AggrePlay: Efficient Record and Replay of Multi-threaded Programs[†]

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

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

ABSTRACT

Deterministic replay presents challenges and often results in high memory and runtime overheads. Previous studies deterministically reproduce program outputs often only after several replay iterations or may produce a non-deterministic sequence of output to external sources. In this paper, we propose AggrePlay, a deterministic replay technique which is based on recording read-write interleavings leveraging thread-local determinism and summarized read values. During the record phase, AggrePlay records a read count vector clock for each thread on each memory location. Each thread checks the logged vector clock against the current read count in the replay phase before a write event. We present an experiment and analyze the results using the Splash2x benchmark suite as well as two real-world applications. The experimental results show that on average, AggrePlay experiences a better reduction in compressed log size, and 56% better runtime slowdown during the record phase, as well as a 41.58% higher probability in the replay phase than existing work.

CCS CONCEPTS

• **Software and its engineering → Synchronization**; **Software verification**; *Dynamic analysis.*

KEYWORDS

Concurrency, Deterministic Replay, Multi-threading.

ACM Reference Format:
Ernest Pobee and W. K. Chan. 2019. AggrePlay: Efficient Record and Replay of Multi-threaded Programs. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338959

[†]This research is supported in part by the GRF of HKSAR Research Grants Council (project nos. 11214116 and 11200015), the HKSAR ITF (projectno. ITS/378/18), and the CityU MF_EXT (project no. 9678180).
[‡]Correspondence Author

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338959

1 INTRODUCTION

Deterministic replay is a process of recording data from a program's execution and guiding a subsequent execution towards a specific program state and/or program output [18, 25, 26, 30, 31]. It can be used in, for example, software debugging [15, 17, 18] and software testing [13, 27]. Deterministic replay consists of two phases, namely record phase and replay phase.

In the record phase, such a technique logs data from an execution of a multithreaded program. Logged data includes thread access interleavings to shared memory locations and synchronization objects' access orders. The replay phase is an execution of the program guided with an execution schedule generated based on the data output from the record phase. Some techniques [11, 31] further include an offline phase in between the record and replay phases to resolve some data missed from the record phase.

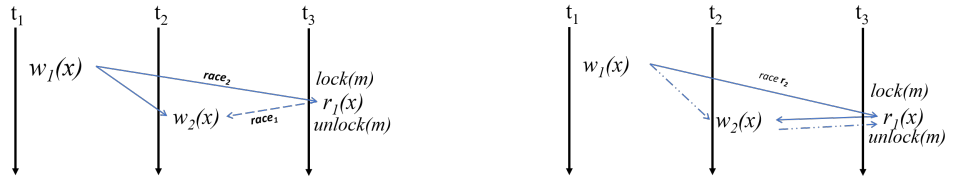
We categorize deterministic replay techniques into hardware-based [2, 20, 21, 29] and software-based [10, 11, 17] for brevity. Hardware-based techniques may require specialized hardware which may not be available on commodity systems [22, 25, 29]. We focus on software-based deterministic replay techniques in this work due to their ease of deployment compared to hardware-based techniques.

Software-based replay techniques need to tackle data races[1] [6, 12]. As such, replay techniques record thread access interleavings on shared memory locations and synchronization objects during the record phase to generate an execution schedule which produces the required interleavings in the replay phase. There are three types of interleavings recorded on shared memory locations: *read-write*, *write-read* and *write-write*. However, logging all the interleavings in an execution incurs high memory and runtime overheads. As such, some techniques record subsets of interleavings [17, 29, 31]. We refer to the proportion of interleavings recorded as the "*degree of recording fidelity*" [6, 8].

We further categorize software-based deterministic replay techniques (hereafter referred to as replay techniques) into order-based and search-based replay techniques for brevity.

Order-based techniques [10, 17]: These techniques record the order of thread accesses to shared memory locations as well as synchronization objects. They may record all sets of thread access interleavings (*read-write*, *write-read* and *write-write*) or subsets of them. In the simplest form, order-based techniques protect instrumentation events with some lock object, ensuring that the

[1]A data race occurs when two or more threads accessing a shared memory location without proper synchronization and at least one of the access operations is a write operation.



(a) trace $\sigma_1=\langle w_1(x), r_1(x), w_2(x)\rangle$ The interleaving $r_1(x) \rightsquigarrow w_2(x)$ is not recorded.

(b) Replay trace $\sigma_{sr1}=\langle w_1(x), r_1(x)\rangle$ Data race $race_1$ is missed by Light.

Figure 1: Illustration of thread interleavings recorded by existing replay techniques.

correct thread interleaving order is preserved. Thus, they exhibit high recording fidelity. However, logging all thread interleavings is costly due to the frequent use of locks during the record phase. Moreover, excessive event matching in the replay phase reduces concurrency and increases the execution overhead. Replay techniques such as LEAP [10] use more than one lock object to protect read and write accesses by to different shared memory locations, increasing parallelism. However, LEAP incurs high memory and runtime overheads as all events (both read and write) executed on each shared memory location are recorded in a serialized manner.

Search-based techniques [1, 11, 23]: These techniques like Stride [31] and ODR [1] sacrifice recording fidelity in exchange for a speedup in record time and low memory usage. Stride records the values read by threads and couples each read value with a previous write access to some shared memory location. Whilst search-based techniques may sometimes reproduce the output of a program, the process often requires several iterations and may reduce the efficiency of such techniques. Another downside is that low record fidelity means search-based techniques may generate new data races or fail to reproduce data races during the replay phase. Stride achieves up to 2.5x in slowdown reduction on average compared to order-based technique LEAP. However, a search-based strategy does not imply low overheads. For example, the order-based technique Light achieves up to 4x in memory overhead reduction in comparison to Stride.

An example is shown in Figure 1a where the execution trace is given as $\sigma_1=\langle w_1(x), r_1(x), w_2(x)\rangle$. There are two data races $race_1$ and $race_2$ on the interleaving $r_1(x) \rightsquigarrow w_1(x)$ and $r_1(x) \rightsquigarrow w_2(x)$ respectively. Assuming both writes are protected by a lock, Light records the *write-read* dependency from the trace σ_1 as $w_1(x) \rightsquigarrow r_1(x)$. Light will reproduce $race_2$ but will miss the dependency $r_1(x) \rightsquigarrow w_2(x)$ and transitively $race_1$. According to the replay strategy of Light, the event $w_2(x)$ may not execute to reproduce $race_1$ [17]. As such, Light will never reproduce $race_1$. Also, supposing the events $w_1(x)$ and $w_2(x)$ were writes to output devices, Light cannot guarantee the two events will be outputted in a deterministic order. Therefore, Light's strategy is not *output-determinisitc*.

