioms.*** $\rightarrow_{\mathrm{TO}}$ *is a strict total order on* E*: it is irreflexive and transitive.*

Remark — Restriction to a resource. For any fixed resource r, the restriction of $\rightarrow_{\mathrm{res}}$ to events on r is a total order over the acquisition events of r observed during execution, and it is consistent with $\rightarrow_{\mathrm{TO}}$ (*i.e.,* $e_1 \rightarrow_{\mathrm{res}} e_2$ implies $e_1 \rightarrow_{\mathrm{TO}} e_2$).

▶ **Definition 7** (Inter-thread synchronization edges). *Let* E *be an execution. We define the* inter-thread synchronization edges *as the set:*
$\rightarrow_{\mathrm{sync}} = \{\, (e_1, e_2) \mid e_1$ *and* e_2 *adjacent in* $\rightarrow_{\mathrm{res}} \wedge\ t(e_1) \neq t(e_2) \,\}$.

▶ **Definition 8** (Data-Race Freedom (DRF)). *An execution* E *is* data-race free *if every pair of conflicting memory accesses across threads is ordered by the program's synchronization (i.e., by the lock/condition-variable discipline captured by* $\rightarrow_{\mathrm{sync}}$ *and intra-thread control order captured by* $\rightarrow_{\mathrm{CO}}$*).*

▶ **Definition 9** (Partial Order (PO)). *A* partial order *is a strict relation* $\rightarrow_{\mathrm{PO}} \subseteq E \times E$ *defined as the transitive closure of the program order and per-resource order for any two distinct events* $e_1, e_2 \in E$*:* $((e_1 \rightarrow_{\mathrm{CO}} e_2) \vee (e_1 \rightarrow_{\mathrm{sync}} e_2)) \implies e_1 \rightarrow_{\mathrm{PO}} e_2$.

Algorithm 1 Procedure to build $E_{nonrelaxed}$ and $E_{relaxed}$. We assume the recorder emits one particular linearization of some total order that extends $\rightarrow_{\text{TO}} \cup \rightarrow_{\text{CO}}$.

Require: A linearization of $\rightarrow_{\text{TO}}$ over the events in the execution: $\langle e_0 = \bot, e_1, e_2, \ldots, e_n \rangle$
Require: $r : E \rightarrow R$ (resource of an event), $t : E \rightarrow \mathbb{T}$ (thread of an event)
Ensure: $E_{relaxed} \subseteq E \times E$, $E_{nonrelaxed} \subseteq E \times E$

1: $E_{relaxed} \leftarrow \emptyset$
2: $E_{nonrelaxed} \leftarrow \emptyset$
3: $lastAcquisition : R \rightharpoonup E \leftarrow \emptyset$ ▷ partial map, immediately previous acquisition
4: **for** $i \leftarrow 1$ to n **do**
5: $r_i \leftarrow r(e_i)$; $t_i \leftarrow t(e_i)$
6: **if** $r_i \notin dom(lastAcquisition)$ **then**
7: $E_{nonrelaxed} \leftarrow E_{nonrelaxed} \cup \{(e_{i-1}, e_i)\}$ ▷ First use of r_i
8: **else**
9: $e_{prev} \leftarrow lastAcquisition[r_i]$
10: **if** $t(e_{prev}) = t_i$ **then**
11: $E_{relaxed} \leftarrow E_{relaxed} \cup \{(e_{prev}, e_i)\}$ ▷ Same owner
12: **else**
13: $E_{nonrelaxed} \leftarrow E_{nonrelaxed} \cup \{(e_{prev}, e_i)\}$ ▷ Different owner
14: **end if**
15: **end if**
16: $lastAcquisition[r_i] \leftarrow e_i$
17: **end for**

2.2 Relaxed Total Order (RTO)

▶ **Definition 10** (Relaxed vs. non-relaxed edges (construction)). *We construct two sets: (i) a set of* relaxed pairs $E_{relaxed} \subseteq E \times E$, *and (ii) a set of* non-relaxed pairs $E_{nonrelaxed} \subseteq E \times E$. *Elements of $E_{relaxed}$ are adjacent in $\rightarrow_{\text{res}}$. Elements of $E_{nonrelaxed}$ are either adjacent in $\rightarrow_{\text{res}}$ (when the same resource is re-acquired) or adjacent in $\rightarrow_{\text{TO}}$ (when acquiring a resource for the first time). Algorithm 1 defines how the sets are constructed.*

▶ **Definition 11** (Relaxed Total Order (RTO)). *The* Relaxed Total Order *relation $\rightarrow_{\text{RTO}} \subseteq E \times E$ is a strict order obtained by the transitive closure of the following base edges:*

1. ***Program-order edges.*** *For any $e_1, e_2 \in E : e_1 \rightarrow_{\text{CO}} e_2 \implies e_1 \rightarrow_{\text{RTO}} e_2$.*
2. ***Non-relaxed edges.*** *For any $e_1, e_2 \in E : (e_1, e_2) \in E_{\text{nonrelaxed}} \implies e_1 \rightarrow_{\text{RTO}} e_2$.*

In other words: $\rightarrow_{\text{RTO}} = (\rightarrow_{\text{CO}} \cup E_{\text{nonrelaxed}})^+$

As an example, we now describe an execution of Algorithm 1 on the execution from Figure 1. Table 1 shows the values of e_i, r_i, t_i and the contents of $E_{nonrelaxed}$, $E_{relaxed}$, and *lastAcquisition* for all TO edges depicted in Figure 1: e_0, e_2, e_4, e_6, e_8, and e_9.

First, Algorithm 1 processes nodes e_0 and e_2 and captures the first use of L_1 and L_2 in Line 6, updating $E_{nonrelaxed}$ in Line 7. Algorithm 1 also captures the last owner of each resource by updating *lastAcquisition* in Line 16. Then, Algorithm 1 processes node e_4 and e_6, which do not change the ownership of either L_1 or L_2 in Line 10 resulting in adding edges (e_2, e_4) and (e_0, e_6) to $E_{relaxed}$ in Line 11, and updating *lastAcquisition* in Line 16. Next, Algorithm 1 processes node e_8 and captures the first use of L_3 in Line 6, updates $E_{nonrelaxed}$ in Line 7, and *lastAcquisition* in Line 16. Finally, when processing node e_9, algorithm 1 identifies edge (e_8, e_9) as nonrelaxed due to the change in ownership of L_3 which causes

Table 1 Trace of running Alogrithm 1 on the execution from Figure 1.

e_i	r_i	t_i	e_{prev}	$t(e_{prev}) == t_i$	$E_{nonrelaxed}$	$E_{relaxed}$	$lastAcquisition$
e_0	L_1	T_1	$\perp$	—	$\{(\perp, e_0)\}$	$\emptyset$	$\{L_1 \to e_0\}$
e_2	L_2	T_2	$\perp$	—	$\{(\perp, e_0), (e_0, e_2)\}$	$\emptyset$	$\{L_1 \to e_0, L_2 \to e_2\}$
e_4	L_2	T_2	e_2	true	$\{(\perp, e_0), (e_0, e_2)\}$	$\{(e_2, e_4)\}$	$\{L_1 \to e_0, L_2 \to e_4\}$
e_6	L_1	T_1	e_0	true	$\{(\perp, e_0), (e_0, e_2)\}$	$\{(e_2, e_4), (e_0, e_6)\}$	$\{L_1 \to e_6, L_2 \to e_4\}$
e_8	L_3	T_1	$\perp$	—	$\{(\perp, e_0), (e_0, e_2), (e_6, e_8)\}$	$\{(e_2, e_4), (e_0, e_6)\}$	$\{L_1 \to e_6, L_2 \to e_4, L_3 \to e_8\}$
e_9	L_3	T_2	e_8	false	$\{(\perp, e_0), (e_0, e_2), (e_6, e_8), (e_8, e_9)\}$	$\{(e_2, e_4), (e_0, e_6)\}$	$\{L_1 \to e_6, L_2 \to e_4, L_3 \to e_9\}$

the test in Line 10 to fail. As a result, Algorithm 1 identifies edge (e_8, e_9) as nonrelaxed in Line 13, and updates *lastAcquisition* in Line 16.

2.3 Strictness between RTO, TO, and PO

▶ **Theorem 12** (RTO is no stronger than TO). *For any execution E, the Relaxed Total Order relation is included in the Total Order relation: $\to_{\text{RTO}} \subseteq \to_{\text{TO}}$.*

Equivalently, for all events $e_1, e_2 \in E$, if $e_1 \to_{\text{RTO}} e_2$ then $e_1 \to_{\text{TO}} e_2$.

Intuition. RTO keeps a subset of TO's constraints: it keeps CO (Definition 11), TO constraints with a first resource usage (Line 7), and inter-thread TO-consistent constraints which change resource ownership (Line 13). Algorithm 1 does not introduce new other ordering constraints, so $\to_{\text{RTO}} \subseteq \to_{\text{TO}}$.

Proof sketch. Recall that $\to_{\text{RTO}}$ is defined as the transitive closure of: (i) program-order edges and (ii) the pairs in $E_{nonrelaxed}$ produced by Algorithm 1. Program order is preserved by $\to_{\text{TO}}$ (by Definition 6), so edges of type (i) are already in $\to_{\text{TO}}$.

