Efficient Transaction-Based Deterministic Replay for Multi-threaded Programs[†]

Ernest Pobee
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
ernestpob@gmail.com

Xiupei Mei
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
xpmei2-c@my.cityu.edu.hk

W.K. Chan[‡]
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Existing deterministic replay techniques propose strategies which attempt to reduce record log sizes and achieve successful replay. However, these techniques still generate large logs and achieve replay only under certain conditions. We propose a solution based on the division of the sequence of events of each thread into sequential blocks called transactions. Our insight is that there are usually few to no atomicity violations among transactions reported during a program execution. We present TPLAY, a novel deterministic replay technique which records thread access interleavings on shared memory locations at the transactional level. TPLAY also generates an artificial pair of interleavings when an atomicity violation is reported on a transaction. We present an experiment using the Splash2x extension of the PARSEC benchmark suite. Experimental results indicate that TPLAY experiences a 13-fold improvement in record log sizes and achieves a higher replay probability in comparison to existing work.**

***Keywords*-Concurrency, Deterministic Replay, Transactions, Multi-threading.**

I. Introduction

Deterministic replay of a program involves recording data from the program execution and subsequently scheduling the program to achieve some desired state or output [8], [11], [16], [23], [29]. Deterministic replay techniques are generally categorized into hardware-based [2], [20], [21], [26], [29] and software-based [11], [12], [16] techniques.

Software-based techniques often consist of a record phase and a replay phase. Some of them [12], [31] further include an offline phase to compute/construct feasible schedules for replay. Due to the non-ubiquitous nature of specialized hardware required for most hardware-based techniques, we present our technique in relation to software-based replay techniques.

In the record phase, replay techniques typically record three types of thread access orders known as *interleavings* ($\rightsquigarrow$) namely, *read-write*, *write-read* and *write-write* interleavings [17], [22], [30] on shared memory locations. Interleavings on synchronization primitives are also recorded. Some techniques [1] also record other types of data such as read values to ensure value determinism in the replay phase. Chen et al. [6] refer to the proportion of interleavings recorded as the "degree of record fidelity" and indicate a positive correlation between the degrees of recording fidelity and replay fidelity.

Towards the goal of achieving high record fidelity with minimal overhead, Huang et al. [11] propose LEAP which records data at a local level for shared memory locations. This strategy reduces the record runtime overhead by 10x compared to the prior local-order and global-order based techniques. Zhou et al. [31] propose Stride, a relaxed recording technique which reduces the need of synchronization whilst recording interleaving orders and achieves a 2.5x improvement in runtime overhead as well as a 3.8x reduction in record log sizes compared to LEAP. Liu et al. [17] also propose Light which achieves a 10x reduction in record log sizes compared to Stride. However due to its recording strategy, Light may fail to reproduce correct output for serialized write events to output devices involving multiple shared memory locations. Also, its replay phase enforces a global order on an execution which increases replay runtime overhead and reduces concurrency. AggrePlay [24] compresses some interleavings during the record phase using a read vector. It then assigns thread-local scheduling constraints during the replay phase improving replay concurrency. However, it suffers higher record overhead compared to Light.

A desirable attribute of a deterministic replay technique is to minimize the impact of high record fidelity whilst ensuring deterministic replay.

We propose our technique based on the following insight: During a program execution, several interleavings may be observed between events executed by different threads on shared memory locations. However, there are usually few to no atomicity violations (*which means a thread accesses a shared memory location in an atomic region at the same time as another thread*) reported during a program execution. Therefore, modeling the sequence of events of each thread as a sequence of transactions enables the application of transactional attributes and conditions to the events executed in the program.

In this paper, we present TPLAY, a novel deterministic replay technique which segregates the sequence of events of each thread into a sequence of transactions. During the record phase, TPLAY creates a new transaction on each thread's first

[†]This research is supported in part by the GRF of HKSAR Research Grants Council (project nos. 11214116 and 11200015), the HKSAR ITF (project no. ITS/378/18), the CityU MF_EXT (project no. 9678180), the CityU SRG (project nos. 7004882 and 7005216) and the CityU SGS Conference Grant.

[‡]Correspondence Author

write event or on any write event preceded by a read event.

Consider a scenario where two transactions tr_1 and tr_2 are created by threads t_1 and t_2 respectively. Suppose that, all interleavings between tr_1 and tr_2 are in the form $e \rightsquigarrow e'$, where $e \in tr_1$ and $e' \in tr_2$, may be reduced to a single interleaving recorded as $tr_1 \rightsquigarrow tr_2$. This reduces the record log whilst preserving record fidelity. During replay the two transactions tr_1 and tr_2 can be executed sequentially, preserving all interleavings between the two threads.

However to ensure deterministic interleaving reproduction at a transactional level, the atomicity of each transaction must be ascertained. We apply an existing transactional atomicity checker during the record phase. For transactions whose atomicity is violated, we propose a solution which we illustrate as follows:

Suppose $\langle e_1, e_4 \rangle \in tr_1$ and $\langle e_2, e_3 \rangle \in tr_2$ and the trace σ is $\langle e_1, e_2, e_3, e_4 \rangle$. TPLAY records $tr_1 \rightsquigarrow tr_2$ based on the interleaving $e_1 \rightsquigarrow e_2$. Then on the interleaving $e_3 \rightsquigarrow e_4$, an atomicity violation is reported on tr_1. TPLAY removes the transactional interleaving $tr_1 \rightsquigarrow tr_2$ and records the interleaving $tr_2 \rightsquigarrow e_4$. However, as at event e_4, no information about the interleaving $e_1 \rightsquigarrow e_2$ is known. To preserve any previous interleavings, the immediate-preceding event of e_4 in the same transaction i.e., e_1, is ordered before tr_2, creating the artificial interleaving $e_1 \rightsquigarrow tr_2$ which is recorded by TPLAY. This preserves all interleaving orders in the presence of an atomicity violation on any transaction.

During the replay phase, TPLAY replays a program using transactional level interleavings if no atomicity violation was reported during the record phase. Otherwise the generated interleavings are used in preserving the observed atomicity violations. For the trace σ in our example, TPLAY enforces the replay constraints $tr_2 \rightsquigarrow e_4$ and $e_1 \rightsquigarrow tr_2$ during replay due to the atomicity violation.

To evaluate the performance of TPLAY, we answer the following questions using our experimental results:

RQ1: Can TPLAY achieve smaller overheads compared to an existing technique in the record phase?

TPLAY's recording strategy results in an average of 25.44MB in record log size compared to an average of 333.20MB for Light [17].

RQ2: Does TPLAY achieve a small runtime overhead in the replay phase?

TPLAY's replay strategy does not enforce a global ordering of events on all threads. This results in a significant decrease in replay runtime overhead especially. Our results indicate that TPLAY's replay phase executes at 76% of the record phase runtime.

RQ3: Does TPLAY exhibit high replay fidelity?

TPLAY reproduces the program state of benchmarks with a probability of 96.6%.

Our main contributions are as follows

- We present a novel deterministic replay technique which models the event sequence of each thread as a sequence of transactions, then records interleavings between the transactions to reduce record log sizes.
- We present an algorithm which preserves the original event interleavings during a transactional atomicty violation by generating an artificial pair of interleavings.
- We show the feasibility of TPLAY by implementing it as a tool and evaluate TPLAY through an experiment.

The rest of the paper is organized as follows. Section II outlines preliminary information. Section III describes a motivating example used in presenting our technique. Section IV details our algorithm, while Section V contains evaluation and experimental results. Section VI discusses the related work. Finally, Section VII concludes the paper.

II. Preliminaries

This section details the preliminary information used in this paper.