We propose our solution with the following insight: A list of read events on a shared memory location by each thread are strictly ordered and can be summarized and recorded before the next immediate write (and *write-write* interleavings are globally ordered). As such, summarized reads by different threads could be ordered before some write to that shared memory location.

In this paper, we present AggrePlay, a novel deterministic replay technique which exploits the advantages of strictly ordered thread-local operations. AggrePlay keeps a thread-local read count vector clock for each shared memory location. Such a read count vector clock is updated in two different ways during the record phase. When a thread reads a value on each shared memory location, the index for the thread in its read count vector clock is updated. Prior to a thread's write on a shared memory location, the thread's read vector is updated with read values from all other threads' read count vector clock. The executing thread's read vector is constructed before the current write event in the log. This enables us to keep track of read-write interleavings.

During the replay phase, each thread maintains a read vector similar to the record phase. On a write to each shared memory location, the read vector for the executing thread is updated with read values from all other threads' read vectors. The thread's read vector is matched with the corresponding read vector from the record log. The thread executes the write operation if the read vectors are matched successfully.

To evaluate the performance of AggrePlay, we design our experiments to answer the following research questions:

RQ: Can AggrePlay achieve smaller log sizes and runtime slowdown compared to existing state of the art in the record phase?

Our experimental results show an average of 6x reduction in log sizes as well as 66% average slowdown compared to Stride. Light however, achieves 0.75x on average of AggrePlay log sizes.

RQ: How much do the various space optimization techniques employed by the replay strategies affect record?

Data recorded for write-read interleavings are compressed for better storage efficiency, and all techniques employ different space optimization approaches. Our experimental results show that AggrePlay achieves a 3.6x compression for write-read logs compared to Stride. AggrePlay is more efficient than Stride on 12 out of 17 benchmarks. Light achieves a 1.3x write-compression on average over AggrePlay.

RQ: Is AggrePlay able to replay applications in high probability?

AggrePlay reproduces all the interleavings (and the output) with 86.8% probability across all 17 benchmarks, higher than Stride by 41.58%. AggrePlay also incurred 2.85x slowdown, compared to 3.54x of Stride during replay.

2 PRELIMINARIES

This section details preliminary information used in this paper.

Table 1: Preliminary Information

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

2.1 Execution Trace

An execution trace $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ is a sequence of operations observed from the execution of a software program. An operation $\boldsymbol{e}$ 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. We omit them for brevity.

2.2 Read Count Vector Clocks

We track *read-write* interleavings with the aim of enforcing these interleavings in the replay phase using read count vector clocks. A read count (RC) vector clock (*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.

3 RUNNING EXAMPLE

We present a running example to motivate our work. Figure 2 illustrates an execution of a multithreaded program with three threads t_1, t_2 and t_3, as well as 9 operations which are write and read, labeled e_1 through to e_9. In Figure 2, t_1 executes a write on location x. Thread t_2 then executes a write e_2 on y. Thread t_3 then executes a write e_3 on x and t_2 executes a read e_4 on x. t_3 then writes on y. t_2 then writes on z. t_3 executes e_7 which is a read on z. Thread t_2 executes a read on z. Finally, t_1 executes e_9, which is a write on z. The trace $\sigma_2 = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9 \rangle$ is produced. The set $\{e_1 \rightsquigarrow e_3, e_3 \rightsquigarrow e_4, e_2 \rightsquigarrow e_5, e_6 \rightsquigarrow e_7, e_6 \rightsquigarrow e_8, e_8 \rightsquigarrow e_9, e_7 \rightsquigarrow e_9\}$ is produced as thread interleavings.

To capture the trace σ_2, a simple strategy is to log all the events. For Figure 2, nine locks events will be inserted during the instrumentation for correctness. However, this recording strategy

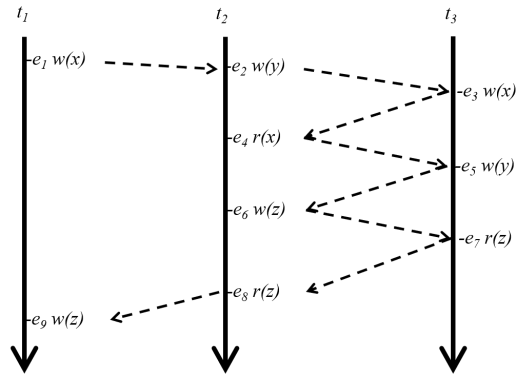


Figure 2: Running example: An execution trace of a multithreaded program t_1 t_2, t_3. (Dashed arrows represent the global trace σ_2)

reduces parallelism and increases runtime slowdown for the record phase.

The existing state of the art Light records only flow dependencies (*write-read* interleavings). Synchronization events are also encoded as read and write events. For trace σ_2, Light records the *write-read* interleavings $\{e_3 \rightsquigarrow e_4, e_6 \rightsquigarrow e_7, e_6 \rightsquigarrow e_8\}$ then passes the set of interleavings to a constraint solver as constraints. Light further includes constraints over the set of constraints per shared memory location. The constraint solver constructs a feasible trace for subsequent replay.

During the replay phase, Light attempts to replay the execution using a member trace of the trace set generated by the constraint solver. The events e_1, e_2 , e_5 and e_9 are not executed by Light since they are considered blind writes by Light. Blind writes are write events not involved in any *write-read* dependence [17]. As such, Light is not able to reproduce the program state by design. The main drawback is Light is unable to guarantee a deterministic order of output for any set of serialized write events which may include blind writes. Light is also limited by the capacity of constraint solvers to generate traces.

Stride [31] is a search-based replay technique which reduces recording overhead by maintaining a write version (counter) for all writes on a shared memory location. To record data on write-read interleavings, it pairs the read value ($value_e$) of each read operation with a possible matching write version ($version_e$).

For trace σ_2, Stride records the *write-write* interleavings $e_1 \rightsquigarrow e_3$ and $e_6 \rightsquigarrow e_9$. The candidate *write-read* pairs $\langle value_{e_4}, version_{e_3} \rangle$, $\langle value_{e_7}, version_{e_6} \rangle$ and $\langle value_{e_8}, version_{e_6} \rangle$ are recorded. During a write operation in the replay phase, the *write-read* pairs are used to infer possible interleaving candidates with the $version_e$ being the upper and the $value_e$ being the matching criterion.