For edges of type (ii), observe that Algorithm 1 never introduces a new ordering that is not already present in the input total-order list TO. Algorithm 1 adds an ordering edge for first-use resources when it inserts the adjacent pair $\{e_{i-1}, e_i\}$ into $E_{nonrelaxed}$ (Line 7); by construction, e_{i-1} immediately precedes e_i in the list TO, hence $e_{i-1} \to_{\text{TO}} e_i$. Algorithm 1 adds an ordering edge $e_{prev} \to_{\text{RTO}} e_i$ when the owner of a resource changes: $r(e_{prev}) = r(e_i) \wedge t(e_{prev}) \neq t(e_i)$ (Line 13). We also know that $e_{prev} \to_{\text{res}} e_i$ because both events have the same resource. Then, by Definition 6, we know that $e_{prev} \to_{\text{TO}} e_i$.

Therefore, every base edge used to generate $\to_{\text{RTO}}$ is already an edge of $\to_{\text{TO}}$. Given that $\to_{\text{TO}}$ is transitive, taking the transitive closure cannot produce an edge outside $\to_{\text{TO}}$ and thus $\to_{\text{RTO}} \subseteq \to_{\text{TO}}$. ◀

▶ **Theorem 13** (PO is no stronger than RTO). *For any execution E, the Partial Order relation is included in the Relaxed Total Order relation: $\to_{\text{PO}} \subseteq \to_{\text{RTO}}$.*

Equivalently, for all events $e_1, e_2 \in E$, if $e_1 \to_{\text{PO}} e_2$ then $e_1 \to_{\text{RTO}} e_2$.

Intuition. Under DRF, the cross-thread constraints captured by PO are the inter-thread handoff edges (where ownership of a resource changes). RTO keeps all such handoffs (as PO does), and adds more order by totally ordering first uses globally.

Proof sketch. By Definition 9, $\to_{\text{PO}}$ is the transitive closure of $\to_{\text{CO}} \cup \to_{\text{sync}}$. Thus it suffices to show $\to_{\text{CO}} \subseteq \to_{\text{RTO}}$ and $\to_{\text{sync}} \subseteq \to_{\text{RTO}}$.

First, $\to_{\text{RTO}}$ contains all program-order edges $\to_{\text{CO}}$ by construction.

Second, consider any $(e_1, e_2) \in \to_{\text{sync}}$ (Definition 7), so $e_1 \to_{\text{res}} e_2$ and $t(e_1) \neq t(e_2)$ (an adjacent cross-thread handoff on some resource). Event e_1 is the latest acquisition of the resource when e_2 happens, therefore, in Algorithm 1, when $e_i = e_2$ we know that $e_{prev} = e_1$.

Given that ownership differs, Algorithm 1 adds such edges directly to $E_{nonrelaxed}$ (Line 13), yielding the $e_1 \rightarrow_{\mathrm{RTO}} e_2$ edge (Definition 11).

Therefore $\rightarrow_{\mathrm{CO}} \cup \rightarrow_{\mathrm{sync}} \subseteq \rightarrow_{\mathrm{RTO}}$ and taking transitive closure gives $\rightarrow_{\mathrm{PO}} \subseteq \rightarrow_{\mathrm{RTO}}$.

Finally, RTO is often strictly stronger than PO: if a resource r is acquired for the first time by an event e, then PO imposes no inter-thread ordering on e (there is no prior owner to induce a $\rightarrow_{\mathrm{sync}}$ edge). In contrast, Algorithm 1 captures such first uses (Line 7) and adds them to $E_{nonrelaxed}$ following TO, thereby adding inter-thread order between e and other events. ◀

2.4 Correctness of RTO

Under the DRF assumption of Section 2, all inter-thread communication happens via synchronization operations on shared resources. Consequently, the observable behavior of an execution is determined by: (i) intra-thread program order, and (ii) the relative order of synchronization across threads.

▶ **Definition 14** (Observational equivalence). *For any execution E, let $\rightarrow_{\mathrm{HB}}$ be the induced happens-before relation on events of E, defined as the transitive closure: $(\rightarrow_{\mathrm{CO}} \cup \rightarrow_{\mathrm{sync}})^+$. Let E_{L} and E_{R} be two executions of the same DRF program (i.e., recorded in the **L**og and **R**eplayed). E_{L} and E_{R} are* observationally equivalent *(i.e., $E_{\mathrm{L}} \approx E_{\mathrm{R}}$) if there is a bijection $\phi : E_{\mathrm{L}} \rightarrow E_{\mathrm{R}}$ that matches events from E_{L} to E_{R} such that:*

1. *Same thread acquires same resource: $\forall e \in E_{\mathrm{L}} : t(e) = t(\phi(e)) \wedge r(e) = r(\phi(e))$*
2. *Preserves $\rightarrow_{\mathrm{HB}}$: $\forall e_1, e_2 \in E_{\mathrm{L}} : e_1 \rightarrow^{L}_{\mathrm{HB}} e_2 \implies \phi(e_1) \rightarrow^{R}_{\mathrm{HB}} \phi(e_2)$*

▶ **Lemma 15** (DRF reduction to preserving happens-before). *Consider an execution E that is* data-race free *(Definition 8), any replay execution E_R over the* same *program and the* same *set of synchronization events as E, such that E_R respects $\rightarrow_{\mathrm{HB}}$ (i.e., $\rightarrow_{\mathrm{HB}}$ is a subset of the replay order of E_R) (Definition 14). To establish correctness for a record/replay scheme on DRF executions, it suffices to show that the replay enforces $\rightarrow_{\mathrm{HB}}$. Then $E \approx E_R$.*

Proof sketch. Fix a DRF execution E. By the DRF assumption, all inter-thread communication that can affect visible outcomes is mediated by synchronization, and thus is ordered by $\rightarrow_{\mathrm{sync}}$. All intra-thread communication relevant to outcomes is ordered by $\rightarrow_{\mathrm{CO}}$. Therefore, any two executions that preserve all $\rightarrow_{\mathrm{CO}}$ edges and all $\rightarrow_{\mathrm{sync}}$ edges preserve the same happens-before relation $\rightarrow_{\mathrm{HB}}$.

Standard DRF reasoning implies that once $\rightarrow_{\mathrm{HB}}$ is fixed, the program's visible outcomes are uniquely determined up to reordering of independent steps. We use only the following consequence: reordering steps that are not ordered by happens-before cannot change the values read by the program or the externally visible I/O. Hence any replay E_R that respects $\rightarrow_{\mathrm{HB}}$ must be observationally equivalent to E.

Operational note: our event set E logs successful synchronization acquisitions; releases do not need to be logged explicitly because the lock semantics implies that the next successful acquisition of the same resource is ordered after the previous release, and this is precisely what $\rightarrow_{\mathrm{sync}}$ models (via $\rightarrow_{\mathrm{res}}$ across threads). ◀

▶ **Theorem 16** (Correctness of RTO for DRF executions). *Let E be a recorded execution and assume E is DRF as in Lemma 15. Let $\rightarrow_{\mathrm{RTO}}$ be the relation defined in Definition 11.*

Then any replayed execution E_{R} that enforces $\rightarrow_{\mathrm{RTO}}$ satisfies $E \approx E_{\mathrm{R}}$.

Proof sketch. By Theorem 13, $\rightarrow_{\mathrm{PO}} \subseteq \rightarrow_{\mathrm{RTO}}$. By construction of $\rightarrow_{\mathrm{PO}}$ (Definitions 7 and 9), $\rightarrow_{\mathrm{PO}}$ contains all $\rightarrow_{\mathrm{sync}}$ edges induced by inter-thread acquires on the same resource, and

$\rightarrow_{\text{PO}}$ also contains all $\rightarrow_{\text{CO}}$ edges. Therefore, $(\rightarrow_{\text{CO}} \cup \rightarrow_{\text{sync}}) \subseteq \rightarrow_{\text{PO}} \subseteq \rightarrow_{\text{RTO}}$, which implies $\rightarrow_{\text{HB}} \subseteq \rightarrow_{\text{RTO}}$. Hence any replay that enforces $\rightarrow_{\text{RTO}}$ necessarily enforces $\rightarrow_{\text{HB}}$.

Applying Lemma 15, any replay that respects $\rightarrow_{\text{HB}}$ is observationally equivalent to the recorded execution E. Thus enforcing $\rightarrow_{\text{RTO}}$ is sufficient for correctness. ◀

3 Design

This section describes the design of an RR system that uses RTO to capture inter-thread orderings in a recording log, which can later be replayed to observe the same execution (Section 3.1), and the design of a *novel combination of MVX with RR* to improve the performance of *recording* I/O bound programs (Section 3.2).

3.1 Capturing and enforcing Relaxed Total Order (RTO)

This section expands the formalization in Section 2 to describe how to design a system that captures and enforces RTO. Note that we define RTO from the sets of edges between synchronization acquisition operations $E_{nonrelaxed}$ and $E_{relaxed}$ using Algorithm 1, which takes as input a list of events ordered by Total Order (TO). As such, it is reasonable to assume the recorder already captures TO, and the replayer already enforces it. In this section, we explain how to design a recorder that captures RTO by building such sets from TO, and replayer that enforces RTO.