TABLE I
Preliminary Information

Operation	$op \coloneqq w(x) \mid r(x) \mid acq(m) \mid rel(m) \mid fork(u) \mid join(u)$ $x \in$ Memory Location; $m \in$ Lock; $u \in$ Thread;
Event	$e \coloneqq \langle t, op \rangle$, $t \in$ Thread; $op \in$ Operation
Execution trace	$\sigma \coloneqq \langle e_1, e_2, e_3, \ldots, e_n \rangle$, $e_i \in$ Event

A. Execution Trace

An execution trace $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ is a sequence of events observed from the execution of a program. An event ***e*** represents one of the following:

- *t.r(x)*: A read instruction executed by thread *t* on memory location *x*.
- *t.w(x)*: A write instruction executed by thread *t* on memory location *x*.
- *t.acq(m)*: A lock acquisition instruction executed by thread *t* on lock *m*.
- *t.rel(m)*: A lock release instruction by thread *t* on lock *m*.
- *t.fork(u)*: Thread *t* forks another thread *u*.
- *t.join(u)*: Thread *t* joins another thread *u*.

Other synchronization primitives such as *wait*, *signal*, and *barrier* are also considered by our algorithm and follow procedures similar to the synchronization primitives above, as show by Table I. We omit them for brevity.

B. Interleaving

An interleaving $\rightsquigarrow$ is defined as the order by which different threads access shared memory objects. Three types of interleavings may be observed in an execution:

1) *write-write*: two threads perform write events consecutively on some shared object.
2) *write-read*: a thread performs a write event followed by a read event by some other thread.
3) *read-write*: a thread performs a read event followed by a write event by some other thread.

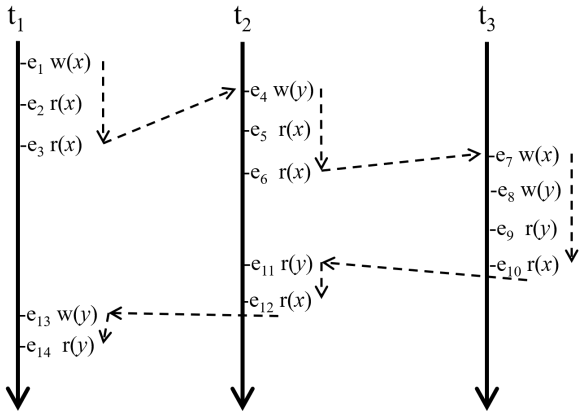

Fig. 1. Running example: An execution trace σ_1 of a multi-threaded program with three threads t_1 t_2, and t_3. (Dashed arrows represent the global trace σ_1)

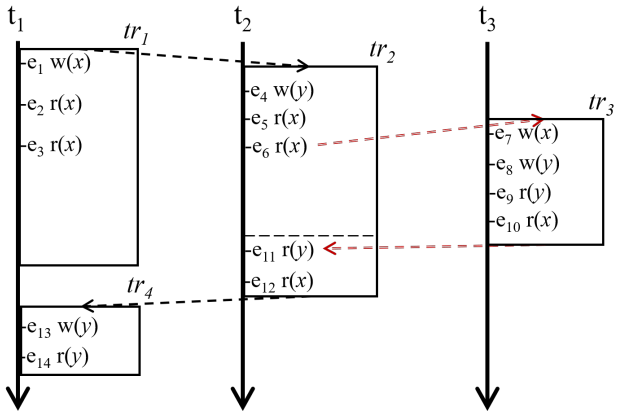

Fig. 2. Illustration of TPLAY record phase on σ_1. Dashed arrows represent transaction-level interleavings and double compound arrows represents event level interleavings.

C. Transactions

RegionTrack [28] defines transactions as "a sequence of events executed by a thread t in between of a matching pair of events *begin(t, l)* and *end(t, l)* as a (regular) transaction $tx = \langle begin(t,l), ..., e_x, ..., end(t,l)\rangle$", where *begin(t, l)* and *end(t, l)* denote the beginning and the ending of an atomic region l and $e_x \in tx$.

We adapt their idea to formulate our notion of a transaction and transactional atomicity violation (or atomicity violation for short) as follows.

Definition 1. (Transaction). A transaction is a sequence of events executed by a thread t represented by $tr = \langle begin(t,l), ..., e_x, ..., end(t,l)\rangle$ where *begin(t, l)* refers to an event e_i which satisfies the following three conditions:

1) e_i is a write event.
2) There is either an *immediate preceding read event* $e_{i\text{-}1}$ or no event $e_{i\text{-}1}$ exists.
3) e_i, $e_{i\text{-}1} \in \sigma_t$. (thread t's execution trace).

Also, *end(t, l)* is a either read event e_j in σ_t which immediately precedes a write event e_{j+1} in σ_t or the *last event* in σ_t.

Definition 2. A **transactional interleaving (TI)** $\rightsquigarrow_{tr}$ is such that if $e_i \rightsquigarrow e_j$ and $e_i \in tr_1$ and $e_j \in tr_2$, then $tr_1 \rightsquigarrow_{tr} tr_2$.

Definition 3. (Happens-Before Relation) [4], [10]: The happens-before (HB) relation $<_\alpha$ for a trace α is the smallest transitively-closed relation over the events in α such that the relation $a <_\alpha b$ holds whenever a occurs before b in and one of the following holds:

- **Program order:** The two operations are performed by the same thread.
- **Locking:** The two events acquire or release the same lock.
- **Fork-join:** One events is *t.fork(u)* or *t.join(u)* and the other events is by thread u.

For events a, b, $c \in \alpha$, If $a <_\alpha b$ and $b <_\alpha c$, then $a <_\alpha c$.

D. Transactional Atomicity Violation

Each thread maintains a vector clock (VC) [14] under RegionTrack and assigns a VC to each transaction created by the thread. RegionTrack captures HB-relations across threads by performing a join operation on the VCs of threads during an HB event. When an HB relation is captured, the VC of the current thread is updated via a join operation.

Given two events e_i and e_j where $e_i \rightarrowtail e_j$ and $e_j \in tx$, RegionTrack will report an atomicity violation on tx if the VC of *tx.begin* $\leq$ the VC of e_i.

E. Read Count Vector Clocks

A read count (RC) vector clock [24] (*a variation of Lamport's vector clock* [14]) is a tuple of values where each value which tracks the number of read events of the corresponding thread in an execution trace. An RC vector clock maintains a count of a thread's read events to a shared memory location in the form of $RC_t[t]$, where t represents the current thread.

III. Motivating Example

We present a running example to motivate our work. Fig. 1 shows an execution trace σ_1 of a multi-threaded program with three threads t_1, t_2 and t_3, 14 write and read events labeled as e_1 to e_{14}, and two shared memory locations x and y. In Fig. 1, t_1 executes a write then two reads e_2 and e_3 on location x. Thread t_2 then executes a write e_4 on y then two reads e_5 and e_6 on location x. Thread t_3 then executes a write e_7 on x, then a write e_8 on y, followed by two reads e_9 and e_{10} to y and x respectively. Thread t_2 executes two reads e_{11} and e_{12} to y and x respectively. Thread t_1 then executes a write on y then executes e_{14}, which is a read on y.

An existing technique Light [17] records inter-thread flow dependencies (write-read interleavings). For σ_1, Light records the set $\{e_1 \rightsquigarrow e_5, e_8 \rightsquigarrow e_{11}, e_7 \rightsquigarrow e_{12}\}$ and passes them as constraints to a constraint solver to generate a feasible trace.