For the trace σ_3 in Figure 3, t_2 executes two writes, e_i and e_k after t_3 executes e_7. Suppose that e_5 and e_k write the value **1**, but e_i writes the value **0**. In this case, Stride may record the pair $\langle value_{e_7}, version_{e_k} \rangle$ because *write-read* interleavings are not ordered in the record phase. During the replay phase, the interleaving $e_6 \rightsquigarrow e_7$ will be missed by Stride because the

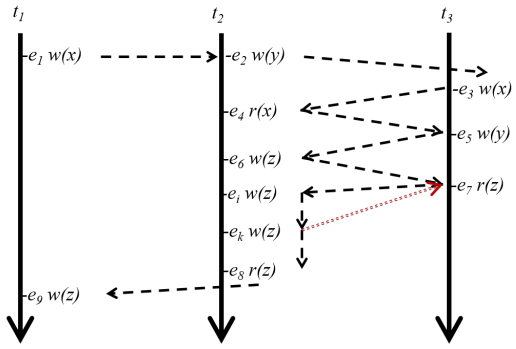


Figure 3: Modified running example with trace σ_3. (Double compound arrow represents the recorded pairing $\langle value_{e_7}, version_{e_k} \rangle$).

interleaving $e_k \rightsquigarrow e_7$ matches Stride's search criterion for the pairing $\langle value_{e_7}, version_{e_k} \rangle$. Note that Stride can correctly infer $e_6 \rightsquigarrow e_7$ if there is no write operation after e_6 which writes to the same memory location having the same value as e_6.

In the case of Light, let us consider the events e_2 *and* e_5. e_2 will be correctly ordered by Light when outputting to external devices only because of the *write-read* interleaving $e_3 \rightsquigarrow e_4$. However, the events e_1 and $e_3$are not explicitly ordered by Light and may be incorrectly ordered.

AggrePlay records *write-write* and *write-read* interleavings with optimizations as well as *read-write* interleavings in a novel strategy. As such, it can keep track of all events in both traces $\sigma_2$and σ_3. We present our AggrePlay algorithm in the next section.

4 AGGREPLAY ALGORITHM

In this section, we present AggrePlay. The following notations are used in our algorithms:

- W_x: The write access list for shared object *x*.
- repW_x: The write access list for shared object *x* during the replay phase.
- L_m: The lock acquisition list for lock *m*.
- EX_t: The thread-local read-write access list for thread *t*.
- WR_x: The write-read access list for shared object *x*.
- Tid(*e*): The executing thread of operation *e*.
- Lock(*e*): The lock object acquired in operation *e*.
- var(*e*): The shared object being accessed by operation *e*.
- $RC_{t.x}$: The RC vector clock for thread *t* for shared memory location x.
- *init*($RC_{t.x}$) : this function instantiates all the elements of RC_t to 0.
- *inc*($RC_{t.x}$*[t']*) : this function increments the read count value of the thread *t'* in RC_t by 1 for shared memory location x.
- last_write(var(*e*)): The last write event to the shared memory location in event *e*.
- lw_x: Last write event to a shared memory location *x*.
- execute(*e*): event *e* is executed.
- *t*.yield(): The executing thread t waits for other threads to advance without blocking.
- top(*Input*): reads data from the first index of *Input*. (*Input* = W_x | L_m | *EX* | WR_x).
- pop(*Input*): removes data from first index position of *Input*.

4.1 AggrePlay Record Phase

The record phase of AggrePlay is presented in Algorithm 1. Lines 1-3 instantiate the RC vector clock for each thread for each shared memory location in the program execution to $\perp_{RC}$, where every element in the vector clock $\perp_{RC}$ is zero. Lines 5-11 detail the *onWrite* function. On a write to a shared memory location, AggrePlay first checks the read accesses to that memory location by the other threads. Line 7 invokes the *updateReadVectors* function. This function enables us to keep track of read-write interleavings. Our insight is that due to read events on all shared memory locations being ordered by that thread, and all write events on each shared memory location are globally ordered, we need not track read-write interleavings at the shared memory level. Rather, we track the total number of reads (for each shared memory location) for each thread prior to a write event.

The *write* event is executed at line 8. The *write* event is added to the write access list of the shared memory location (line 9).

ALGORITHM 1: -The AggrePlay recording algorithm

INPUT: Execution trace $\sigma := \langle e_1, e_2, e_3, \ldots, e_n \rangle$, $e \in$ Event
OUTPUT: $EX_{Tid(e)}$; $W_{var(e)}$; $WR_{var(e)}$; $L_{Lock(e)}$;

```
(1)  for each t,e ∈ Thread, Event {
(2)     init(RC_t.var(e));
(3)  }
(4)  // write access performed by event e
(5)  onWrite (Event e) do
(6)     Sync {
(7)        updateReadVectors (e);
(8)        execute(e); // the write instruction is executed
(9)        W_var(e) := W_var(e) ^ ⟨e⟩;
(10)    }
(11) end onWrite
(12) // read access returned by event e
(13) onRead(Event e) do
(14)    t := Tid(e);
(15)    RC_t.var(e) [t] := inc(RC_t.var(e) [t]);
(16)    Synch { lw = last_write(var(e)); }
(17)    execute(e); // the read instruction is executed
(18)    WR_var(e) := WR_var(e) ^ ⟨lw, e⟩;
(19) end onRead
(20) updateReadVectors(Event e):
(21)    for each t' ∈ Thread and RC_t'.var(e) [t'] > 0 do
(22)       RC_Tid(e).var(e)[t'] := RC_t'.var(e) [t'];
(23)    end for
(24)    EX_Tid(e) := EX_Tid(e) ^ ⟨RC_Tid(e), e⟩;
(25) end updateReadVectors
(26) onLockAcquire(Event e):
(27)    execute(e);
(28)    L_Lock(e) := L_Lock(e) ^ ⟨Tid(e)⟩;
(29) End onLockAcquire
```

For lines 13-19, on every read to a shared memory, the executing thread and shared memory location are obtained. Then, the index

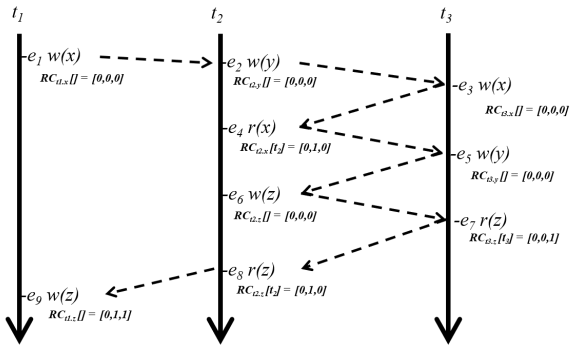


Figure 4: AggrePlay record phase for running example.

for the current thread in its RC vector clock is incremented by 1 at line 15. The last write to the shared memory location for the current read operation is retrieved at line 16. The read event is executed at line 17 and is ordered before the current read in the write-read access list of the shared memory location at line 18.