3.1.1 Capturing TO with one global vector clock

One common way to capture TO in both RR and MVX to use *one global vector clock (VC)* with one vector entry per thread. To record each successful synchronization operation on thread T_i, the system increments position i in the VC and scans the contents of the whole VC to obtain a unique *timestamp (TS)* that totally orders the event (thus capturing $\rightarrow_{TO}$ used in Section 2). Furthermore, the increment and scan must be atomic.

For instance, Figure 2 shows an execution in which the recorder logs the order between events e_0 and e_1 by incrementing the VC from $(0,0)$ to $(1,0)$ in T_1 and then to $(1,1)$ in T_2. The TO recorder saves those events in the log together with their respective TS. Later, a replayer can ensure the same order of events, even if T_2 reaches e_1 first because the TS of e_1 is $(1,1)$ and incrementing the replayer VC yields $(1,0)$, which does not match the recorded TS $(1,1)$. The replayer thus pauses T_2 until the VC updates to a compatible state, which happens once T_1 reaches e_0.

3.1.2 Capturing RTO

Algorithm 1 iteratively processes a sequence of events in a total order. Implementing such an abstraction directly requires storing an unbounded number of old events for an unbounded amount of time in *lastAcquisition*, which is hard to implement efficiently in practice. Instead, our design captures the RTO order via a combination of each resource's *last owner* (the thread which last acquired the resource) with the global VC already present to capture TO. This is a conservative approach that, in the worst case, captures some stricter TO orderings by accident. However, such a pragmatic approach results in the dramatic performance improvement we report in Section 5.

Tracking $E_{relaxed}$ via resource ownership. Our design associates each resource with its last owner, initially set to $t_\perp$ in Definition 2 in Section 2.1 (*i.e.,* no thread has acquired this

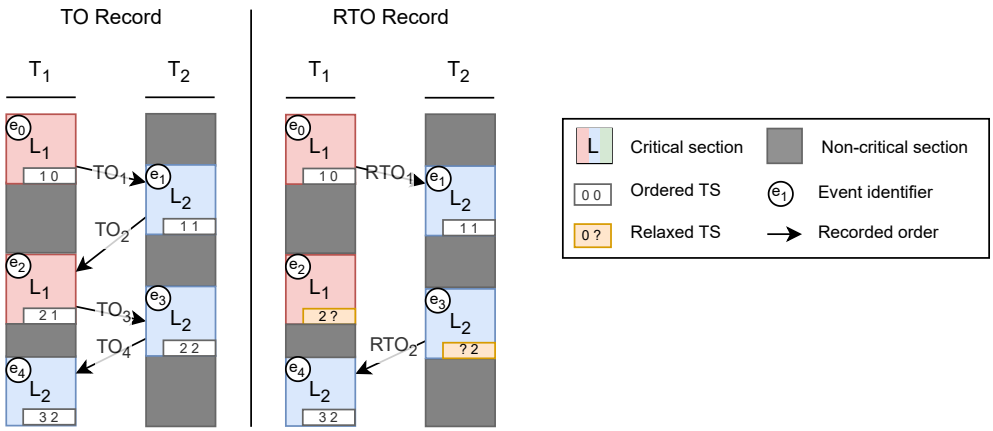

Figure 2 Possible execution recorded using TO and RTO, together with TO time-stamps and RTO relaxed events.

resource yet). Tracking the last owner still allows the recorder to distinguish between relaxed and non-relaxed edges (Line 10 in Algorithm 1). For instance, in Figure 2, notice that the last owner of L_1 on e_2 does not change, which allows T_1 to make progress without any RTO constraints (equivalent to Line 11 in Algorithm 1).

Incrementing a VC during recording involves returning a timestamp (a snapshot of the VC after the increment) to write to the log. Later, during replay, incrementing a VC synchronizes on the timestamp read from the log, potentially waiting until the global VC matches the recorded timestamp. Both the snapshot and the synchronization are costly operations that can be skipped for relaxed events. To support efficient RTO, our design requires the ability to increment the VC without making a snapshot or synchronizing. We denote such increments in our example as $(2, ?)$ as we ignore other threads' entries. Given that replaying RTO relaxes the ordering, e_2 may execute before or after e_3, which results in VC $(2, 1)$ or $(2, 2)$ after T_1's increment, respectively.

Tracking $E_{nonrelaxed}$ via the global VC. When resource ownership changes (past first usage), RTO imposes an ordering relationship with the previous event that acquired the same resource (Line 13 in Algorithm 1). In the example we are following, this is the ordering RTO_2 in Figure 2 between e_3 and e_4. Our design tracks e_{prev} indirectly using the pre-existing VC required for TO via the TS in ordered events. In this case, e_4's TS is $(3, 2)$. Note that T_1's VC entry, "blindly" incremented when relaxing e_2, still matches T_1's TS on e_4: 3. This is the reason why relaxing events still requires incrementing the VC, so that a later ordered TS still matches. Note also that all the other threads' entries (only T_2 in this case) capture the RTO ordering: 2. As a result, the recorder need not to record $(2, ?)$ for relaxed events such as e_2 (a `RELAXED` tag suffices) because a later ordered event (such as e_4) carries a TS that establishes the ordering constraints.

Fast relaxed events at the cost of stricter non-relaxed constraints. Our design uses a global VC and tracks ownership to minimize overhead when executing relaxed events. *Relaxed events are never ordered in any way and require no waiting on the replayer.*

However, our design may impose stricter restrictions on non-relaxed events than the intended RTO constraint. For instance, consider the execution in Figure 3. The key RTO constraint is RTO_2 between T_1 and T_2, which ensures that e_3 executes after e_0. If the replayed T_1 manages to reach e_3 earlier still in an RTO-compliant instant (*e.g.,* concurrently

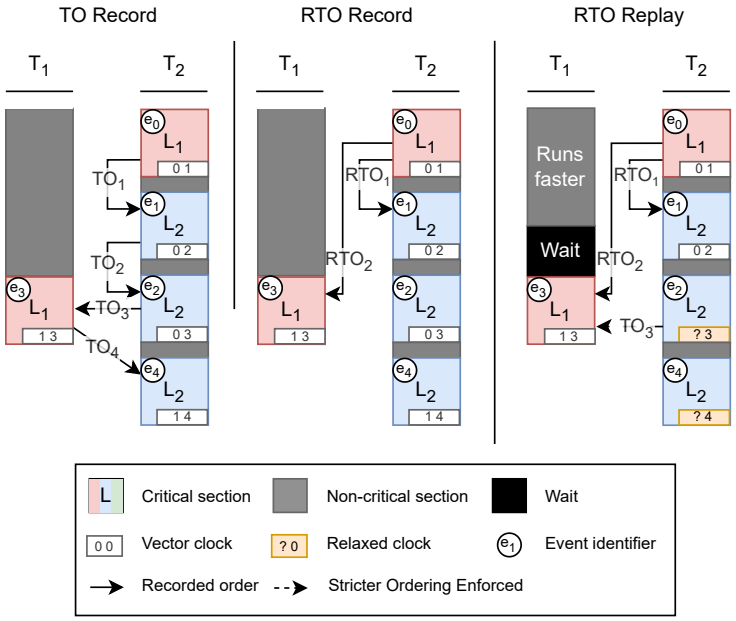

Figure 3 Depiction of a possible execution: recorded with TO, recorded with RTO, and replayed with RTO. Note that capturing RTO edges using a global vector clock may result in stricter than necessary orderings. In this case, e_3 is replayed concurrently with e_2 but nevertheless needs to wait for e_2 to finish, due to the recorded time-stamp.

with e_1), e_3's TS $(1, 3)$ causes the replayer to needlessly synchronize until the VC matches a later unrelated event e_2.

The unfortunate fact that ordering events is sometimes stricter than necessary is a worthwhile trade-off for two reasons. **First**, synchronization resources are typically sparsely used [10,31], and split between many threads to minimize contention. If a thread synchronizes uncontended on a resource once, chances are it will do so again often in the near future. Furthermore, relaxed events naturally support idiomatic Java code such as wait-notify and reentrant locks (discussed more in Sections 4.1.1 and 4.1.2, respectively). **Second**, ordering events using the VC is already an expensive operation that requires heavyweight inter-thread communication. We expect that imposing a stricter ordering results in slightly more waiting.

In short, our design is guided by the principle of making an uncommon slow operation slower to ensure a common fast operation is faster.