To enforce intra-thread access orders, Light also encodes the constraints $e_1 \rightsquigarrow e_2$, $e_2 \rightsquigarrow e_3$, $e_3 \rightsquigarrow e_{13}$, $e_{13} \rightsquigarrow e_{14}$, $e_4 \rightsquigarrow e_5$, $e_5 \rightsquigarrow e_6$, $e_6 \rightsquigarrow e_{11}$, $e_{11} \rightsquigarrow e_{12}$, $e_7 \rightsquigarrow e_8$, $e_8 \rightsquigarrow e_9$, $e_9 \rightsquigarrow e_{10}$ as constraints in the constraint solver. As such, Light records a total 3 inter-thread interleavings and 11 intra-thread interleavings for σ_1.

One main drawback of Light is that it may fail to reproduce a serialized sequence of writes to an external device involving multiple shared memory locations. As an example, the interleaving $e_4 \rightsquigarrow e_8$ may be $e_8 \rightsquigarrow e_4$ in some trace σ_1'. Light is also limited by the capacity of constraint solvers to generate traces.

TPLAY reproduces the program state by enforcing *write-write*, *read-write*, and *write-read* interleavings in σ_1 at a transactional level. We discuss TPLAY in detail in the next section.

IV. TPLAY Algorithm

In this section, we present TPLAY. The algorithms are implemented as callback functions which are triggered on the execution of specific events. The following notations are used in algorithms 1 and 2:

- $\perp$: represents a null instruction.
- *{}*: represents an empty set.
- $dSet_t$: The set of interleavings for thread *t*.
- $rwSet_t$: The set of read-write interleavings for thread *t*.
- $aSet_t$: The set of artificial interleavings for thread *t*.
- $bSet_t$: The set of event interleavings for thread *t*.
- $trID_t$: Current transaction ID for thread *t*.
- lw_x: Last write operation to a shared memory location *x*.
- Ψ_t: Last instruction type for thread *t*.
- L_m: Last lock operation on lock *m*.
- *Tid(e)*: Executing thread of operation *e*.
- *Lock(e)*: Lock object acquired in operation *e*.
- *var(e)*: Shared object being accessed by operation *e*.
- RC_t: The RC vector clock for thread *t*.
- *execute(e)*: The operation for event *e* is executed.
- *t.yield()*: The executing thread *t* waits for other threads to advance without blocking.
- *pop()*: removes data from the first index position of a data structure.
- *first()*: represents data in the first index position of a data structure.
- *sinkThread*: the thread which performs the second event in an interleaving.
- *sourceThread*: the thread which performs the first event in an interleaving.

A. Record Phase

During the record phase, a modified version of RegionTrack (RT_m) is used to create transactions and track HB relations across threads. When an interleaving is detected, TPLAY retrieves the transactional information of the interleaving events from RT_m and records a transactional interleaving (TI) in *dSet*. When an atomicity violation over two events is reported by RT_m, TPLAY removes any recorded TI associated with the two events, then generates two event access orders which are stored in two different sets *aSet* and *bSet*.

The TPLAY record phase is presented in Algorithm 1. Lines 1-2 initialize the RC vector clock of thread *t* to 0, $rwSet_t$, $aSet_t$, $bSet_t$ and $dSet_t$ to empty, $trID_t$ to 0 and Ψ to $\perp$ for each thread. During a write access event, the transaction count of thread *t* is incremented by 1 if no previous instruction exists or there is a read for *t* (line 4). The write operation is executed at line 5. If the last write to the shared memory location was performed by some other thread *t'*, the last write (made up of *t'* and the $trID_{t'}$) is ordered before the current write event and appended to $dSet_t$ (lines 6 -7). This also eliminates thread-local orderings which are inherently deterministic. Then the last write for the shared memory location is updated using *t* and $trID_t$ at line *8*. Line 9 records any read-write interleavings in which the write event is involved and updates the last instruction type for thread *t*.

For each read access event, the index for the current thread in its RC vector clock is incremented by 1 and the read operation is executed at line 12. If the last write to the shared memory location was performed by some other thread *t'*, the last write is ordered before the current read event and appended to $dSet_t$ (lines 13 -14). Line 15 updates the last instruction type for thread *t*.

On lock acquisition, the lock operation is executed at line 18. If the last lock access to some lock object *m* was performed by some other thread *t'*, L_m is ordered before the current synchronization event and appended to $dSet_t$ (lines 19 -20). The last lock access to the shared object is updated at line 21.

The function *recordRW* (lines 23-26) is invoked at line 9 in the *onWrite* function. This function retrieves the read count values for every other thread in the set of Threads (line 24). The updated RC vector clock for thread *t* and the current write event are appended as a triple to $rwSet_t$ (line 25).

Lines 27-33 detail the action taken when an atomicity violation is detected by RT_m. If the executing thread's id and current transaction id exist in the *dSet* of the last thread *t* to access the shared object refrenced by *e*, the record is removed (lines 29-30). Then at line 31, $\langle Tid(e), e', t, tr\rangle$ is recorded in $aSet_t$ with *e'* being the preceding event of *e* in $\sigma_{Tid(e)}$. Finally $\langle t, tr, Tid(e), trID_{Tid(e)}\rangle$ is appended to $bSet_{Tid(e)}$. Algorithm 1 produces a record log which consists of *aSet*, *rwSet*, *bSet* and *dSet*.

Fig. 2 illustrates the TPLAY record scheme for σ_1. On event e_1, a transaction tr_1 is created by t_1. lw_x is also updated with the pair $\langle t_1, tr_1\rangle$. Events e_2 and e_3 each increments $RC_{t_1}[t_1]$ by 1. On e_4, a transaction tr_2 is created by t_2. lw_y is also updated with the pair $\langle t_2, tr_2\rangle$. RC_{t_2} is updated using RC_{t_1} and RC_{t_3} and $\langle RC_{t_2}, t_2, tr_2\rangle$ is appended to $rwSet_{t_2}$. Events e_5 and e_6 each increments $RC_{t_2}[t_2]$ by 1. For e_5, lw_x is retrieved and the interleaving $\langle t_1, tr_1, t_2, tr_2\rangle$ is appended to $dSet_{t_2}$. On e_6, $\langle t_1, tr_1, t_2, tr_2\rangle$ is not recorded since it already exists in $dSet_{t_2}$. On event e_7, a transaction tr_3 is created by t_3. lw_x is also updated with the pair $\langle t_3, tr_3\rangle$. RC_{t_3} is updated using RC_{t_1} and RC_{t_2}. The triple $\langle RC_{t_3}, t_3, tr_3\rangle$ is recorded in $rwSet_{t_3}$. On e_8, lw_y is retrieved and the interleaving $\langle t_2, tr_2, t_3, tr_3\rangle$

is appended to $dSet_{t_3}$. lw_y is then updated with the pair $\langle t_3, tr_3\rangle$. However, $\langle RC_{t_3}, t_3, tr_3\rangle$ is not recorded on e_8 since the values in RC_{t_3} have not changed. Events e_9 and e_{10} each increments $\text{RC}_{t_3}[t_3]$ by 1. An atomicity violation is reported on e_{11} for tr_2. Since an interleaving with the pair $\langle t_2, tr_2\rangle$ exists in $dSet_{t_3}$, the interleaving is removed and an artificial interleaving $\langle t_2, e_6, t_3, tr_3\rangle$ is then appended to $aSet_{t_3}$. This ensures that tr_2 will be paused until e_6 has been executed to reproduce the atomicity violation on tr_2. Then the interleaving $\langle t_3, tr_3, t_2, e_{11}\rangle$ is appended to $bSet_{t_2}$. On event e_{13}, t_1 creates transaction tr_4. lw_x is also updated with the pair $\langle t_1, tr_4\rangle$. RC_{t_1} is updated using RC_{t_2} and RC_{t_3}. The triple $\langle RC_{t_1}, t_1, tr_4\rangle$ is recorded in $rwSet_{t_1}$. Event e_{14} finally increments $\text{RC}_{t_1}[t_1]$ by 1.