The *updateReadVectors* (lines 20-25) function is invoked at line 7 of the *onWrite* function. This function retrieves the read count values for every other thread in the set of Threads with a read count value greater than zero. The updated read count for the current thread is then paired with the write event and added to the thread-local read-write access list.

Lines 26-30 details the lock acquisition function. The lock acquisition event is executed at line 27. The executing thread is appended to the lock's access order list at line 28.

Figure 4 illustrates the RC vector clock values for the trace σ_2 in the running example. On events e_1, e_2 and e_3, the write access lists for shared memory locations x and y are updated by t_1,t_2 and t_3, respectively. On event e_4, the read value for t_2 in $RC_{t_{2.x}}[t_2]$ is incremented by 1. The *write-read* dependency $e_3 \rightsquigarrow e_4$ *is also recorded by AggrePlay* Events e_5 and e_6 also result in the *write access lists* for y and z being updated by t_3 and t_2, respectively. On event e_7, the read value in $RC_{t_{3.z}}[t_3]$ is incremented by 1. The *write-read* dependency $e_6 \rightsquigarrow e_7$is also recorded. Then $RC_{t_{2.z}}\ [t_2]$ is incremented by 1 on handling e_8. On e_9, the RC vector clock for t_1 is updated with the read count values from t_2 and t_3. Then $\mathrm{RC}_{t_{1.z}}[0,1,1]$, is constructed before event e_8 and saved in the record. The record output for the trace is shown in Figure 5.

4.2 AggrePlay Replay Phase

The AgrrePlay replay phase is shown in Algorithm 2. The output from the record phase (EX, $W_{\mathrm{var(e)}}$, $WR_{Tid(e)}$, $L_{Lock(e)}$) are used as input in the replay phase.

$$t_{1.z} = \langle\langle[011], e_9\rangle\rangle \qquad \mathrm{W_x} = \langle e_1, e_3\rangle$$
$$\mathrm{WR}_x = \langle e_3 \rightsquigarrow e_4\rangle \qquad \mathrm{W_y} = \langle e_2, e_5\rangle$$
$$\mathrm{WR_z} = \langle e_6 \rightsquigarrow e_7\rangle \qquad \mathrm{W_z} = \langle e_6, e_9\rangle$$

Figure 5: AggrePlay output data for trace σ_2 in Figure 2

The $repW_{var(e)}$ is the access list for the shared memory location *var(e)* in the replay phase. Line 3 instantiates the RC vector clocks for each thread in the program execution to $\perp_{RC}$, where every element in the vector clock $\perp_{RC}$ is zero.

Lines 6-15 detail the *onWrite* function: During a write event, an *updateReadVectors* function is invoked. This function updates the currently executing thread's RC vector clock with read count values from all other threads. Next, the write event is matched with the first element in the write input list ($W_{var(e)}$) for that shared memory location *var(e)* at line 8. If the two events match, then the *checkReadVector* function is invoked to ensure that all necessary reads have been executed prior to the write event. If *checkReadVector* returns *true*, the write event is executed, the $repW_{var(e)}$ is updated with the *write* event. The first element is removed from the $W_{var(e)}$ input list (lines 10-12). Otherwise, if *checkReadVector* returns *false*, the thread *yields* the processor at line 13. Lines 17-27 show the *onRead* function. During a *read* event, the last *write* event for the shared memory location in the replay phase is matched to the first element in the $WR_{var(e)}$ input list (line 13). If the two elements match, the *read* event is executed (line 20). The read count vector clock for the current thread is incremented, then the first element in the $WR_{var(e)}$ input list is removed (lines 22-23). Otherwise, if the condition at line 19 is evaluated to *fals*e, the thread yields at line 25. This way, each thread can individually replay the correct sequence of read events without synchronization with other threads.

Lines 28-34 detail the *onLockAcquire* function. If the currently executing thread does not match the first element of the input list $L_{Lock\ (e))}$, then the thread yields (lines 29-30). Otherwise, the lock acquisition event is executed and the first element of the input list $L_{Lock\ (e))}$ is removed (lines 32-33).

The *updateReadVectors* function at lines 35-39 updates the RC vector clock of the currently executing thread with read count values from the RC vector clocks of threads other than the currently executing thread with read count values greater than zero. This function is invoked in the *onWrite* function at line 7.

The *checkReads* function is detailed at lines 40-52. This function is invoked at line 8 in the *onWrite* function. During the record phase, we record a null line instead of the original write-read interleaving which is removed by **O1.** Lines 41-44 show the implementation for handling such null lines (Θ) in the replay phase. If the first element of EX_t is a null line**,** then the first element is removed and *checkReads* returns a *true* value. The thread is allowed to execute as there are no preceding read(s) or the preceding read(s) are ordered by the executing thread. Lines 45-52 match the read count value for all threads other than the current thread t in the read count vector clock of the current thread against the first element of the $\mathrm{EX_t}$ input list (line 46). If the condition at line 46 evaluates to *false*, the function returns *false*. If the condition at line 45 is evaluated to *true*, the algorithm at line 50 removes the first element of the EX_{t} input list. The function then returns *true* at line 51.

AggrePlay replays the execution in Figure 2 as follows: t_3 attempts to execute e_3 but the match between e_3 and the first element in the write access list W_{x} evaluates to false. t_3 yields the underlying thread scheduler. t_1 then attempts to execute e_1. As no reads have been executed prior to e_1, the *checkReads* function evaluates to true. e_1 is then matched with the first element in the write access list for W_x. t_1 proceeds to execute e_1 which is then

added to $repW_x$ and the matched element is removed from W_x. t_2 is scheduled to execute e_2. The matched element of W_y is removed. When t_2 tries to executes e_3, the last write event on x (e_3) is retrieved from $repW_x$ and matched with the first element in WR_x. Since the two events do not match, t_2 yields the scheduler. t_3 then executes e_3 and the matched element is removed from W_x. t_2 then executes e_4, then increments $RC_{t_{2.x}}[t_2]$ by 1. The matched element is removed from WR_x. t_2 executes e_6 as no reads have been executed on z prior to e_6. Next, t_3 then executes e_5 and the matched element is removed from W_y. t_3 executes e_7 and updates $RC_{t_{3.z}}[t_3]$ by *1*. t_2 executes e_8 and updates $RC_{t_{2.z}}[t_2]$ by *1*. Finally t_1 tries to execute e_9. The *updateReadVectors* function updates $RC_{t_{1.z}}$ with the read count values from $RC_{t_{2.z}}$ and $RC_{t_{3.z}}$. t_1 executes e_9 when e_9 is matched with the first element in W_z and the *checkReads* function returns *true*.

5 EVALUATION

5.1 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 16.04 Linux with 4 Intel Xeon(R) CPU E7- 4850 v3 @ 2.20Ghz processors, as well as 16GB of RAM.