3.2 Hybrid MVX/RR

RR typically operates under the assumption that recorded operations are sparse. Even though such an assumption typically holds for CPU-bound workloads with buffered I/O, it breaks for I/O-bound workloads, resulting in pathologically high performance overhead (as high as 4.6x–8.3x, see Section 5.5). Figure 4-RR shows why: the (online) recorder effectively doubles the I/O performed — once with the underlying OS, and a second time to record operations in the log. One possible solution is to simply skip such I/O and require the replaying system to match the state of the system that captured the recording [8, 9, 15]. Unfortunately, such an approach negates one of the main strengths of RR: one system can record a log that a completely different system can still replay [22]. Another option is to spawn a new process dedicated to write the log [21], which slows the recorder as it still

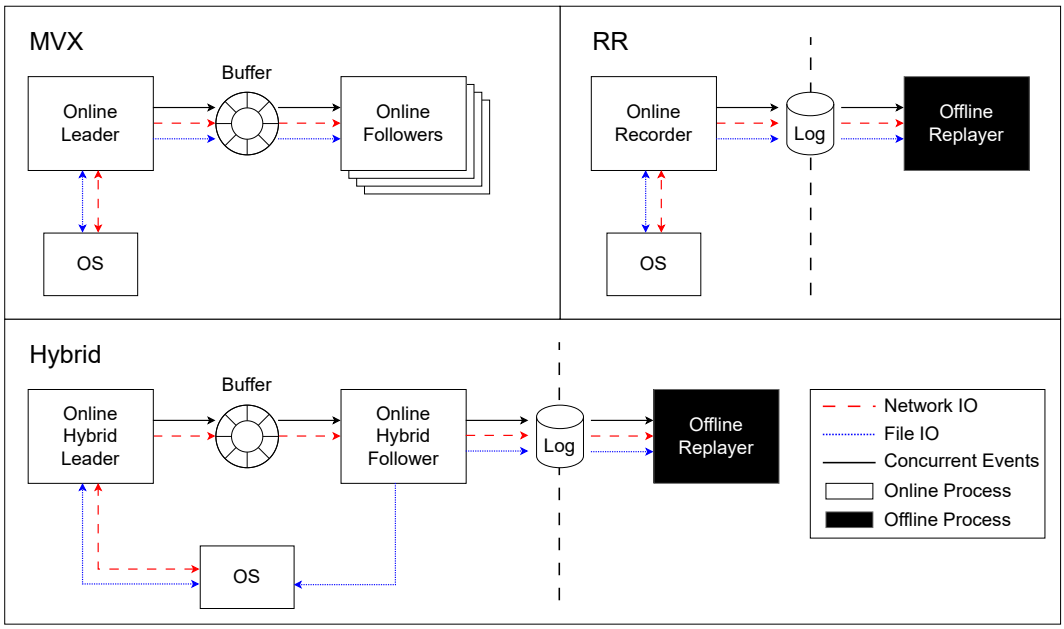

Figure 4 RR, MVX, and Hybrid MVX+RR architectures.

doubles the I/O performed, even if using an more efficient inter-process communication mechanism.

To provide efficient support for recording I/O-bound workloads, we propose a novel combination of Multi-Version eXecution (MVX) with RR. Modern MVX architectures [14, 23, 24, 27, 34, 36], depicted in Figure 4-MVX, deploy many *variants* executing diversified versions of the same program (*e.g.,* combining incompatible dynamic analyses over the same execution [23]) with one *leader* acting as a recorder and generating an online log (*buffer*) of non-deterministic operations that potentially many *followers* drain to obtain the same non-determinism for the same operations. The user always interacts with the leader, as the followers execute in the background (*e.g.,* "catching-up" as they run heavy dynamic analyses after a native leader, effectively masking the overhead/latency [23]).

We propose a *hybrid MVX/RR recorder*, as shown in Figure 4-Hybrid that generates recording logs replayable by unmodified (offline) recorders. The main insight is that a follower variant can access the same system state as the leader while the leader executes. Building on that insight, we use propose a recorder that deploys the same program as both leader and follower using a modified MVX system that only buffers events from the leader that the follower cannot access on the same system (*e.g.,* network I/O and concurrency-related events). The follower writes the recording log by executing the same program as the leader, obtaining as much information as possible from the (same) underlying system without involving the leader, and obtaining as little information as needed from the leader via the MVX buffer. As a result, the leader executes unimpeded by logging large amounts of I/O that the trailing follower can replicate and log accurately.

4 Implementation

We implemented a prototype for Relaxed Total Order (RTO) and Hybrid MVX/RR on on top of a state-of-the-art MVX and RR system for Java — JMVX [28] — which support

```
18:  //Recorder
19:  void monitorenter(Object o){
20:    Thread t = currentThread();
21:    JMVX.monitorEnter(o);
22:    Clock cl = globalClock.inc(t);
23:
24:    //cannot track
25:    if(!o instanceof OwnedObject){
26:      record(TOTAL, cl);
27:      return;
28:    }
29:
30:    //can track
31:    OwnedObject u =(OwnedObject)o;
32:    if (u.owner == t) { //relaxed
33:      record(RELAXED, null);
34:    } else { //total order
35:      u.owner = t;
36:      record(TOTAL, cl);
37:    }
38:  }
```

```
39:  //Replayer
40:  void monitorenter(Object o){
41:    Thread t = currentThread();
42:    //read from log
43:    Event e = readLog();
44:    switch(e.tag){
45:      case TOTAL:
46:        globalClock.sync(e.clockCopy);
47:        //fallthrough
48:      case RELAXED:
49:        JMVX.monitorEnter(o);
50:        globalClock.inc(t);
51:        return;
52:    }
53:  }
```

Figure 5 Implementation of acquiring a lock (`monitorenter`) for the recorder (left-hand side) and the replayer (right-hand side).

multi-threading by capturing the Total Order (TO) of synchronized events via a global vector clock, and combines MVX and RR in a single system. Furthermore, the source code [1] and a research artifact [29] are publicly available. We believe the techniques we describe in Section 3 can be applied to other languages and MVX/RR systems that capture TO using vector clocks besides Java and JMVX [14, 23].

4.1 RTO Implementation

Recall from Definition 1 in Section 2.1 that, for each synchronization acquisition event e, we require the resource $r(e)$ and the thread $t(e)$. Also, our design (Section 3.1) requires a mechanism to associate each resource to the last thread that acquired it (instead of the last acquisition event used in Algorithm 1).

JMVX intercepts monitor acquisitions via instrumentation that redirects them to method `<role>.monitorenter` as shown in Figure 5 for roles `Recorder` and `Replayer`. For each acquisition, JMVX captures the resource $r(e)$ as argument `o`, and thread $t(e)$ as the current thread `t`. We note that JMVX produces per-thread logs, so all entries in a given log belong to the thread using that log. The only information missing is the last thread that acquired the resource.

Keeping a map from each resource to the last owner is not practical for two reasons. First, any object in Java can be used as a synchronization resource, and concurrent Java programs rely on fine-grained locking (*e.g.,* highly concurrent data-structures in `java.util.concurrent`, used by many Java programs), resulting in a large map in memory. Second, such a map is subject to high contention as many threads attempt to access last owners in parallel.

Instead, we extend JMVX's instrumentation pass to: add a field to all classes to track the last owner. We note that we cannot add such field to the base class `java.lang.Object`

because doing so breaks internal JVM assumptions about the sizes of objects and offsets of certain fields (*e.g.,* the field that keeps the native stack trace on `java.lang.Exception`) [5, 25, 28], resulting in a crash. We modify the class hierarchy instead so that all direct subclasses of `java.lang.Object` (including classes without any explicit superclass) extend `jmvx.OwnedObject` instead, which in turn has field `jmvx.OwnedObject.lastOwner`. We represent the lack of previous owner ($t_\perp$ in Definition 2) simply as `null`. Modifying classes inside the package `java.lang` results in crashes similar to the crash experienced with `java.lang.Object`, we believe the crashes have the same root cause (*i.e.,* our changes break assumptions on the size and layout of certain objects). As a consequence, we take a conservative approach and never relax events for which the resource is an object belonging to any class in `java.lang`.

The left-hand side of Figure 5 shows how the recorder implements RTO. First, the recorder acquires the monitor (Line 21) and increments the global vector clock, obtaining a copy that can be used as a time-stamp (Line 22). Line 25 checks if the object is instrumented; *i.e.,* is not part of `java.lang`. If it is not, then the recorder logs the vector clock in a nonrelaxed fashion and returns (Lines 26–27). Next, the recorder determines if the monitor's acquisition can be relaxed (Line 32) via check adapted from Algorithm 1 (line 10). If the last owner matches the current one, then the synchronization can be relaxed (Line 33), and the recorder logs a relaxed marker. Otherwise, the recorder updates the object's owner (Line 35) and records the time-stamp obtained above in the log (Line 36).

The right-hand side of Figure 5 shows the replayer implements RTO. First, the replayer obtains the next event from the recording log (Line 43). A totally ordered event (Line 45) causes the replayer to *synchronize* the provided time-stamp with the global vector clock (Line 46), which in turn causes the thread to wait until *at least all prior (with respect to the total order) monitor entry events complete.* Relaxed markers skip synchronization (Line 48). Either way, the replayer acquires the lock (Line 49). Note that the replayer always increments the clock (Line 50) so that the time-stamps of future events matches.

4.1.1 `wait/notify`

In Java, `Object.wait` releases the monitor and blocks until another thread acquires the same monitor and calls `Object.notify` on the same object. Such `wait` calls inherently involve a transfer of ownership and cannot be relaxed. However, it is a good practice [12] to call `wait` with a timeout that expires if no other thread notified the same object. In that case, the timeout causes `wait` to re-aquire the same monitor and return. RTO naturally relaxes (common) expired `wait` calls.

4.1.2 Reentrant monitors

In Java, it is idiomatic for synchronized methods to call other synchronized methods. For instance, Figure 6 shows a possible method `isEmpty` implemented by calling method `size`, both synchronized. Given that both methods are synchronized, the program calls `monitorenter` twice on the same object: once when entering method `isEmpty`, and a second time when entering method `size`. We note that RTO naturally relaxes this very common case, as the same thread is guaranteed to be the last owner when re-entering a lock.