TPLAY produces $dSet_{\sigma_1}$= $\{\langle t_1, tr_1, t_2, tr_2\rangle\}$, $bSet_{\sigma_1}$= $\{\langle t_3, tr_3, t_2, e_{11}\rangle\}$, $aSet_{\sigma_1}$= $\{\langle t_2, e_6, t_3, tr_3\rangle\}$, $rwSet_{\sigma_1}$= $\{\langle [2,0,0]_{t_2}, t_2, tr_2\rangle, \langle [2,2,0]_{t_3}, t_3, tr_3\rangle, \langle [2,4,2]_{t_1}, t_1, tr_4\rangle\}$ as the record log for trace σ_1.

B. Replay Phase

During the replay phase, RT_m is used to create transactions only. TPLAY divides the record log into constraint sets based on the *sinkThread* in each interleaving. Each thread is assigned a set in the form {$aSet_t$,$rwSet_t$, $bSet_t$ and $dSet_t$}. The constraint set is treated as a stack where only the first record in each data structure is used in evaluating the current event. When a record is successfully used in evaluating an event, the record is popped out of the stack.

The TPLAY replay phase is shown in Algorithm 2. The record log (*aSet, rwSet*, *bSet* and *dSet*) is used as input to replay callback functions. Lines 1-3 initialize RC vector clock to 0, $trID_t$ to 0 and Ψ to $\perp$ for each thread respectively.

For each write event, the *checkEvent* and *checkAv* functions are called on line 3 to ensure that the current event is not involved in some atomicity violation. The current operation is aborted if read-write conditions are not satisfied (line 4).

If the thread's $dSet_{Tid(e)}$ is not empty, and the current transaction id of the thread is lower than the sink transaction id of the topmost record in $dSet_{Tid(e)}$, the write event is executed (lines 5 - 7). This means each sinkThread can execute up until its first transaction involved in an interleaving. Also, If the thread's $dSet_{Tid(e)}$ is not empty, and the current transaction id of the sourceThread is greater than or equal to the source transaction id of the topmost record in $dSet_{Tid(e)}$, this means the interleavings has been fulfilled. The topmost record is popped out of $dSet_{Tid(e)}$ and the write event is executed. (lines 8-9). When neither of the two previous conditions are met at lines 7 and 8, the thread waits without blocking other threads at line 10. At Lines 11 and 12 the transaction id for the thread is incremented if the last event for the thread is either a read event or a null event. The write event is executed and the last instruction for the thread is updated at line 13.

On handling read events, on line 16, the two functions *checkEvent* and *checkAv* are called to ensure that the current event is not involved in some atomicity violation. Lines 17-18 update the RC vector clock of *Tid(e)*, execute the read event and update the last instruction of the thread with a read instruction.

Lock access events follow a similar replay strategy to read and write events. If the current transaction id for *Tid(e)* is less than the topmost transaction id in $dSet_{Tid(e)}$, the lock operation is executed (lines 21-23). Otherwise if the sourceThread involved in the dependency has a $trID_{t'}$ greater than or equal to the transaction id from $dSet_{Tid(e)}$, the lock operation is executed (line 24-25) and the topmost record in $dSet_{Tid(e)}$ is removed since the interleaving constraint has been satisfied at line 24. The thread waits without blocking other threads at line 26 if the conditions on lines 22 and 24 are not satisfied.

The *checkRW* (lines 28-33) function is invoked at line 4 in the *onWrite* function. This function retrieves the read count values for every other thread in the thread set *Thread* and updates the RC vector clock of *t* (line 29). Line 31 iterates

Algorithm 1 TPLAY record algorithm

1) $\forall t \in$ Thread **do** RC_t=0; Ψ_t=$\perp$; trID_t=0; dSet_t={};
2) rwSet$_\text{t}$={}; aSet$_\text{t}$={}; bSet$_\text{t}$={};
3) **onWrite** (*Event e*) **do**
4) **if** ($\Psi_{Tid(e)} \in$ {$\perp$,read}){ $trID_t$++; }
5) execute(e);
6) **if** (Tid($\text{lw}_{\text{var}(e)}$) $\neq$ Tid(*e*)){
7) dSet_t ^= $\langle Tid(lw_e), lw_e, Tid(e), trID_{Tid(e)}\rangle$; }
8) lw_e = $\langle Tid(e), trID_{Tid(e)}\rangle$;
9) **recordRW**(Tid(*e*), *e*); $\Psi_{Tid(e)}$= write;
10) **onWrite**
11) **onRead**(*Event e*) **do**
12) $RC_{Tid(e)}$ *[t]* ++; execute(*e*);
13) **if** (Tid(lw_e) $\neq$ Tid(*e*)){
14) dSet_t ^= $\langle Tid(lw_e), lw_e, Tid(e), trID_{Tid(e)}\rangle$; }
15) $\Psi_{Tid(e)}$= write;
16) **end onRead**
17) **onLockAcquire**(*Event e*)**:**
18) execute(*e*); m = var(*e*);
19) **if** (Tid(L_m) $\neq$ Tid(*e*)){
20) dSet_t ^= $\langle Tid(L_m), L_m, Tid(e), trID_{Tid(e)}\rangle$; }
21) L_m ^= $\langle Tid(e), trID_{Tid(e)}\rangle$;
22) **End onLockAcquire**
23) **recordRW**(*Thread* t, *Event e*) **do**
24) $\forall t'$ $\in$ Thread **do** RC_t[t'] = $\text{RC}_\text{t'}$ [t'];
25) rwSet_t ^= $\langle RC_t, Tid(e), trID_{Tid(e)}\rangle$;
26) **end recordRW**
27) **onAtomicityViolation**(*Event e*)**:**
28) $\forall\theta \in (L_{var(e)}, lw_{var(e)})$\{ t =Tid(θ); tr = θ.trID; }
29) **if**($\langle Tid(e), trID_{Tid(e)}\rangle \in dSet_t$){
30) dSet_t = dSet_t /$\langle Tid(e), trID_{Tid(e)}, t, tr\rangle$);
31) aSet_t ^= $\langle Tid(e), (e'), t, tr\rangle$; }
32) $\text{bSet}_{\text{Tid}(e)}$ ^= $\langle t, tr, Tid(e), trID_{Tid(e)}\rangle$;
33) **end onAtomicityViolation**