We implemented AggrePlay, Light and Stride using Intel PIN version 3.0-76991 [19]. To be specific, for each tool, we implemented two separate pintools, one each 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](See Appendix[2]). The original implementation of Light is made available without the constraint solver and replay phases. Both can only handle Java applications. We also implemented Stride by following the paper despite the unavailability of implementation by the authors. A precaution taken was to test the correctness of our implementations of Stride and Light on different variants of a small benchmark we developed that includes different thread-interleaving code patterns.We also used code inspection on our implementations.

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [3], specifically *barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_nsquared, water_spatial* as well as kernel applications *cholesky, fft, lu_cb, lu_ncb, radix*. We selected the Splash2x extension of PARSEC for its focus on concurrent computation on parallel machines.

We also include 2 real-world applications/simulations including mysql 5.6.28 database server and apache 2.0.65 webserver. We further include an implementation of blockchain[3] to test the robustness of our tool for emerging software technology.

ALGORITHM 2: -The AggrePlay replay algorithm

INPUT: EX; $W_{var(e)}$; $WR_{Tid(e)}$; $L_{Lock(e)}$; // *output from record phase*
OUTPUT: Program output; //*valued outputted by subject program*

```
(1)  for each t ∈ Thread do
(2)     init(RC_t);
(3)     init(EX_t);
(4)  end for
(5)  // write access performed by Event e
(6)  onWrite (Event e) do
(7)     updateReadVectors(e):
(8)     if (e = top(W_var(e) )) ∧ checkReads( Tid(e)) then
(9)        execute(e);
(10)       repW_var(e) := repW_var(e) ^ ⟨e⟩;
(11)       pop(W_var(e));
(12)    else
(13)       Tid(e).yield();
(14)    end if
(15) end onWrite
(16) // read access returned by operation e
(17) onRead(e):
(18)    /* match write event from WR Input list */
(19)    if (last_write(var(e)) = top(WR_var(e)).lw) then
(20)       execute(e);
(21)       t = Tid(e);
(22)       RC_t [t] := inc(RC_t [t]);
(23)       pop(WR_var(e));
(24)    else
(25)       Tid(e).yield();
(26)    end if
(27) end onRead
(28) onLockAcquire(e):
(29)    if (Tid (e) ≠ top(L_Lock (e))) then
(30)       Tid(e).yield();
(31)    else
(32)       execute(e);
(33)       pop(L_Lock (e));
(34) End onLockAcquire
(35) updateReadVectors(e):
(36)    for each 't ∈ Thread and RC't ['t] > 0 do
(37)       RC_Tid(e)['t] = RC't ['t];
(38)    end for
(39) end updateReadVectors
(40) checkReads(thread t):
(41)    if top(EX_t) = Θ then
(42)       pop(EX_t);
(43)       return true;
(44)    end if
(45)    for each t' ∈ Thread do
(46)       if (RC_t [t'] < top(EX_t.RC[t']) then
(47)          return false;
(48)       end if
(49)    end for
(50)    pop(EX_t);
(51)    return true;
(52) end checkReads
```

[2]Github Link https://github.com/testrepo007/AggrePlay-Appendix
[3]Github Link https://github.com/teaandcode/TestChain

Table 2: Execution metadata for benchmarks used in our experiment. All benchmarks were configured with 8 worker threads except *apache*, *mysql*, and *Test Chain* which were configured with 2 threads.

Benchmarks	# of Events		
	Write	Read	Locks
cholesky	1.16 x 10^7	2.9 x 10^8	22,066
fft	1.6 x 10^9	3.0 x 10^9	66
fmm	8.0 x 10^8	6.2 x 10^9	6.3 x 10^5
lu_cb	2.9 x 10^9	6.0 x 10^9	2,082
lu_ncb	2.9 x 10^9	6.0 x 10^9	2,082
ocean_cp	1.6 x 10^9	5.6 x 10^9	6,666
ocean_ncp	2.1 x 10^9	5.6 x 10^9	6,506
raytrace	1.0 x 10^9	15.2 x 10^9	985,599
radiosity	1.2 x 10^9	1.3 x 10^9	481,866
radix	6.7 x 10^8	1.1 x 10^9	217
volrend	8.6 x 10^8	3.5 x 10^9	705,342
water_nsquared	10.2 x 10^9	26.0 x 10^9	160,298
water_spatial	3.0 x 10^9	8.8 x 10^9	315
barnes	6.5 x 10^9	11.5 x 10^9	1.0 x 10^6
mysql	5.1 x 10^5	1.6 x 10^6	21
apache	5.8 x 10^5	6.5 x 10^5	6
Test Chain	4.17 x 10^{11}	8.14 x 10^{11}	0

5.3 Methodology

To carry out our experiment, we first conducted a "dry" run on each benchmark using a pintool with no instrumentation functions. We recorded the execution time as the *base time* for each benchmark in the "dry" run.

In the record phase, each Splash2x benchmark program was configured with 8 worker threads with the *gcc-pthreads* option and the *simlarge* workload. The inputs for *apache* and *mysql* are *Apache (Bug #25520)* and *MySQL (Bug #85413)* respectively. This configuration provided each program adequate concurrency and input. The blockchain implementation was set to mine 2 blocks (append two transactions) to a block chain.

5.4 Thread Abstraction and Matching

To ensure that any pintool matched threads between record and replay runs, we abstracted each thread as a pair of unsigned integers abs_t := <*a*,*b*>, where *a* represents the child count value assigned to the thread by its parent thread and *b* represents the index value for the parent thread. We assumed that the main thread (usually thread 0) always begins first in each run and as such we did not abstract and record it. During thread creation, we mapped each thread's index to the thread's uniquely assigned system ID. (*e.g.,* all threads created by the main thread had a pairing abs_t := <*a*,*0*>, where a was the child count value under the main thread, and 0 is main thread's index value). The abstraction pair was then outputted to a log file. On thread creation during replay, the frequency of encountering a particular thread as a parent index was matched against the first item in the logged pair of values (i.e., $freq_{Parent\_ID}$ == *abs.a*).

5.5 Record and Replay Setup

During the record phase, each thread kept a thread-local counter, one each for read and write events to shared memory locations and synchronization objects respectively. Each thread also kept a thread-local data structure to maintain the thread's RC vector for each shared memory location. The write file was accessed by all the threads and we used a lock in moderating accesses to it. AggrePlay maintained internal data structures for writes, reads and synchronization events which were indexed by the shared memory locations and locks (in the case of synchronization events). AggrePlay stored all recorded data in memory and wrote to the log file when the benchmark terminated, with the exception of *read-write* data which was written to file intermittently. With the exception of thread-local data structures, accesses to all remaining data structures was controlled by locks. For Stride, we used locks to protect write events and synchronization events being recorded to the log file. Apart from write events, thread-local read events and their associated read values were not protected by locks based on the algorithm of Stride.