4.1.3 Spinning vs waiting

JMVX's design assumes that replayer threads experience short waits for events on a global vector clock synchronization (Line 46), because enforcing a total order naturally keeps the

```
54: synchronized boolean isEmpty(){
55:    return this.size() > 0;
56:  }
```

```
57: synchronized int size(){
58:    //compute the size
59:    return ...
60:  }
```

Figure 6 An example of code which acquires a monitor multiple times. Calling `isEmpty` acquires the monitor once, and the internal call to `size` acquires it again.

progress of threads clustered together (*i.e.,* the difference between elements in the global vector clock is small). When a replayer thread is much faster than in the recording, the total order causes such fast thread to wait to synchronize with the recorded total order. Such a fast thread cannot keep making progress until other threads catch up.

Given such short waits, JMVX uses busy waiting (*i.e.,* spinning) to ensure threads waiting on vector clock synchronization react as quickly as possible to an update as other threads "catch-up". **RTO fundamentally breaks the "short waits" assumption.** Using RTO, synchronizing on the vector clocks happens only ownership changes: a thread attempts to acquire an object that was acquired by another thread. Such an observation has two important consequences: (1) there is a dramatically lower number of total order synchronization events, and (2) synchronizing on the vector clock may take a longer time, which defeats the idea of spinning while waiting in Line 46.

Turning frequent small waits into infrequent longer waits means that spinning is not the best approach anymore. Instead, we can use `wait/notify`. As explained in Section 4.1.1 above, calling `wait` suspends the current thread until another thread calls `notify` on the same object (or a timeout occurs).

Deciding between spinning and waiting depends on the program being run. For instance, when there are more threads than CPUs, waiting effectively releases a CPU so that a ready thread that can make progress. On the other hand, spinning can react to vector clock updates much faster than waiting. Given that there is no clear winner, we implemented a hybrid strategy that adapts between spinning and waiting based on the perceived behavior of the target program. First, threads wait by spinning until they have observed 1,000 failed spin loops, which indicates that the current synchronization operation is under contention. In that case, the thread backs-off to a padded array to avoid cache-coherence traffic. Threads track the ratio of back-offs to successful synchronizations, and, when it becomes larger than 2, threads snoop on the other threads' ratio to compute the global ratio of the whole program. Our implementation changes to waiting when the ratio is below $\frac{1}{2}$ or above 2. If the ratio is between $\frac{1}{2}$ and 2, our implementation spins.

4.2 Hybrid MVX/RR

To implement the hybrid MVX/RR we describe in Section 3.2 (Figure 4), we started by modifying the existing JMVX's leader to skip most file I/O operations, creating the new *recorder-leader (RL)* variant. We then created a *recorder-follower (RF)* variant in two steps: (1) combining the existing follower with the existing recorder, to create the recording log by draining the buffer from the leader; and (2) issuing the same file I/O operations that the RL skips to the underlying OS. Step 2 allows the RF to log the same file I/O as the RL with zero communication between variants, or any extra I/O from the RL to generate the log.

We note that not all file I/O operations can be skipped, in particular: checking the status of files (*i.e.,* the `stat` family of operations), and deleting files. **First**, it is idiomatic for programs to check if a file exists before creating it. Such a check fails on the RL, causing it to

create the file. Then, the same check succeeds on the RF, causing a divergence. Our proposed hybrid MVX/RR still communicates operations that check the status of files. **Second**, it is also idiomatic to create files, read data from them, and then delete them (*e.g.,* extracting a compressed file from an archive). Such programs may result in the RL deleting the file before the RF has a chance to open it, resulting in a divergence. Our proposed hybrid MVX/RR does not delete files from the RL, postponing them until the RF reaches the same operation.

Finally, we note that random access files in Java are always opened for both reading and writing. Allowing the RF to read from a random access file that the RL is actively writing results in the RF observing corrupted data, which in turn leads to a divergence. Both variants track uses of random access files, so that the RL skips communicating reads until it observes a write. After the first write, the RL sends reads to the RF.

4.3 Limitations

As we explain in Section 4.1, our prototype does not support relaxing operations on objects being used as synchronization resources that belong to the `java.lang` package. We also inherit all Jmvx's limitations, as it is our base system, such as assuming the target programs do not use non-blocking synchronization.

5 Evaluation

In this section we evaluate our prototype implementation, answering the following Research Questions (RQs):

- **RQ1**: What is the runtime performance improvement when using RTO for RR?
- **RQ2**: What is the impact on memory usage when using RTO for RR?
- **RQ3**: What is the runtime performance improvement when using RTO for MVX?
- **RQ4**: How much memory does RTO save in recordings?
- **RQ5**: Which vector clock synchronization strategy yields the lowest runtime overhead?
- **RQ6**: Do RR systems suffer from poor performance on IO bound workloads?
- **RQ7**: What is the runtime performance improvement when using Hybrid MVX/RR to record IO heavy programs?
- **RQ8**: What is the runtime performance impact when using Hybrid MVX/RR to record common workloads?

To answer a majority of these questions we will analyze how our changes impact the performance of Jmvx. Unfortunately, it is difficult to make direct comparisons to other RR and MVX systems. With respect to Java based RR systems: Chronicler [6] does not support replay; Octet [8, 9] and DejaVu [16] use different VMs with custom modifications; LEAP [15] does not record IO nor does it support the same VM version (JDK 7 and lower due to a dependency); and JaRec [11] is not publicly available. In addition, Octet and LEAP make different assumptions on DRF, making comparisons against Jmvx unfair. Other lower level RR systems [17, 19, 21, 22, 32], would have to record and replay the JVM, including inherently non-deterministic features like the garbage collector and just-in-time compiler. Such systems either outright crash or suffer from poor performance due to recording/replaying the underlying VM [28]. To our knowledge there is no other MVX system besides Jmvx that supports the JVM.

Section 5.5 uses the industry-strength `rr` [22] to demonstrate the severity of IO overhead. For fairness (*i.e.,* so `rr` does not have to record and replay the entire JVM), we compare `rr` executing native `dd` against a Java equivalent for Jmvx.

Table 2 Comparison between the JMVX configured to act as close to its original publication as possible and JMVX's original published performance. Old columns are taken from the paper, and we recomputed the averages to account for the missing benchmark "h2 server".

	Harness Runtime (msec)	Overhead			
Benchmark	Vanilla	Recorder	Old Rec.	Replayer	Old Rep.
avrora	12333.333 +- 35.921	1.137x	1.26x	0.698x	1.28x
batik	8941.000 +- 100.792	1.071x	1.08x	1.066x	1.04x
fop	7693.667 +- 65.041	1.211x	1.18x	1.287x	1.17x
h2	11083.333 +- 137.849	1.438x	1.62x	2.169x	2.60x
jme	9113.667 +- 19.732	1.065x	1.05x	1.080x	1.15x
jython	29004.000 +- 148.354	1.651x	1.87x	1.643x	2.14x
luindex	2403.000 +- 30.315	1.233x	1.35x	1.121x	1.65x
lusearch	2373.667 +- 147.242	1.253x	1.32x	3.494x	4.10x
pmd	16072.000 +- 319.526	1.159x	1.11x	2.659x	2.31x
sunflow	31570.333 +- 719.952	0.924x	0.90x	1.001x	0.96x
xalan	8445.667 +- 281.521	1.171x	1.17x	2.208x	1.95x
AVG	—	1.210x	1.26x	1.675x	1.85x

5.1 Experimental Setup

We use the Dacapo benchmark suite [7] to evaluate the performance of JMVX [28] after implementing RTO and the Hybrid RR/MVX. We use the same benchmarks used to evaluate JMVX, except for the h2 server variant (we do test on the in-memory variant included with Dacapo). We follow a similar evaluation to JMVX, and use h2, luindex, and lusearch from Dacapo version 9.12 (dubbed *Bach*), and the rest of the benchmarks from version 23.11 (the newest version at the time of writing, dubbed *Chopin*).

To evaluate the hybrid MVX/RR system, we use a Java implementation of `dd` [26], and compare the overhead of our prototype against `rr` [22] recording the same workload using regular `dd`.

For each benchmark we execute the *vanilla* uninstrumented benchmark 10 times. Then we execute the record/replay phases 10 times. We report overhead the average record/replay time normalized to the average vanilla runtime. For hybrid MVX/RR, recording time is the leader's execution time as that is the variant a user would interact with.

We conducted all experiment on a NUMA machine running Ubuntu 22.04.3 LTS, equipped with four Intel(R) Xeon(R) Gold 5318H CPU nodes, each with 194GB of RAM. We limited experiments to an single node (18 physical cores with hyper-threading disabled). RR experiments were allotted half of the available cores. Hybrid MVX/RR experiments allot half of the cores to the leader and the other half of the cores to the follower. When possible, we limited the Dacapo benchmark to use 9 threads, but some benchmarks ignore the setting.

5.2 Baseline

Since its original publication, JMVX has undergone many *"quality of life"* code improvements that may have shifted the original performance [28]. We established the baseline performance by rerunning the benchmarks reported in JMVX's original paper with the same configuration as the original publication: using total ordering (Section 3.1), and spin based backoffs for the vector clock (Section 4.1.3). Table 2 shows the results for all benchmarks we use. The baseline recording performance remains comparable, and replaying is now 21% faster.

Table 3 Performance comparison of JMVX's RR mode with RTO and the wait-notify vector clock enabled.