Algorithm 2 The TPLAY replay algorithm

1) $\forall t \in$ Thread **do** RC_t=0; Ψ_t=$\perp$; $trID_t$=0;
2) **onWrite** (*Event e*) **do**
3) **checkEvent(); checkAv();**
4) **if (!checkRW**(Tid(*e*), *e*))**{** Tid(e).yield(); **}**
5) **if (***$dSet_{Tid(e)}$* contains records)**{**
6) rec = *$dSet_{Tid(e)}$*.first();
7) **if (***$trID_{Tid(e)}$* < rec.*$trID_{Tid(e)}$*)**{** *skip to line 11* **}**
8) **else if (***$trID_{t'}$* >= rec.*$trID_{t'}$*)**{**
9) *$dSet_{Tid(e)}$*.pop(); *skip to line 11* **}**
10) **else {** Tid(e).yield(); **} }**
11) **if (**$\Psi_{Tid(e)} \in \{\perp$, read$\}$)**{**
12) *$trID_{Tid(e)}$*++; **}**
13) execute(e); $\Psi_{Tid(e)}$= write;
14) **end onWrite**
15) **onRead**(*Event e*) **do**
16) **checkEvent(); checkAv();**
17) *$RC_{Tid(e)}$ [Tid(e)] ++;* execute(e);
18) $\Psi_{Tid(e)}$= read;
19) **end onRead**
20) **onLockAcquire(***Event e***):**
21) rec = *$dSet_{Tid(e)}$*.first();
22) **if (***$trID_{Tid(e)}$* < rec.*$trID_{Tid(e)}$*)**{**
23) execute(e); **}**
24) **else if (***$trID_{t'}$* >= rec.*$trID_{t'}$*)**{**
25) execute(e); *$dSet_{Tid(e)}$*.pop(); **}**
26) **else {** *Tid(e).yield();* **}**
27) **End onLockAcquire**
28) **checkRW**(*Thread* t, *Event e*) **do**
29) $\forall t'$ $\in$ Thread **do** RC_t[t'] = $RC_{t'}$ [t'];
30) $\forall t'$ $\in$ Thread **do**
31) **if (***$RC_t[t']$* < *$rwSet_t$*[0].*$RC_t[t']$*)**{** return false; **}**
32) *$rwSet_t$.pop()*; return true;
33) **end checkRW**
34) **checkEvent**(*Event e*) **do**
35) rec = *$bSet_{Tid(e)}$*.first();
36) **if (***$e_{Tid(e)}$* < rec.*e*)**{**
37) execute(e); **}**
38) **else if (***$trID_{t'}$* > rec.*$trID_{t'}$*)**{**
39) execute(e); *$bSet_{Tid(e)}$*.pop(); **}**
40) **else {** *Tid(e).yield();* **}**
41) **end checkEvent**
42) **checkAv**(*Event e*) **do**
43) rec = *$bSet_{Tid(e)}$*.first();
44) **if (***$trID_{Tid(e)}$* < rec.*trID*)**{**
45) execute(e); **}**
46) **else if (***$e_{t'}$* > rec.*e*)**{**
47) execute(e); *$aSet_{Tid(e)}$*.pop(); **}**
48) **else {** *Tid(e).yield();* **}**
49) **end checkEvent**

over RC_t and returns false if each thread's current read count value is less than the recorded value in $rwSet_t$. Or else the interleaving is satisfied and removed from $rwSet_t$ at line 32.

The *checkEvent* function ensures that the sinkThread blocks prior to an event which is involved in an atomicity violation. Lines 36-37 ensure thread execution until the the event in the topmost record in $bSet_{Tid(e)}$ has been executed.

Alternatively line 38 checks if the sourceThread's transaction id is greater than the recorded transaction id in $bSet_{Tid(e)}$. If the condition is satisfied, the event is executed and the interleaving is removed from $bSet_{Tid(e)}$ at line 39. When both conditions are not met, the thread identifying by *Tid(e)* waits without blocking other threads at line 40.

The *checkAv* function ensures the artificial interleaving is enforced by the source thread. Lines 44-45 ensure thread execution until the event in the topmost record in $aSet_{Tid(e)}$. Otherwise line 46 checks if the source thread's current event is greater than the recorded event. If the condition on line 46 is satisfied, the event is executed and the interleaving is removed from $bSet_{Tid(e)}$ at line 47. Otherwise *Tid(e)* waits without blocking other threads at line 48.

The trace $\sigma_1$is replayed as follows: During the replay phase, t_1 creates tr_1 with event e_1 in the abscence of any interleaving constraint involving tr_1. Events e_2 and e_3 are executed and each event increments $RC_{t_1}[t_1]$ by 1. Then t_3 attempts to create tr_3 but fails since there are two constraints on tr_3 ($\{\langle t_2, e_6, t_3, tr_3\rangle\}$ and $\{\langle [2,2,0]_{t_3}, t_3, tr_3\rangle\}$ not yet satisfied. t_1 then attempts to create tr_4 and t_1 fails due to the interleaving $\{\langle [2,4,2]_{t_1}, t_1, tr_4\rangle\}$ not yet satisfied. t_2 then proceeds to create tr_2 since the constraints $\{\langle t_1, tr_1, t_2, tr_2\rangle\}$ and $\{\langle [2,0,0]_{t_2}, t_2, tr_2\rangle\}$ are satisfied. t_2 executes e_5 and e_6 and updates $RC_{t_2}[t_2]$. t_2 pauses at e_{11} due to the interleaving $\{\langle t_2, e_6, t_3, tr_3\rangle\}$ not yet satisfied. t_3 proceeds to create tr_3 and executes events e_7 and e_8. Events e_9 and e_{10} each increments $RC_{t_3}[t_3]$ by 1. t_2 then executes e_{11} and e_{12}, reproducing the atomicity violation on tr_2. Finally, t_1 creates tr_4 and executes e_{13} and e_{14}.

C. Thread abstraction & matching

Algorithms 3 and 4 present the thread abstraction process in the record phase and subsequent matching process in the replay phase using the following notations:

- $osTid_t$: System-assigned value for thread t.
- $opsTid_t$: System-assigned value for parent thread of thread t.
- $pTid_t$: The ID of the parent thread of thread t.
- Ω: Data Structure mapping $opsTid_t$ to $osTid_t$.
- *tLog*: Set of thread abstractions.
- Γ: Data Structure mapping each replay thread to its current number of children threads.
- Φ: Data Structure mapping replay threads to threads from *tLog*.

At line 1 of algorithm 3, Ω and *tLog* are initialized as empty structures. On thread creation, a value pair made up of the system-assigned value and the thread id is appended to Ω (line 3). For all threads with the exception of the main thread, a

value pair made up of the thread id and its parent thread id is appended to the set of thread abstractions *tLog* at line 4.

Algorithm 3 TPLAY thread abstraction algorithm

1) tLog={}; Ω={};
2) **onThreadCreate** (*Thread t*) **do**
3) Ω ^= $\langle osTid_t, Tid_t \rangle$;
4) **if** (*Tid*$_t \neq 0$)**{** tLog ^= $\langle Tid_t, \Omega[opsTid_t] \rangle$; **}**
5) **end onThreadCreate**

For Fig. 1, the resulting thread abstraction set is $\{\langle 1,0 \rangle, \langle 2,0 \rangle, \langle 3,0 \rangle\}$.

Algorithm 4 TPLAY thread matching algorithm

1) Ω={}; Γ={};Φ={}; tLog;
2) **onThreadCreate** (*Thread t*) **do**
3) Ω ^= $\langle osTid_t, t \rangle$;
4) **if** (Tid$_t$ == *0*)**{** Φ ^= $\langle 0,0 \rangle$; **}**
5) **else {** Γ[Ω[opsTid$_t$]]++;
6) freq = 1; parent = Φ[Ω[opsTid$_t$]];
7) childFreq = Γ[Ω[opsTid$_t$]];
8) $\forall \langle a,b \rangle \in$ tLog **{**
9) **if** (b == parent && freq == childFreq)**{**
10) Φ ^= $\langle t,a \rangle$; break; **}**
11) **elseif** (b == parent && freq $\neq$ replayFreq)**{**
12) freq++ ; **} } }**
13) **end onThreadCreate**

Algorithm 4 presents the thread matching algorithm in the replay phase. *tLog* is used as an input to the thread matching algorithm. Data structures Ω, Γ and Φ are all initialized as empty sets at line 1. On thread creation, the system-assigned value and the thread id pair is appended to Ω at line 3. If the thread is the main thread, the pair $\langle 0,0 \rangle$ is appended to Φ (line 4). The main thread in each execution is always assigned with id 0.

Otherwise, the number of children threads for the current thread's parent is incremented by 1 at line 5. Then we iterate over each thread abstraction in *tLog* (lines 8-12). If the second value (*b*) of a thread abstraction matches a parent thread and the parent thread's freq value matches the value of *childFreq*, the current thread id and the first value (a) of the matching abstraction is appended to Φ (lines 9-10). Otherwise, the *freq* value is incremented by 1 at line 12 when the thread abstraction is parent thread is matched but the child frequency is not.