We recorded *time spent* (total processor time spent on each execution) using the clock function[4]. The slowdown factor was computed as the *time spent* divided by the *base time*.

For AggrePlay, each thread kept a local counter for all events. The record log format for synchronization events was a list of pairs in the form of $\langle a, b \rangle$ where *a* represents the *thread id* and *b* refers to the value for th*is* thread local counter. The write logs also feature a similar format to the synchronization logs.

Each *read-write* interleaving is recorded during a write event by some thread on some shared memory location. As such, the *read-write* log was recorded as a pair $\langle RC, write \rangle$ where *RC* refers to the read count vector clock of the current thread and *write* refers to the write event.

For Stride, the write and read event pairs which made up write-read interleaving candidates were recorded separately. We also implemented the last one value predictor for the write-read candidate interleavings as described in the paper [31].

We have made our AggrePlay implementation available online for data reproduction [5]

5.6 Record Phase Results

Table 3 shows our experimental results. The first and second columns show the names of the benchmarks and their application domains. The third, fourth and fifth columns show the log sizes (in MB) for AggrePlay, Light and Stride respectively after compression with *gzip*. The sixth column shows the base execution time for each benchmark with no instrumentation or analysis. The seventh, eighth and ninth columns show the slowdown factor for AggrePlay, Light and Stride with respect to the base time. The last two columns show the ratio of slowdown factor for Light and Light over AggrePlay.

From Table 3, AggrePlay resulted in an average of 44.39 MB, which was a 6-fold improvement on average compared to Stride. Out of 17 benchmarks, AggrePlay recorded a smaller log size for 16 of these subject programs; whereas Stride experienced about

[4] `http://man7.org/linux/man-pages/man3/clock.3.html`
[5] Github Link `https://github.com/testrepo007/AggrePlay`

Table 3: Experimental Results on AggrePlay (AP), Light (LI), and Stride (ST). The column BT represents the Base runtime slowdown presented in seconds. Columns A, B and C represent the slowdown runtime slowdown factors for Stride, AggrePlay and Light respectively.

Benchmark	Application Domain	Log Size (MB)			BT	Slowdown factor				
		AggrePlay	*Light*	*Stride*		*A*	*B*	*C*	B/A	C/A
cholesky	HPC	25.30	18.00	130	5.92	25.41	16.60	18.26	0.63	0.72
fft	Signal Processing	15.10	11.30	259.90	2.53	52.48	45.70	125.65	0.87	2.39
fmm	HPC	19.02	12.02	160.24	3.71	40.4	34.20	45.25	0.84	1.12
lu_cb	HPC	17.37	11.37	9.80	21.84	3.74	4.54	2.18	1.21	0.58
lu_ncb	HPC	14.60	9.60	8.70	21.36	3.90	4.60	2.20	1.17	0.56
ocean_cp	HPC	52.30	41.70	215.10	4.53	53.12	45.21	35.8	0.85	0.67
ocean_ncp	HPC	76.50	53.50	292.62	5.7	62.5	39.5	42.36	0.63	0.68
raytrace	Graphics	82.60	68.40	432.48	6.58	23.1	20.50	35.2	0.89	1.52
radiosity	Graphics	40.23	32.35	290.36	5.74	36.81	39.64	41.1	1.07	1.12
radix	General	31.39	24.25	253.12	2.9	43.1	35.70	118.26	0.82	2.74
volrend	Graphics	17.20	14.20	249.40	2.63	43.65	40.60	120.4	0.93	2.76
water_nsquared	HPC	18.00	11.10	143.70	2.2	20.2	18.80	98.97	0.93	4.9
water_spatial	HPC	18.80	11.60	170.90	4.77	31.6	26.40	74.5	0.83	2.36
barnes	HPC	172.40	120.40	864.58	5.63	46.82	41.70	108.2	0.89	2.31
mysql	Database	0.65	0.53	2.10	6.25	1.45	1.12	1.35	0.77	0.93
apache	webserver	1.20	0.90	3.50	5.7	3.1	3.30	4.6	1.06	1.48
Test Chain	Blockchain simulation	152.00	135.10	935.12	287.23	8.9	9.50	12.4	1.07	1.39
	Mean	**44.39**	**33.90**	**260.10**	**23.25**	**29.43**	**25.16**	**52.16**	**0.90**	**1.66**

55% reduction in log size for over AggrePlay for two benchmarks *lu_cb* and *lu_ncb,* two kernel-based applications**.** A contributing factor to this result is the optimization of Stride's *last one value predictor* [4, 5] for write-read candidate pairs which enabled Stride to compress a number of read events if they returned the same value. AggrePlay maintained read count vector clocks for each thread and outputs *read-write* events to logs.

Stride did not log read-write relations, it had some advantage in generating smaller log sizes over AggrePlay. Light generated a 30% lower log size on average compared to AggrePlay. Light achieved smaller log sizes by recording only *write-write* dependencies.

The seventh, eighth and ninth columns show the runtime slowdown results. AggrePlay had better slowdown on 12 out of 17 benchmarks. On average, AggrePlay incurred a 66% improvement in slowdown over Stride. In the recording phase, Light also achieved only a 1.2x speedup over AggrePlay. This is because AggrePlay monitored more events than Light did.

5.7 Thread Interleaving Results

During the replay phase, We followed [10] in reading log files as input. AggrePlay retrieves *read-write* interleavings by comparing the read count VC values of the executing thread against the logged read vector values (Line 46 of Algorithm 2). The complexity of determining a *read-write* interleaving is *O(n),* where *n* represents the number of threads in the subject program. Table 4 shows the comparison on *write-read* pairs between all 3 replay techniques.

Recall that AggrePlay logged write-read interleavings (as well as other interleaving data as stated in Algorithm 1). During this process, AggrePlay achieved small logs by applying three optimization functions (See Section 5.9**).** Stride records write-read interleaving candidate pairs and read values to infer *write-read* interleavings during each replay execution. This strategy can increase parallelism in the record phase.

Despite the space optimization used by Stride, AggrePlay achieved similar *write-write* event numbers.

AggrePlay achieved relatively small read-write logs due to the read count vector clock strategy. Also, AggrePlay is the only technique among the three to record read-write events.

5.8 Replay Phase Results

Table 5 presents the experimental results on replay phases for AggrePlay, Light and Stride. A successful replay run for AggrePlay in our experiment was the reproduction of interleavings observed in the record phase. Each subject program is run 50 times. AggrePlay outperforms Stride in successfully reproducing thread interleavings in all benchmarks with an average probability of 86.78% compared to 45.22% for Stride. However, Stride does not guarantee to reproduce all interleavings.