(a) Recorder

Benchmark	TO	RTO	Delta
avrora	1.137x	1.116x	-0.021
batik	1.071x	1.067x	-0.004
fop	1.211x	1.203x	-0.008
h2	1.438x	1.049x	**-0.389**
jme	1.065x	1.065x	0.000
jython	1.651x	1.422x	**-0.229**
luindex	1.233x	1.203x	-0.030
lusearch	1.253x	1.281x	0.028
pmd	1.159x	1.124x	-0.035
sunflow	0.924x	0.898x	-0.026
xalan	1.171x	1.252x	0.081
AVG	1.210x	1.153x	-0.057

(b) Replayer

Benchmark	TO	RTO	Delta
avrora	0.698x	0.707x	0.009
batik	1.066x	1.066x	0.000
fop	1.287x	1.296x	0.009
h2	2.169x	1.577x	**-0.592**
jme	1.080x	1.078x	-0.002
jython	1.643x	1.448x	**-0.195**
luindex	1.121x	1.084x	-0.037
lusearch	3.494x	1.462x	**-2.032**
pmd	2.659x	1.486x	**-1.173**
sunflow	1.001x	1.029x	0.028
xalan	2.208x	2.254x	0.046
AVG	1.675x	1.317x	-0.358

5.3 RTO Optimization

To understand the impact of RTO (Section 3.1), we recorded and replayed the execution of all benchmarks in the DaCapo benchmark suite compatible with JMVX. We compare against JMVX's original Total Order (TO) implementation.

5.3.1 Performance overhead

Table 3 shows the results as reported by DaCapo's test harness when comparing the record and replayer modes of JMVX with TO and RTO. The overhead presented in the table was computed by normalizing the average of 10 runs (without warmups) to the average of 10 uninstrumented runs of the benchmark.

The results show that RTO reduces the overall performance overhead in half for both recording, from **21.0%** to **15.3%**, and replaying, from **67.5%** to **31.7%**. RTO captures fewer events, which in turn translates to faster recording and replaying times. On the other hand, replaying strictly TO involves lengthy wait times for all threads to reach the same scheduling. Taking a closer look at particular benchmarks that experience high concurrency (h2, lusearch, and pmd), we can see a significant improvement when replaying because RTO allows the replay to relax such wait times. Jython is single-threaded but uses a large amount of synchronization to enforce single-threaded semantics when executing Python code in multi-threaded Java. In this case, RTO naturally relaxes *all* lock acquisitions because there is only one owner for all locks, resulting in a faster recording and replay. As expected, benchmarks that do not experience such high levels of concurrency do not see a clear benefit (or penalty), with results within experimental error.

Answer to RQ1: RTO reduces the overall performance overhead of recording and replaying by half, from **21.0%** to **15.3%**, and from **67.5%** to **31.7%**, respectively. As expected, replaying benefits the most. Focusing on replaying benchmarks with high concurrency (h2, lusearch, and pmd), the overhead drops dramatically from **177.4%** to only **50.8%**. RTO also helps benchmarks that use synchronization on a single thread (*e.g.,* Jython), dropping the overhead from **64.3%** to **44.8%**. We note that RTO does not hurt performance (slower results are within experimental error).

Table 4 JMVX's MVX mode with RTO.

	Leader		Follower	
Benchmark	TO	RTO	TO	RTO
avrora	1.212x	1.191x	1.201x	1.648x
batik	1.067x	1.075x	1.011x	1.020x
fop	1.153x	1.160x	1.158x	1.160x
h2	1.437x	1.373x	2.366x	**1.924x**
jme	1.047x	1.045x	1.228x	1.200x
jython	1.761x	1.800x	1.911x	1.965x
luindex	1.299x	1.278x	1.298x	1.277x
lusearch	1.313x	1.274x	3.309x	**1.476x**
pmd	1.109x	1.096x	2.462x	**1.387x**
sunflow	0.956x	0.975x	1.059x	1.043x
xalan	1.497x	1.297x	2.095x	1.510x
AVG	1.259x	1.233x	1.736x	1.419x

Regarding memory overhead, our approach has the baseline overhead from JMVX plus one field per object (not belonging to the `java.lang` package) to track the last owner (Section 4.1). We ran the same memory overhead experiment from the original JMVX publication for recording [28], and measured 1.15x memory overhead (in line with the original 1.22x).

Answer to RQ2: Capturing RTO does not impact the memory overhead significantly when compared to the original TO implementation, changing the overhead from 1.22x to 1.15x.

Given how flexible JMVX is, our prototype implementation of RTO works for MVX out of the box. We ran a second experiment, similar to the record-replay described above, but using MVX with one leader and one follower instead. Once again, the results presented are the average of 10 runs, without warmups, normalized against the average time of the uninstrumented benchmark. Table 4 shows the results. The results are similar to the replay results described above, with the follower benefiting the most for benchmarks that are highly parallel — h2, lusearch, and pmd.

Answer to RQ3: RTO also benefits MVX, reducing the performance overhead in the follower from **73.6%** to **41.9%**, with the same highly concurrent benchmarks benefiting the most (h2, lusearch, and pmd) see a combined reduction from **171.2%** to **59.6%**. The benefits for the leader are smaller, from **25.9%** to **23.3%**. However, similarly to record/replay, RTO never hurts performance.

5.3.2 Log sizes

Our implementation of RTO records relaxed events using a single `RELAXED` tag instead of a time-stamp, which reduces the amount of data that needs to be recorded, which in turn reduces the amount of IO performed. Table 5 shows the proportion of optimized events per benchmark when using RTO. In addition, it shows the size reduction between an RTO log and a TO one.

The number of synchronization operations confirms the trends observed in Section 5.3. The benchmarks that benefit the most (h2, jython, lusearch, and pmd) see the highest reductions of events — **92%** at least — with a similar reduction in overall log size.

Table 5 Recording sizes after compression (with gzip) when recording with TO and RTO. *Monitor Events* shows the total number of logged monitor events between both `monitorenter` and `wait` events in all threads. *Relaxed* represents the proportion of vector clocks replaced with relaxed tags when using RTO. A high proportion suggests that the benchmark experiences time locality when accessing locks (*i.e.,* when a thread accesses a lock, it is likely that it will access the same lock again in the near future).

Benchmark	TO (Mb)	RTO (Mb)	Monitor Events	Relaxed
avrora	24.945	22.527	2601539	0.278
batik	150.672	149.404	261120	0.652
fop	35.890	35.304	35661	0.536
h2	74.174	**16.178**	26383709	**0.994**
jme	424.156	424.149	2477	0.833
jython	205.782	**142.816**	32069316	**0.981**
luindex	22.736	**13.345**	250006	**1.000**
lusearch	119.742	**91.906**	1919219	**0.999**
pmd	38.448	**24.774**	4621309	**0.920**
sunflow	14.911	14.889	1257	0.024
xalan	73.088	45.960	260597	0.012
AVG	107.686	89.205	6218746.363	0.657

Answer to RQ4: Using RTO allows the recorder to log less information reducing the average log sizes from **107.686 Mb** to **89.205 Mb** (a **13.3%** reduction). In highly concurrent benchmarks (h2, jython, lusearch, and pmd) at least **92%** of synchronization events use less storage, leading to an average of **37.1%** log size reduction for those benchmarks.

5.4 Synchronization Strategies

As we discuss in Section 4.1.3, the original version of JMVX assumes frequent vector clock synchronization with short wait times in between, an assumption which RTO invalidates.

We measured the impact of different implementations of the vector clock by replaying all DaCapo benchmarks, with both TO and RTO, and the synchronization strategies we propose: spin, wait, and adaptive. *Spin* uses the spinning back-off array from JMVX's original publication [28]. *Wait* relies on `wait-notify` as opposed to spinning. Finally, *adaptive* switches between the two based on the ratio of syncs to back-offs (as described in Section 4.1.3). Table 6 shows the results. We note that recording does not require waiting for particular synchronization operations, so we exclude it for this experiment.

The results show that combining wait-notify with RTO reduces the performance overhead on the replayer by half, from **67.5%** to **35.4%**. However, benchmarks based on frequent fast synchronization operations — avrora and luindex — are still faster when using spinning. Attempting a balance that achieves the *"best of both worlds"* further reduces the TO overhead from **63.0%** to **51.4%**, and the RTO overhead from **35.4%** to **31.7%**.

Answer to RQ5: The wait-notify strategy works well for most workloads that do not require frequent fast synchronization, reducing the overhead from **67.5%** to **63.0%** for TO and from **43.8%** and **35.4%** for RTO. Our proposed adaptive synchronization technique, that starts by spinning and moves to waiting progressively, further drops the overhead to **51.4%** for TO and **31.7%** for RTO.

Table 6 Replayer's performance using difference vector clock strategies and orderings. Spin uses the spinning back-off optimization from JMVX's older version [28]. Wait uses wait-notify and Adaptive alters between the two strategies based on the ratio of syncs to back-offs.