V. Evaluation

A. Execution Environment

Our hardware setup consisted of a Dell PowerEdge R930 running the Dell Customized Image ESXi 6.0.0 Update 2 A01. Our experiments were conducted on a 64 bit virtual machine running the guest OS Ubuntu 18.04 Linux with 8 Intel Xeon(R) CPU E7- 4850 v3 @ 2.20Ghz processors, and 16GB of RAM.

We have implemented TPLAY and Light using Intel PIN version 3.0-76991 [19]. To be specific, for each tool, we implemented two separate pintools, for the record and replay phases respectively. In the case of Light, we followed the implementation in the paper using a solver for the Integer Difference logic theory in *z3* [9]. The record implementation of Light is publicly available without the replay phase and constraint solver and can only handle Java applications. Our benchmarks were run on C/C++ programs with the *pthread* standard. This made direct comparison difficult. A precaution taken was to test the correctness of our implementation of Light on a small benchmark we developed prior to our experiments. We also used code inspection on our implementations.

TABLE II
Execution metadata for benchmarks used in our experiment.

Benchmarks	# of Events		
	Read	Write	Lock
cholesky	13,539,277	3,576,547	1,529
fft	23,738,757	14,934,083	34
lu_cb	95,376,483	45,684,566	274
lu_ncb	94,188,973	45,662,755	274
ocean_cp	95,195,697	18,453,146	4,434
ocean_ncp	95,164,212	18,453,231	4,427
raytrace	319,947,352	69,805,540	239,444
radiosity	3,403,130	2,061,336	76
radix	237,630,987	78,661,575	213,960
volrend	35,082,358	10,642,543	7,133
water_nsquared	117,250,198	46,340,899	6,294
water_spatial	104,690,984	41,625,532	159
barnes	2,784,956,319	1,680,710,156	275,331
mysql	1,560,872	542,635	6

B. Benchmarks

We evaluated our implementation using benchmarks from the Splash2x extension of the PARSEC 3.1 benchmark suite [3], specifically *barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, water_nsquared*, *water_spatial* as well as kernel applications *cholesky, fft, lu_cb, lu_ncb, radix*, and *mysql*. We selected the Splash2x extension of PARSEC for its focus on concurrent computation on parallel machines. Table II details the number of read, write and lock acquisition events in each program under the experiment configuration.

C. Methodology

We ran each benchmark in the native configuration to establish native execution runtime. We reported this as the base time in our experiment results.

Each Splash2x benchmark program was configured with 4 worker threads with the *gcc-pthreads* configuration option and the *test-input* workload. This configuration provided each program adequate concurrency and input. The input for mysql was MySQL (Bug #85413) .

1) Record and Replay setup: For our record phase, each thread kept a local instruction counter and incremented it by 1 for each instruction executed. Each thread also kept a vector

TABLE III
EXPERIMENTAL RESULTS ON RECORD PHASE OF TPLAY AND LIGHT.

Benchmark	Application Domain	Log Size (MB)		Base time(s)	Normalized Slowdown
		TPLAY	*Light*		Light/TPLAY
cholesky	HPC	10.5	39.90	0.02	0.19
fft	Signal Processing	12.20	32.70	0.05	0.19
lu_cb	HPC	8.30	503	0.09	0.64
lu_ncb	HPC	8.30	502.50	0.125	0.72
ocean_cp	HPC	17.50	44	0.188	0.47
ocean_ncp	HPC	27.70	43.90	0.19	0.77
raytrace	Graphics	50.90	765	0.0574	0.03
radiosity	Graphics	7.50	13.10	0.092	0.05
radix	General	59.30	720	0.554	0.01
volrend	Graphics	12	152.30	0.086	0.06
water_nsquared	HPC	10.10	561.10	0.17	0.13
water_spatial	HPC	5.20	396.90	0.16	0.11
barnes	HPC	126	890	1.044	1.97
mysql	Database application	0.60	0.40	0.8548	0.75
	Mean	**25.44**	**333.20**	**0.26**	**0.44**

TABLE IV
RECORD PHASE DATA FOR TPLAY AND LIGHT. (TI REFERS TO TRANSACTIONAL INTERLEAVINGS. AV REFERS TO ATOMICITY VIOLATIONS. THE VALUES MARKED WITH * REPRESENT THE INTERLEAVING SET SIZES NOT SOLVED BY Z3).

Benchmark	# of Transactions	# of TI	# AV	# of Light W-R interleavings
cholesky	9,632	761	-	2,009,143*
fft	11,355,241	639,294	-	1,672,548*
lu_cb	46,392,503	399,169	-	23,785,125*
lu_ncb	46,117,695	397,970	-	23,757,388*
ocean_cp	18,536,819	239,971	-	2,360,277*
ocean_ncp	17,545,516	287,698	-	2,351,007*
raytrace	17,108,800	1,203,113	2	73,440,436*
radiosity	2,057,238	413,560	-	777,345*
radix	21,165,176	2,177,458	-	77,013,873*
volrend	5,559,482	421,344	98	7,711,556*
water_nsquared	42,878,818	254,304	-	28,316,969*
water_spatial	39,523,924	217,046	-	20,725,403*
barnes	69,510,023	8,510,541	-	411,164,199*
mysql	49,821	9,581	-	352
Mean	**24,129,334**	**1,083,700**	**50**	**48,220,401**

clock to track reads by other threads as well as a transaction counter which was incremented for every transaction created by the thread. For all shared memory locations, we maintained a data structure to store data on the last access to each shared memory location. We also kept a global map of each memory address with its associated data structure which is write protected by a single lock during initial creation and storage of the data structure. However, for subsequent access to each memory address index in the map, we maintained a set of 2^{10} locks which were acquired via a hash function (similar to Light).

For read events, we configured each thread to keep track of its read accesses to shared memory locations and made this data structure accessible to other threads. We protected access to this data structure by assigning each thread its own lock. Throughout the record phase, TPLAY was configured to keep all recorded data in memory until the program exited or was terminated.

Apart from read-write interleavings, we stored each interleaving (regardless of interleaving type) in the format $\langle a, b, c, d\rangle$ where a, b, c and d represented the sourceThread id, the transaction/event id executed by the sourcThread, sinkThread id and transaction/event id executed by the sinkThread respectively. A thread id was typically a 32-bit integer whereas an event/transaction id was a 64-bit unsigned integer. Read-write interleavings were stored in the format $\langle RC, a, b\rangle$ where RC represented a vector of read aggregate values from all threads and a and b represent the sink thread id and sink transaction/event id respectively.

We recorded *time spent* (total processor clock ticks for each run) using the clock function[1]. The time spent for each tool is calculated by recording the value returned by the clock function at the beginning and at the end of each run. The normalized slowdown difference between TPLAY and Light was calculated using the following formula (*time spent for Light / time spent for TPLAY*) and is shown in column 6 of Table III.

During the replay phase, each thread created during the replay phase was matched to a record phase thread using our thread matching algorithm i.e. Algorithm 4, and each thread only kept a record of interleavings within which it served as a sinkThread. During replay, we maintained an

[1]man7.org/linux/man-pages/man3/clock.3.html

instruction counter, transaction counter for each thread. To ensure optimal concurrency, each thread was configured to evaluate the interleaving constraint without synchronization. This meant for every interleaving $\langle a, b, c, d\rangle$, the sinkThread *c* only had to check if the sourceThread *a*'s current transaction id was greater than the recorded transaction id *b*. This ensured that the sourceThread was not blocked until the interleaving was confirmed by the sinkThread. The artificial interleavings created on atomicity violations were evaluated in the same manner.