Recall that Light employs a constraint solver in generating possible schedules for replay. The constraint solver iteratively and progressively searches for candidate schedules which satisfy the given constraints. However, in our experiment we found the constraint solver to be incapable of generating candidate schedules mainly due to the high number of constraints generated for our benchmarks. With the exception of *mysql* (26 possible traces) and *apache* (322 traces), the constraint solver did not return any output even after several hours of constraint solving. For *mysql* and

Table 4: Comparison on thread interleaving sets recorded by AggrePlay (AP), Light (LI), and Stride (ST).

Benchmark	write-read			write-write			read-write		
	AP	LI	ST	AP	LI	ST	AP	LI	ST
cholesky	16,835,223	5,244,680	211,316,561	116,454,698	-	116,455,141	649,168	-	-
fft	37,755,408	20,354,645	1,179,517,629	1,589,040,873	-	1,630,086,251	802,974	-	-
fmm	86,480,840	59,944,785	4,992,290,167	802,102,093	-	802,509,215	498,204	-	-
lu_cb	1,147,123,333	455,176,574	420,449,550	2,906,631,297	-	2,918,515,356	1,200,549	-	-
lu_ncb	1,118,695,121	582,823,435	414,739,444	3,022,418,803	-	3,000,325,289	1,410,328	-	-
ocean_cp	23,393,772	626,687,427	3,092,604,375	1,559,238,042	-	1,588,397,241	740,572	-	-
ocean_ncp	22,176,858	522,651,502	3,035,684,527	1,309,305,136	-	1,320,918,835	740,904	-	-
raytrace	1,755,124	7,852,886,222	8,497,788,518	1,002,209,716	-	1,023,168,563	974,021	-	-
radiosity	3,072,561,026	1,877,512,232	9,542,258,648	3,980,326,223	-	4,195,687,007	2,093,562	-	-
radix	169,934,127	123,660,735	688,189,344	638,572,953	-	671,870,742	500,023	-	-
volrend	167,140,775	104,818,614	2,244,606,453	861,000,945	-	860,001,217	508,693	-	-
water_nsquared	2,650,563,143	1,896,254,028	12,309,100,041	10,0452,536,823	-	10,197,195,505	1,670,426	-	-
water_spatial	138,149,919	241,985,885	3,946,594,383	3,030,847,358	-	3,011,361,640	608,759	-	-
barnes	847,372,502	379,742,097	9,778,880,554	5,972,324,,278	-	6,467,986,600	800,173	-	-
mysql	426	352	2,399,719	15,087,675	-	19,387,551	873	-	-
apache	6,123	5,235	6,389,212	4,802,304	-	4,982,341	2,498	-	-
Test Chain	95,275,694,122	62,921,563,148	379,170,163,729	265,504,242,072	-	269,831,003,128	200,333,522	-	-

Table 5: Replay Results on AggrePlay (AP), Light (LI), and Stride (ST).The second column COE (Correct Output Executions) represents number of executions with successfully reproduced output. The third and fourth columns CIE (Correct Interleaving Executions) show the number of successfully reproduced interleavings. (-) represents no results for the benchmark under that column. (*) represents worse-performing runtimes for AggrePlay.

Benchmark	# COE	# CIE			Mean Slowdown factor		
	ST	*AP*	*LI*	*ST*	*AP*	*LI*	*ST*
cholesky	36	48	-	21	2.1	-	3.2
fft	28	43	-	20	3.2	-	3.65
fmm	26	40	-	18	2.7*	-	2.3
lu_cb	27	41	-	15	2.1	-	2.9
lu_ncb	25	44	-	19	2.64	-	3.3
ocean_cp	36	41	-	26	3.15	-	4.72
ocean_ncp	32	39	-	24	3.18*	-	2.8
raytrace	29	37	-	19	3.35	-	3.73
radiosity	32	42	-	23	2.2	-	2.9
radix	32	40	-	22	3.5	-	4.28
volrend	24	42	-	18	1.7	-	2.4
water_nsquared	36	48	-	24	1.9	-	2.85
water_spatial	37	45	-	21	1.85	-	2.3
barnes	31	39	-	17	4.8	-	5.2
mysql	40	48	40	31	4	5.4	5.5
apache	40	46	38	28	2.4	3.5	3.9
Test Chain	30	48	-	25	4.9	-	5.25
Mean	32.56	43.39	39	22.61	2.85	4.45	3.54

apache, Light succeeded in reproducing correct interleavings 78% of the time.

The second column in Table 5 shows the number of executions with successfully reproduced outputs by Stride. Stride produced output of benchmarks with an average probability of 65.12%. In the case of Stride, unexpected program halts due to its replay strategy counted as unsuccessful runs. The mean slowdown factor for each technique on all the benchmarks is shown in the last two columns of Table 5. For *mysql* and *apache*, AggrePlay reproduced all interleavings with an average probability of 94% compared to 80% for Stride. AggrePlay incurs the least amount of slowdown among the three techniques except for *fmm*.

5.9 AggrePlay Optimization Functions

To reduce the number of entries in the record log, the implementation of AggrePlay includes three optimization functions in the record phase as follows:

O1: *If the read count vector clock for the current thread remains unmodified during the updateReadVector function, the event is not recorded.*

O2: *If there is a change in the read count vector clock for the current thread, and the only modified value is that of the current executing thread, the event is not recorded.*

O3: *If more than one thread has the same read count value, the values are recorded in the format* ***c,d*** *where* ***c*** *refers to the read count value and* ***d*** *refers to its frequency in the read vector.*

For **O1** and **O2**, we modify Line 24 of Algorithm 1 with as follows: if a condition *RC_changed* (evaluates to *true* when a thread's RC vector clock is modified, and evaluates to *false* otherwise) evaluates to *true*, then $EX_{\text{Tid}(e)} := EX_{\text{Tid}(e)} \hat{} \langle RC_{Tid(e)}, e \rangle$. If *RC_changed* evaluates to *false* then a null line Θ is appended to $EX_{\text{Tid}(e)}$ *as* $EX_{\text{Tid}(e)} := EX_{\text{Tid}(e)} \hat{} \Theta$;

In algorithm 2, lines 41-42 show how the records missed by **O1** and **O2** are recovered. If the first element of EX_t matches a null line (Θ), it means the last read event to the shared memory location prior to the current write was executed by the currently executing thread. As such, the first element of EX_t is removed and the *checkReads* function returns *true*. For **O3** in algorithm 2, the init(EX_t) function recovers all $\langle c, d \rangle$ pairs recorded and resolves each pair to a RC_t object.