	Spin		Wait		Adaptive	
Benchmark	TO	RTO	TO	RTO	TO	RTO
avrora	0.698x	0.683x	1.483x	1.617x	0.704x	0.707x
batik	1.066x	1.064x	1.080x	1.070x	1.076x	1.066x
fop	1.287x	1.281x	1.286x	1.279x	1.295x	1.296x
h2	2.169x	1.788x	1.978x	1.529x	2.095x	1.577x
jme	1.080x	1.074x	1.090x	1.083x	1.081x	1.078x
jython	1.643x	1.456x	1.775x	1.579x	1.727x	1.448x
luindex	1.121x	1.076x	1.155x	1.120x	1.102x	1.084x
lusearch	3.494x	1.828x	3.766x	1.462x	2.590x	1.462x
pmd	2.659x	2.343x	1.492x	1.374x	1.741x	1.486x
sunflow	1.001x	0.985x	1.003x	0.983x	1.040x	1.029x
xalan	2.208x	2.245x	1.825x	1.793x	2.208x	2.254x
AVG	1.675x	1.438x	1.630x	1.354x	1.514x	1.317x

Table 7 Results from recording `dd` via `rr` and `jdd` via JMVX. Each program was set to copy 100 MBs from `/dev/zero` to `/dev/null`. The runtime for `dd` without recording is presented in the leftmost "Vanilla Runtime"; this value was taken from the output of `dd` itself. The rightmost "Vanilla Runtime" column shows the runtime for the uninstrumented and unrecorded *jdd*. Overheads for `rr` are normalized to `dd`'s runtime. Similarly, overheads for JMVX's Recorder as well as our Hybrid MVX/RR are both normalized to `jdd`.

	dd		Java dd		
Buffer Size	Vanilla Runtime (msec)	rr	Recorder	Hybrid	Vanilla Runtime (msec)
1 bytes	12.294 +- 0.163	8.283x	1.130x	**1.021x**	201966.100 +- 3634.945
512 bytes	13.219 +- 0.255	8.475x	1.726x	**1.116x**	495.300 +- 4.762
8192 bytes	27.281 +- 0.166	7.428x	4.371x	**1.448x**	88.400 +- 3.307
16382 bytes	42.253 +- 0.125	6.697x	4.472x	**1.435x**	81.300 +- 4.762
AVG	—	7.721x	2.925x	**1.255x**	—

5.5 MVX Recorder

Our intuition is that RR systems struggle on IO bound programs. We recorded `dd`, a simple program which copies bytes from one stream to another using a provided block size, with the industry grade tool `rr` to see if IO is a pathological performance case. Our selection of `dd` was inspired by the experimental evaluation of SaBRe [4]. In addition, we implemented a simple Java version of `dd`, which we refer to as `jdd`, and measured JMVX's performance as well. We used three buffer sizes — 1 byte, 512 bytes (`dd`'s default), and 8192 bytes (8 kbs, Java's `BufferedInputStream` default) — to copy 100MB from `/dev/zero` to `/dev/null`. Table 7 shows the results for `rr`, JMVX's regular recorder, and our MVX recorder approach.

Answer to RQ6: Both `rr` and JMVX's struggle to record a purely IO bound benchmark, introducing **672.1%** and **192.5%** overhead respectively.

Answer to RQ7: Combining MVX with RR is an effective technique handle I/O bound workloads as JMVX's overhead dropped from **192.5%** to **25.5%**. In absolute terms, combining MVX with RR can result in faster recordings than using `rr` on similar native code: 128msvs 203ms.

Table 8 Hybrid MVX/RR compared with JMVX's recorder for Dacapo

	Total		Relaxed	
Benchmark	Recorder	Hybrid	Recorder	Hybrid
avrora	1.137x	1.117x	1.116x	1.135x
batik	1.071x	1.090x	1.067x	1.084x
fop	1.211x	1.152x	1.203x	1.148x
h2	1.438x	1.423x	1.049x	1.222x
jme	1.065x	1.067x	1.065x	1.068x
jython	1.651x	1.732x	1.422x	1.652x
luindex	1.233x	1.381x	1.203x	1.347x
lusearch	1.253x	1.254x	1.281x	1.270x
pmd	1.159x	1.150x	1.124x	1.140x
sunflow	0.924x	0.962x	0.898x	0.931x
xalan	1.171x	1.167x	1.252x	1.088x
AVG	1.210x	1.227x	1.153x	1.189x

We recorded the DaCapo suite using our hybrid MVX recorder to understand the performance impact on a more general workload. We test with both TO and RTO based concurrency mechanism. Table 8 shows the results.

Benchmarks in the DaCapo suite are CPU bound and perform little I/O, mostly buffered. Furthermore, most of the I/O in each benchmark (*e.g.,* uncompressing files for processing by the benchmark) happens *outside* of the timed harness. As a result, the potential performance improvement for the hybrid MVX recorder is low. Still, we can see that some benchmarks benefit from the hybrid MVX recorder: fop (code linter), sunflow (3D ray tracer), and xalan (XML file transformer). For the rest of the benchmarks, the hybrid MVX recorder stays within the same level of overhead as the regular recorder. Overall, using the hybrid MVX recorder does not hurt the performance of more common workloads.

Answer to RQ8: Hybrid MVX recording does not penalize common workloads that are CPU-bound. The performance is comparable to a regular recorder.

5.6 Threats to validity

The threats to the validity of our experiments are:

- Benchmarks represent common software workloads, but not all possible workloads, and therefore our results may not hold for all types of programs [2, 3].
- There are sources of randomness that we cannot control, such as the Just-In-Time compiler, Garbage Collector, and the scheduler of the underlying Operating System. Each can introduce noise in the runtime performance data at random points. We mitigated this threat by running each result 10 times and reporting the average.
- We collected all results using a single large NUMA machine, which may not be representative of performance on smaller machines.
- The hardware built-in thermal throttling logic may reduce the performance during experiments as the CPUs warm up. We mitigated this threat by fixing the frequency at its minimum for all experiments, disabling turbo boosting, and staggering experiments over time.

6 Related Work

Record/Replay systems

Octet [8, 9] is a RR system that captures cross-threads dependencies with a high level of granularity (single shared variables), which can be used to record [9] and replay [8] Java programs. Octet captures the **true** Partial Order in which each thread accesses each shared memory location, which allows Octet to elide synchronization operations during replay. For instance [8]: *"two critical sections that acquire the same lock but do not have a data dependence between them, can execute in parallel in the replayed execution"*. However, capturing such a fine-grained level of concurrency comes with the expected costs: 31% performance overhead when recording and 49% when replaying. Furthermore, Octet's RR only captures inter-thread dependencies (*i.e.,* no I/O, class loading, or any other sources of non-determinism) and requires a custom JVM. Our RTO design builds on JMVX, which captures all non-determinism on an unmodified JVM, and still captures enough inter-thread dependencies for accurate replay but with dramatically lower costs, even though our proposed technique cannot reproduce data-races.

Castor [21] is an RR system that relies on hardware timestamps to capture the (partial) order of events across threads with zero synchronization. When replaying, Castor needs to sort the log first to obtain a total order of events, and relies on transactional memory to do so efficiently. Our RTO approach does not require sorting or other log post-processing before replay, and does not require special hardware for high performance. Similarly to our approach, Castor uses two processes to record: the program being recorded, and a special *recording agent*. Castor's program writes **all** captured events to shared memory, which the recording agent then writes to the recording log. Our Hybrid RR+MVX approach is fundamentally different. The leader only captures multi-threading and networking events, and does not capture file I/O. The follower then can reconstructs the skipped events by executing the same program on the same system.

LEAP [15] uses a static analysis to over-approximate globally accessible variables that can be modified by many threads, and records the partially-ordered updates of such variables using Lamport clocks [18]. When replaying, LEAP requires a thread scheduler to post-process the log and ensure threads match the recorded partial order. Similarly to Octet, LEAP can replay data-races by capturing a finer granularity of events than our proposed RTO with the expected higher costs: 626% for recording the Avrora benchmark, and 74% for lusearch. Building on JMVX, our RTO approach assumes the program is data-race free, but our coarser RTO approach results in dramatic performance improvements when recording: 11% for Avrora and 18% for Lusearch. LEAP's paper does not report the performance overhead when replaying. **Respec** [19] is an MVX system with a leader and a single follower (dubbed *"online record-replay"* in the paper). Respec's *"recorder"* (*i.e.,* MVX leader) captures synchronization operations together with their PO, using a vector clock with one entry per synchronization object (*i.e.,* lock), which is not feasible for managed languages where typical programs use a large number of dynamically allocated locks (*e.g.,* one lock per bucket in a concurrent hash map). Respec's *"replayer"* (*i.e.,* MVX follower) attempts to replay the captured operations following the logged PO. If the replay fails (*e.g.,* due to a data-race), Respec rolls-back both recorder and replayer and re-executes one thread at a time, also including the kernel scheduling decisions in the log. As a consequence, Respec breaks the program execution into *"epochs"* that can be rolled-back. If Respec's recorder reaches the end of an epoch early, Respec blocks it until the replayer catches up, even for data-race free programs. As a result, Respec reports performance overheads in the range 18%–55% depending on the

number of threads. Our approach, building on JMVX, cannot capture data-races but does not require custom OS support, never stops the recorder to wait for the replayer, captures logs that can always be replayed successfully, and introduces a lower performance overhead that is not dependant on the number of threads. Respec can, optionally, save the log on disk for future offline replay, but the paper does not evaluate such replays or describe its implementation. **DoublePlay** [32] builds on Respec to replay captured epochs concurrently with many threads, potentially out-of-order. To replay each epoch, DoublePlay starts by restoring the checkpoint captured at the start of the epoch. Then, it replays the program using the log. If an epoch fails to replay, by diverging during execution or by generating a final state that is different from the checkpoint of the following epoch, DoublePlay terminates all following epochs, waits for all preceding epochs to finish replay, and then repeats the divergent epoch in isolation. DoublePlay reduces the overhead from the base technique Respec to 15%–28%, depending on the number of threads, which is still higher than our proposed technique.