Each interleaving was removed from the data structures once it was satisfied. TPLAY relinquished control of the program execution to the system scheduler when all interleavings were satisfied.

We recorded each interleaving for Light in the format $\langle a, b, c, d\rangle$ where *a*, *b*, *c* and *d* represent the source thread id, the source event id, sink thread id and sink event id respectively. Our implementation of TPLAY in the Pin framework is available online[2].

D. Constraint solving Approach

To benchmark the Light replay technique, the Z3 constraint solver was used with *write-read* interleavings from the Light record phase as inputs. We did this in accordance with the steps stated in [17]. A constraint solver accepts as input a set of statements which are encoded as constraints. In our experiments, we implemented a constraint solver using the Z3Py library (version 4.8.4) of Z3 [9] theorem solver for python programming. We present an example of the constraint solving approach as follows:

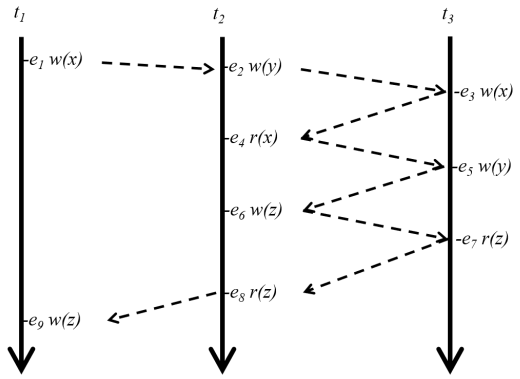

Fig. 3. Running example: An execution trace of a multi-threaded program with threads t_1 t_2, t_3. (Dashed arrows represent the global trace σ_2)

The trace $\sigma_2 = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\rangle$ represents the execution shown in Fig. 3. The interleavings produced from σ_2 are $e_1 \rightsquigarrow e_3$, $e_3 \rightsquigarrow e_4$, $e_2 \rightsquigarrow e_5$, $e_6 \rightsquigarrow e_7$, $e_6 \rightsquigarrow e_8$, $e_8 \rightsquigarrow e_9$ and $e_7 \rightsquigarrow e_9$.

Light records the inter-thread interleavings $e_3 \rightsquigarrow e_4$, $e_6 \rightsquigarrow e_7$ and $e_6 \rightsquigarrow e_8$. We encode each interleaving for the constraint solver in the form $\{a - b - c - d\}$ where *a*, *b*, *c* and *d* represent the thread id from the write event, the event id for the interleaving write event, the thread id for the read event and the event id for the interleaving read event respectively.

The set of *write-read* interleavings for σ_2 is encoded as $\{\{3-3-2-4\}, \{2-6-3-7\}, \{2-6-2-8\}\}$. The constraint solver also accepts the intra-thread interleaving constraint set $\{\{3-3-3-7\}, \{2-6-2-8\}\}$.

The execution schedules $\langle 3-3, 2-4, 2-6, 2-8, 3-7\rangle$ and $\langle 3-3, 2-4, 2-6, 3-7, 2-8\rangle$ are generated by the constraint solver.

Our Z3Py code for the constraint solver is available online[3].

E. Record Phase results

Table III details our experimental results for the record phase. The first two columns detail the benchmarks and their respective application domains. The next two columns show the record log sizes for TPLAY and Light in megabytes. Base Time represents the native runtime for each benchmark in seconds. Column 6 of Table III shows the difference in time spent between TPLAY and Light. On average, Light achieves 44% of runtime overhead of TPLAY because TPLAY incurs runtime cost in creating and maintaining transaction data structures as well as the transaction atomicity checker $\mathrm{RT_m}$.

Our experimental results indicate an average of 25.44MB, which is a 13-fold improvement when compared to Light's average of 333.20MB. Specifically, TPLAY records a smaller log size for 13 out of 14 programs. Light experiences a slight gain for *mysql,* which does not produce enough transactions to highlight the efficiency of TPLAY.

Table IV shows the results on trace reduction for TPLAY and Light. Column 2 shows the number of transactions created by TPLAY for each program execution. Column 3 shows the number of transactional interleavings whilst column 4 shows the number of atomicity violations reported for each program execution. Finally, Column 5 shows the interleavings recorded by Light.

We also compare the number of transactional interleavings with the number of write-read interleavings recorded by Light. Table IV shows a 44-fold improvement in trace reduction by TPLAY over Light.

Atomicity violations were reported for two benchmarks *volrend* (98) and *radiosity* (2)**.** For each atomicity violation reported, TPLAY created a pair of interleavings, one interleaving was stored for the sourceThread in *aSet* and another was saved for the sinkThread in *bSet*. No atomicity violations are reported for 12 out of 14 benchmarks, thereby confirming our insight. This also means that these benchmarks could be replayed exclusively at the transactional level.

F. Replay Phase results

Table V presents the experimental results on replay phases for TPLAY. A successful replay run for TPLAY in our experiment was the reproduction of interleavings observed in

[2]github.com/testrepo007/TPLAY

[3]github.com/testrepo007/Constraint-solver

TABLE V
EXPERIMENTAL RESULTS FOR REPLAY PHASE.

Benchmark	Normalized Slowdown		# Successful Executions	% Replay Probability
	TPLAY	*Light*		
cholesky	0.86	-	45	90
fft	0.83	-	50	100
lu_cb	0.8	-	50	100
lu_ncb	0.77	-	50	100
ocean_cp	0.77	-	50	100
ocean_ncp	0.71	-	41	82
raytrace	0.72	-	50	100
radiosity	0.8	-	50	100
radix	0.77	-	50	100
volrend	0.79	-	50	100
water_nsquared	0.85	-	50	100
water_spatial	0.64	-	50	100
barnes	0.7	-	40	80
mysql	0.57	0.77	50	100
Mean	**0.76**	**0.77**	**48.3**	**96.6**

the record phase and the program output. A successful replay for Light was the reproduction of write-read interleavings for a program. Each subject program was run 50 times.

Recall that Light employs a constraint solver in generating possible schedules for replay. However, in our experiment we found the constraint solver strategy to be limited in constructing feasible schedules due to the high number of write-read interleavings generated for our benchmarks. Column 5 of Table IV shows the number of inter-thread write-read interleavings recorded by Light. With the exception of *mysql* (26 possible traces), the constraint solver did not return any output even after several hours of constraint solving. For *mysql*, Light succeeded in reproducing correct interleavings 80% of the time.

The *time spent* for each tool in the replay phase was calculated in a manner similar to the record phase (See paragraph 6 of subscetion C of Section V). The normalized runtime for each tool in the replay phase is calculated as the *(time spent for tool in replay phase / time spent for tool in record phase)*. The results are presented in columns 2 and 3 of of Table V. TPLAY's replay phase is able to execute at 76% of its record phase on average.

On average, TPLAY replays programs with a 96.6% probability, with a mean of 48.3 successful executions out of 50 executions. For *cholesky*, and *ocean_ncp*, TPLAY fails to achieve 100% probability due to the program outputs being different from that observed in the record phase.

G. Threats to Validity

The ability of the TPlay tool in the experiment to report atomicity violations was dependent on the algorithm in [28]. Our implementation of Light (and TPLAY) was also subject to inherent binary instrumentation limitations of Pin. Also the constraint solver was developed using the python library of z3 (Z3Py)[4]. Another way of encoding constraints or other constraint solvers may make Light able to produce thread schedules for its replay phase. TPLAY may also suffer overheads in the record phase mainly during the creation and maintainance of data structures for transactions. Threads not previously encountered in the record phase but observed in the replay phase may not be handled by TPLAY.

[4]github.com/Z3Prover/z3

VI. RELATED WORK

There are many existing deterministic replay techniques such as Samsara [26] and Odr [1] that are implemented for C/C++ applications based on the *pthread* execution model. TPLAY follows a similar implementation style to these techniques.