5.10 Limitations

Our implementations of all 3 replay techniques suffer the constraints of any tool created using PIN, which serializes the instrumented events from the subject program to a pintool regardless of support for multi-threading. AggrePlay may not deterministically reproduce interleavings which involve worker threads spawned non-deterministically by long-running applications like apache or mysqlas the workload increases during the replay phase.

6 RELATED WORK

Some replay techniques aim to reproduce a target output (including particular program state) only. ODR [1] takes a core dump as input, extracts some program state values from the latter, and generates a trace to reproduce these values at target code locations through a search process. Bbr [7] needs a predefined set of location checkpoints to reduce log sizes and uses a search process with symbolic execution in constructing feasible traces passing through these checkpoints. AggrePlay does not rely on symbolic executions to construct target traces. Rather, AggrePlay records the counts on the numbers of thread-local read accesses at the execution point of handling the write instructions. AggrePlay can be further augmented with a technique [5] that records the non-deterministic inputs in the record phase and compares the corresponding values in the replay run against these logged inputs.

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 solving approach to find thread schedules having these interleavings whilst constructing replay traces. AggrePlay also records write-read interleavings explicitly but uses no constraint solver. Keeping thread-local data for efficient recording has also been explored by existing work such as CLAP [11]. CLAP relies on constraint solving in an offline phase to symbolically analyze expression values, which is proven by our experiment with Light to be unscalable, and "has limitations in handling complex arithmetic computations in practice." CARE [12] 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. Maintaining such caches incur high memory overheads. CARE keeps each read-write interleaving in the record phase. AggrePlay does not need constraint solving nor have an offline phase and records a read count vector for read-write dependencies.

Some techniques combines record and replay into one phase. DoubleTake [18] divides an exaction into epochs by the locations of irrevocable system calls. It iteratively logs the system state right before the epoch, executes the instructions in the epoch, analyzes the program state for derivations and replays that fragment if derivations are found; otherwise it proceeds to reply the next epoch. DoubleTake requires special hardware to support. Unlike MobiPlay [24], AggrePlay does not need to modify the underlying framework to support its replay.

Checkpointing is frequently used in replay techniques. iReplayer [16] stores the system state in memory and allows users to specify checkpointing rules. It monitors data races in its replay phase and iteratively executes an epoch (in a sense similar to DoubleTake) if the replay schedule for the involving interleavings differs from the required thread interleavings. AggrePlay does not use checkpointing.

Processor-oblivious record and replay [28] has been proposed for data-race free systems such as Cilk programs. This replay strategy focuses on recording the synchronization order for programs which employ task-parallelism. Such programs do not have any notion of threads or data. However, current mainstream software supports multi-threaded parallelism. AggrePlay can be applied to most mainstream software.

7 CONCLUSION

We have presented AggrePlay, a deterministic replay technique which is based on recording read-write interleavings leveraging thread-local determinism and summarized read values. During the record phase, AggrePlay records a read count vector clock for each read on each shared memory location. In the replay phase, each thread matches the logged read count against each executing read event to ensure a target number of read events prior to the next write. We have presented an experiment and analyzed the results of our experiment using the Splash2x benchmark suite, apache and mysql, and a blockchain implementation. The experimental results indicated that on average, AggrePlay experiences better reduction in compressed log size, and 56% better runtime slowdown during the record phase, as well as a 41.58% higher probability in the replay phase than an existing technique.

REFERENCES

[1] Altekar, G., and Stoica, I. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 193–206.

[2] Bacon, D. F., and Goldstein, S. C. *Hardware-assisted replay of multiprocessor programs*, vol. 26. ACM, 1991.

[3] Bienia, C., Kumar, S., and Li, K. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (2008), IEEE, pp. 47–56.

[4] Burckhardt, S., Dern, C., Musuvathi, M., and Tan, R. Line-up: A complete and automatic linearizability checker. *SIGPLAN Not. 45*, 6 (June 2010), 330–340.

[5] Burtscher, M., and Zorn, B. G. Exploring last n value prediction. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)* (1999), pp. 66–76.

[6] Chen, Y., Zhang, S., Guo, Q., Li, L., Wu, R., and Chen, T. Deterministic replay: A survey. *ACM Comput. Surv. 48*, 2 (Sept. 2015), 17:1–17:47.

[7] Cheung, A., Solar-Lezama, A., and Madden, S. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), ACM, pp. 135–145.

[8] Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., and Bosschere, K. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet* (2003).

[9] De Moura, L., and Bjørner, N. 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* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[10] Huang, J., Liu, P., and Zhang, C. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 207–216.

[11] Huang, J., Zhang, C., and Dolby, J. Clap: Recording local executions to reproduce concurrency failures. *SIGPLAN Not. 48*, 6 (June 2013), 141–152.

[12] Jiang, Y., Gu, T., Xu, C., Ma, X., and Lu, J. Care: Cache guided deterministic replay for concurrent java programs. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 457–467.

[13] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[14] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[15] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36*, 4 (Apr. 1987), 471–482.

[16] Liu, H., Silvestro, S., Wang, W., Tian, C., and Liu, T. 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* (New York, NY, USA, 2018), PLDI 2018, ACM, pp. 344–358.

[17] Liu, P., Zhang, X., Tripp, O., and Zheng, Y. Light: Replay via tightly bounded recording. *SIGPLAN Not. 50*, 6 (June 2015), 55–64.

[18] Liu, T., Curtsinger, C., and Berger, E. D. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 911–922.

[19] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not. 40*, 6 (June 2005), 190–200.

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

[21] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 73–84.

[22] Narayanasamy, S., Pokam, G., and Calder, B. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News 33*, 2 (May 2005), 284–295.

[23] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K. H., and Lu, S. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 177–192.

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

[25] Ren, S., Tan, L., Li, C., Xiao, Z., and Song, W. Leveraging hardware-assisted virtualization for deterministic replay on commodity multi-core processors. *IEEE Transactions on Computers 67*, 1 (Jan 2018), 45–58.

[26] Saito, Y. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging* (New York, NY, USA, 2005), AADEBUG'05, ACM, pp. 69–76.

[27] Steven, J., Chandra, P., Fleck, B., and Podgurski, A. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2000), ISSTA '00, ACM, pp. 158–167.

[28] Utterback, R., Agrawal, K., Lee, I.-T. A., and Kulkarni, M. Processor-oblivious record and replay. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2017), PPoPP '17, ACM, pp. 145–161.

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

[30] Yang, Z., Yang, M., Xu, L., Chen, H., and Zang, B. Order: Object centric deterministic replay for java. In *USENIX Annual Technical Conference* (2011), pp. 30–30.

[31] Zhou, J., Xiao, X., and Zhang, C. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 892–902.