Multi-Version eXecution

Multi-version execution (MVX) systems have been used for reliability [13,14], security [34, 35], to deploy incompatible dynamic analyses [23, 36], and to improve availability [24, 27]. `VARAN` [14] introduced the *leader/follower* architecture, in which a leader process interacts with the OS and forwards the results of nondeterministic operations (*e.g.,* file reads/writes and concurrency decisions) to one or more followers which replay the execution concurrently. Interestingly, `VARAN` sketches a combination of RR with MVX, that records using an unmodified leader and a special follower that writes the recording log, and replays using a special leader that streams the log to a special follower that replays the operations. Our Hybrid RR+MVX approach is fundamentally different, as we use an unmodified replayer and split the responsibility of recording nondeterminism between leader and follower. Furthermore, `VARAN` does not address how the two techniques can benefit one another when composed, and does not provide an implementation.

Most MVX systems use Total Ordering [14, 23, 34, 35]. Volkaert *et al.* [33] propose an MVX system that also supports Partial Ordering (PO). The authors report poor scalability of PO due to increased inter-thread communication required to decide whether each use of a shared synchronization resource (*e.g.,* lock) can be relaxed. Instead, they introduce a *"wall-of-clocks"* approach that uses one buffer per thread (which JMVX also uses), and requires inter-thread synchronization only when two threads attempt to use the same resource. Their approach requires a Lamport clock with one entry per synchronization resource, which in turn requires static identification of such resources. In contrast, we implemented our proposed RTO ordering by tracking only the last owner of each synchronization resource, which makes it feasible to deploy in managed languages where typical programs use a large number of dynamically allocated locks (*e.g.,* one lock per bucket in a concurrent hash map).

The synergy of MVX and RR was first proposed in Sinatra [27], which uses RR to record the JavaScript events fed to each open tab on a browser, together with MVX to spawn an updated browser in the background that replays the same events to obtain the same state on each tab. As a result, Sinatra can perform dynamic software updates on unmodified browsers without any noticeable user disruption. We propose yet another combination of MVX and RR to address the pathological high overhead when recording I/O-bound workloads.

Data-Race Freedom (DRF) assumption

All MVX systems cited assume DRF. Castor [21] and JMVX [28] also assume DRF, and `rr` [22] limits concurrency to a uniprocessor with a deterministic schedule which captures

data-races but reduces the observability window on data-races due to concurrent thread memory accesses.

Castor [21] argues that DRF is an adequate assumption: bugs that lead to system outages are more pressing and are less likely to be caused by data races. The paper analyzes bug reports reported in the Chromium project between 2012 and 2016. Of 65,861 bugs reported and fixed within the analyzed dates, only 165 were from data races. The authors note that the majority of those bugs were solved by using a dynamic analysis for data-races (ThreadSanitizer [30]), making RR unnecessary.

7 Conclusion

This paper argues that the biggest remaining barrier to deployable record/replay is the way determinism is enforced: (1) by imposing overly strict ordering on concurrent executions, and (2) by forcing the recorder to pay the cost of duplicated I/O. We address both issues with two techniques that preserve deterministic replay while moving overhead off the critical path.

Our first contribution, Relaxed Total Order (RTO), shows that RR does not need to enforce a single global total order to remain correct and useful. By computing a relaxed order that preserves only the inter-thread constraints that replay actually relies on, RTO reduces contention and waiting during both recording and replay. We implement RTO in JMVX and show that it reduces recording overhead on 11 benchmarks from the DaCapo test suite from **21.0%** to **15.3%** overhead, and cuts replay overhead from from **67.5%** to **31.7%**, thus demonstrating that much of the cost of RR comes from unnecessary serialization rather than inherent replay requirements.

Our second contribution revisits a long-standing limitation of RR on I/O-heavy workloads: conventional recorders effectively perform each I/O twice, once for the program and once for the log. We show that hybrid MVX/RR can hide this cost by shifting the duplicate I/O to a follower variant and backfilling I/O events into the recording as needed. On an I/O-dominated workload, our novel approach reduces recording overhead from **192.5%** to **25.5%**. In comparison, `rr` incurs a **672.1%** slowdown on the same workload. Importantly, hybrid MVX/RR does not penalize CPU-bound workloads, although it also provides limited benefit, making it practical for any workloads.

Overall, RTO and hybrid MVX/RR are complementary: RTO reduces synchronization-induced overhead across the board, while hybrid MVX/RR targets the workloads where RR has historically been least viable. Together, their combination moves deterministic replay closer to a practical foundation for *"always-on"* debugging and monitoring in production systems.

References

1 Jmvx github repository. `https://github.com/bitslab/jmvx`. Accessed: 2025-07-15.

2 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. On benchmarking for concurrent runtime verification. In *International Conference on Fundamental Approaches to Software Engineering*, pages 3–23. Springer, 2021.

3 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. Runtime Instrumentation for Reactive Components. *LIPIcs, Volume 313, ECOOP 2024*, 313:2:1–2:33, 2024.

4 Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. SaBRe: Load-time selective binary rewriting. *International Journal on Software Tools for Technology Transfer*, 24(2):205–223, April 2022.

5 Jonathan Bell and Luís Pina. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:31, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

6 Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 362–371, May 2013.

7 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, October 2006. Association for Computing Machinery.

8 Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 90–101, New York, NY, USA, September 2015. Association for Computing Machinery.

9 Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: Capturing and controlling cross-thread dependences efficiently. *SIGPLAN Not.*, 48(10):693–712, October 2013.

10 David Dice, Mark S. Moir, and William N. Scherer III. Quickly reacquirable locks, oct 2010. US Patent.

11 Andy Georges, Mark Christiaens, Michiel Ronsse, and Koenraad De Bosschere. JaRec: A portable record/replay environment for multi-threaded Java applications. *Software: practice and experience*, 34(6):523–547, 2004.

12 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, 2008.

13 Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 612–621, May 2013.

14 Petr Hosek and Cristian Cadar. VARAN the Unbelievable: An Efficient N-version Execution Framework. *ACM SIGARCH Computer Architecture News*, 43(1):339–353, March 2015.

15 Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 207–216, New York, NY, USA, 2010. ACM.

16 Jong-Deok Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 10 pp.–. IEEE, 2001.

17 Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):155–166, June 2010.

18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: The Works of Leslie Lamport*, pages 179–196. Association for Computing Machinery, New York, NY, USA, October 2019.

19 Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient online multiprocessor replayvia speculation and external determinism. *SIGPLAN notices*, 45(3):77–90, 2010.

20 Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 378–391, New York, NY, USA, 2005. Association for Computing Machinery.

21 Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. *Computer architecture news*, 45(1):693–708, 2017.

22 Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, 2017.

23 Luís Pina, Anastasios Andronidis, and Cristian Cadar. FreeDA: Deploying incompatible stock dynamic analyses in production via multi-version execution. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 1–10, New York, NY, USA, 2018. ACM.

24 Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 573–585, New York, NY, USA, April 2019. Association for Computing Machinery.

25 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for java on a stock JVM. In *Proceedings of the ACM 2014 International Conference on Object-Oriented Programming Languages, Systems, and Applications*, OOPSLA '14. ACM, October 2014.

26 Paul Rubin, David MacKenzie, and Stuart Kemp. Dd. https://man7.org/linux/man-pages/man1/dd.1.html.

27 Ugnius Rumsevicius, Siddhanth Venkateshwaran, Ellen Kidane, and Luís Pina. Sinatra: Stateful Instantaneous Updates for Commercial Browsers Through Multi-Version eXecution. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

28 David Schwartz, Ankith Kowshik, and Luís Pina. Jmvx: Fast Multi-threaded Multi-Version eXecution and Record-Replay for Managed Languages. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA 2024, Pasadena, California, United States, October 2024. Association for Computing Machinery.

29 David Schwartz and Luís Pina. JMVX Repository — RTO implementation branch. https://github.com/bitslab/jmvx/tree/rto.

30 David Schwartz and Luís Pina. Artifact for Optimizing Record/Replay through Relaxed Total Ordering and Multi-Version eXecution. `https://zenodo.org/records/19643217`, April 2026.

31 David Schwartz and Luís Pina. Artifact for Jmvx: Fast Multi-threaded Multi-Version eXecution and Record-Replay for Managed Languages. `https://doi.org/10.5281/zenodo.12637140`, July 2024.

32 Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

33 Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 65–74, New York, NY, USA, 2010. Association for Computing Machinery.

34 Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. *ACM transactions on computer systems*, 30(1):1–24, 2012.

35 Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming Parallelism in a Multi-Variant Execution Environment. In *Proceedings of the*

Twelfth European Conference on Computer Systems, EuroSys '17, pages 270–285, New York, NY, USA, April 2017. Association for Computing Machinery.

36 Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and Efficient Application Monitoring and Replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 167–179, Denver, CO, June 2016. USENIX Association.

37 Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. Ghumvee: Efficient, effective, and flexible replication. In Joaquin Garcia-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, Ali Miri, and Nadia Tawbi, editors, *Foundations and Practice of Security*, pages 261–277, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

38 Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing Security Mechanisms through Diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.