Some recent techniques such as DoubleTake [18] conduct record and replay in one sinlge phase. In these techniques, an execution is divided into epochs based on either irrevocable system calls or user-defined conditions. The programs state is logged just before each epoch creation and execution, and the epoch is analyzed afterwards. If any error is found in an epoch, the program state prior to the epoch is restored and the epoch is replayed to reproduce the error. Otherwise, the next epoch is created and executed. DoubleTake requires special hardware support whereas TPLAY does not. Unlike MobiPlay [25], TPLAY does not modify the underlying framework to facilitate replay. AggrePlay [24] tracks the number of read accesses by each thread prior to some write event, then aggregates these numbers of read accesses by all threads in a read vector. The read vector is then associated with the write event. AggrePlay however, incurs higher overheads in the record phase compared to TPLAY by recording all interleaving sets.

Checkpointing is often used in replay techniques. iReplayer [16] stores the system state in memory and enables user-customizable checkpointing rules. iReplayer acheives small log sizes by not monitoring data races during record. It may report them in the replay phase by iteratively replaying an epoch if there is a divergence from the thread interleavings observed during replay and the recorded thread interleavings. TPLAY does not use checkpointing and replays data races without iteration.

Processor-oblivious record and replay [27] focuses on recording the synchronization order for programs which employ task-parallelism. Such programs do not have any notion of threads or data and are inherently data-race free, e.g., Cilk programs. However, current mainstream software supports multi-threaded parallelism. TPLAY is applicable to most mainstream software.

Recording write-read interleavings has been explored by existing work. Light [17] records inter-thread and intra-thread write-read interleavings and uses a constraint solver to generate feasible execution schedules with the interleavings as constraints. TPLAY records write-read interleavings at a transactional level and does not require any constraint solver. Minimizing record impact on runtime overhead via thread-local data storage has been explored by existing work CLAP [12], which relies on constraint solving to generate execution schedules. CARE [13] maintains thread-local caches for shared memory locations, records cache-missed write-read interleavings but does not record write-read interleavings if the interleaving results in a cache hit. CARE records exact read-write interleavings unlike TPLAY. Some existing techniques focus on trace reduction strategies. Netzer [22] has proposed an adaptive tracing strategy which records the minimal amount of data required to replay a specific race condition. This is achieved by optimizing transitively-implied inter-thread dependencies similar to Light. TPLAY by default does not record interleavings at the event level.

Xu et al. [30] proposes a regulated transitive reduction algorithm which improves Netzer's trace reduction technique by generating artificial dependencies on inter-thread dependencies. TPLAY generates artificial dependencies but only when an atomicity violation is reported on an event within a transaction. OCTET [5] avoids creating thread interleavings by associating each shared object with a thread-local state variable and a mechanism based on the concurrent read, exclusive write strategy. TPLAY reduces thread interleavings by executing threads' instructions as atomic code regions and recording interleavings on these atomic regions.

Existing replay techniques like Stride [31] include a search process and attempt to re-construct the missing interleavings before generating a trace with the targeting output in its replay phase. Bbr [7] uses a predefined set of location checkpoints to minimize record log sizes as well as a symbolic-execution based search process to generate feasible traces involving the checkpoints. TPLAY does not rely on a search process, symbolic execution nor any offline phase.Similar to Stride, ODR aims to reproduce a specific program output by extracting some execution trace data from a core dump and generating a trace through a search process.

Deterministic replay is applied in other fields of discipline. DETER [15] is a deterministic replay tool used in recording transmission control protocol (TCP) packets to reproduce communication network states. TPLAY does not focus on replay of communication networks.

VII. Conclusion

Existing deterministic replay techniques proposed strategies which attempt to reduce record log sizes and achieve successful replay. However, these techniques still generate large logs and and achieve replay only under certain conditions. We have proposed a solution based on the division of the sequence of events of each thread into sequential blocks called transactions. Our insight is that there are usually few to no atomicity violations among transactions reported during a program execution. We have presented TPLAY, a novel deterministic replay technique which records thread access interleavings on shared memory locations at the transactional level. TPLAY also generates an artificial pair of interleavings when an atomicity violation is reported on a transaction. We present an experiment using the Splash2x extension of the PARSEC benchmark suite. Experimental results have indicated that on average, TPLAY experienced a 13-fold improvement in record log sizes and achieved a high replay probability.

Future work includes solving the problem of high runtime overhead in the record phase for TPLAY.

References

[1] G. Altekar and I. Stoica, "Odr: Output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 193–206. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629594

[2] D. F. Bacon and S. C. Goldstein, *Hardware-assisted replay of multiprocessor programs*. ACM, 1991, vol. 26, no. 12.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[4] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "Doublechecker: Efficient sound and precise atomicity checking," *SIGPLAN Not.*, vol. 49, no. 6, pp. 28–39, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594323

[5] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang, "Octet: Capturing and controlling cross-thread dependences efficiently," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 693–712. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509519

[6] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: A survey," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 17:1–17:47, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2790077

[7] A. Cheung, A. Solar-Lezama, and S. Madden, "Partial replay of long-running applications," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 135–145. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025135

[8] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/281035.281041

[9] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[10] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs," *SIGPLAN Not.*, vol. 43, no. 6, pp. 293–303, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375618

[11] J. Huang, P. Liu, and C. 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*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882323

[12] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," *SIGPLAN Not.*, vol. 48, no. 6, pp. 141–152, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462167

[13] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "Care: Cache guided deterministic replay for concurrent java programs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 457–467. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568236

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] Y. Li, R. Miao, M. Alizadeh, and M. Yu, "DETER: Deterministic TCP replay for performance diagnosis," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 437–452. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/li-yuliang

[16] H. Liu, S. Silvestro, W. Wang, C. Tian, and T. Liu, "ireplayer: In-situ and identical record-and-replay for multithreaded applications," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 344–358. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192380

[17] P. Liu, X. Zhang, O. Tripp, and Y. Zheng, "Light: Replay via tightly bounded recording," *SIGPLAN Not.*, vol. 50, no. 6, pp. 55–64, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2738001

[18] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: Fast and precise error detection via evidence-based dynamic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 911–922. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884784

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065034

[20] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 289–300.

[21] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 284–295, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1080695.1069994

[22] R. H. B. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," in *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, ser. PADD '93. New York, NY, USA: ACM, 1993, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/174266.174268

[23] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 177–192. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629593

[24] E. Pobee and W. K. Chan, "Aggreplay: Efficient record and replay of multi-threaded programs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 567–577. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338959

[25] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 571–582.

[26] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Leveraging hardware-assisted virtualization for deterministic replay on commodity multi-core processors," *IEEE Transactions on Computers*, vol. 67, no. 1, pp. 45–58, Jan 2018.

[27] R. Utterback, K. Agrawal, I.-T. A. Lee, and M. Kulkarni, "Processor-oblivious record and replay," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 145–161. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018764

[28] S. Wu, "Efficient and sound dynamic atomicity violation checking," Ph.D. dissertation, City University of Hong Kong, 2016. [Online]. Available: https://scholars.cityu.edu.hk/en/theses/theses(b64a7371-a086-434c-9ff4-a042a3c8effa).html

[29] M. Xu, R. Bodik, and M. D. Hill, "A" flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 122–133.

[30] M. Xu, M. D. Hill, and R. Bodik, "A regulated transitive reduction (rtr) for longer memory race recording," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 49–60, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168919.1168865

[31] J. Zhou, X. Xiao, and C. Zhang, "Stride: Search-based deterministic replay in polynomial time via bounded linkage," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 892–902. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337328