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.

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† [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

† This research is supported in part by the GRF of HKSAR Research Grants Council (project nos. 11214116 and 1120015), the HKSAR ITF (project no. ITS/378/18), and the CityU MF_EXT (project no. 9678180).
‡ Correspondence Author
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 $s_1=(w_1(x), r_1(x), w_2(x))$. There are two data races $race_1$ and $race_2$ on the interleaving $r_1(x) \rightarrow w_1(x)$ and $r_1(x) \rightarrow w_2(x)$ respectively. Assuming both writes are protected by a lock, Light records the write-read dependency from the trace $s_1$ as $w_1(x) \rightarrow r_1(x)$. Light will reproduce $race_2$ but will miss the dependency $r_1(x) \rightarrow w_2(x)$ 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-deterministic**.

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 | \( \sigma \) \( \equiv \) \( w(x) \mid r(x) \mid \text{acq}(m) \mid \text{rel}(m) \mid \text{fork}(u) \mid \text{join}(u) \) \( x \in \text{Memory Location}; m \in \text{Lock}; u \in \text{Thread}; \) |
| Event     | \( e \) \( \equiv \) \( (t, op) , t\in \text{Thread}; op \in \text{Operation} \) |
| Execution trace | \( \sigma \equiv (e_1, e_2, e_3, \ldots, e_n), e_i \in \text{Event} \) |

2.1 Execution Trace
An execution trace \( \sigma \equiv (e_1, e_2, \ldots, e_n) \) is a sequence of operations observed from the execution of a software program. An operation \( e \) 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 \( x \).
- \( t.\text{acq}(m) \): A lock acquisition instruction executed by thread \( t \) on lock \( m \).
- \( t.\text{rel}(m) \): A lock release instruction by thread \( t \) on lock \( m \).
- \( t.\text{fork}(u) \): Thread \( t \) forks another thread \( u \).
- \( t.\text{join}(u) \): Thread \( t \) joins another thread \( u \).

Other synchronization primitives such as \( \text{wait} \), \( \text{signal} \), and \( \text{barrier} \) 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 \( RC(t) \), 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 \( \sigma_2 = (e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9) \) is produced. The set \( \{e_1 \sim e_3, e_3 \sim e_4, e_2 \sim e_5, e_5 \sim e_7, e_6 \sim e_8, e_8 \sim e_9, e_7 \sim e_9\} \) is produced as thread interleavings.

To capture the trace \( \sigma_2 \), 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 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 \( \sigma_2 \), Light records the write-read interleavings \( \{e_3 \sim e_4, e_6 \sim e_7, e_6 \sim e_8\} \) 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 \((\text{value}_e)\) of each read operation with a possible matching write version \((\text{version}_e)\).

For trace \( \sigma_2 \), Stride records the write-write interleavings \( e_1 \sim e_3 \) and \( e_6 \sim e_8 \). The candidate write-read pairs \((\text{value}_e, \text{version}_e)\), \((\text{value}_e, \text{version}_e)\) and \((\text{value}_e, \text{version}_e)\) are recorded during a write operation in the replay phase, the write-read pairs are used to infer possible interleaving candidates with \(\text{version}_e\) being the upper and \(\text{value}_e\) being the matching criterion.

For the trace \( \sigma_3 \) in Figure 3, \( t_2 \) executes two writes, \( e_5 \) and \( e_6 \) after \( t_1 \) executes \( e_7 \). Suppose that \( e_5 \) and \( e_6 \) write the value 1, but \( e_7 \) writes the value 0. In this case, Stride may record the pair \((\text{value}_e, \text{version}_e)\) because write-read interleavings are not ordered in the record phase. During the replay phase, the interleaving \( e_6 \sim e_7 \) will be missed by Stride because the
As such, it can keep track of all events in both traces \( \sigma \). In this section, we present AggrePlay. The following notations are optimizations as well as read-write incorrectly ordered. events \( e \) and \( v \) are not explicitly ordered by Light and may be incorrectly ordered.

AggrePlay records write-write and write-read interleaveings with optimizations as well as read-write interleaveings in a novel strategy. As such, it can keep track of all events in both traces \( \sigma_2 \) and \( \sigma_3 \). 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 \( e \).
- \( RC_t \): The lock acquisition list for lock \( m \).
- \( var(e) \): 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.
- \( t\cdot\text{yield}() \): The executing thread \( t \) waits for other threads to advance without blocking.

- \( \text{top}(\text{Input}) \): reads data from the first index of \( \text{Input} \). \( \text{Input} = W_x \mid L_m \mid EX \mid WR_x \).
- \( \text{pop}(\text{Input}) \): removes data from first index position of \( \text{Input} \).

#### 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 interleaveings. 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 interleaveings 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 \( \sigma := (e_1, e_2, e_3, \ldots, e_n), e \in \text{Event} \)

**Output:** Execution trace \( \sigma := (e_1, e_2, e_3, \ldots, e_n), e \in \text{Event} \)

1. for each \( t, e \in \text{Thread} \), Event \{ 
   2. \( \text{init}(RC_{t,var(e)}); \)
   3. \}
2. // write access performed by event \( e \)
3. **onWrite** (Event \( e \)) do
   4. \( \text{Sync} \)
   5. \( \text{updateReadVectors}(e); \)
   6. \( \text{execute}(e); // \text{the write instruction is executed} \)
   7. \( W_{var(e)} := W_{var(e)} \cdot (lw, e); \)
   8. \} 
9. end onWrite
10. // read access returned by event \( e \)
11. **onRead** (Event \( e \)) do
12. \( t := Tid(e); \)
13. **for each** \( t \in \text{Thread} \) and \( RC_t[var(e)] [t'] > 0 \) do
14. \( RC_t[var(e)][t'] := \text{inc}(RC_t[var(e)][t]); \)
15. \} 
16. end for
17. \( \text{updateReadVectors}(e); \)
18. \} 
19. end onRead
20. **updateReadVectors** (Event \( e \)) do
21. \( \text{for each} \ t' \in \text{Thread} \) and \( RC_{t',var(e)} [t'] > 0 \) do
22. \( RC_{t'[var(e)][t']} := RC_{t'[var(e)][t']}; \)
23. \} 
24. end for
25. \} 
26. **end updateReadVectors**
27. **onLockAcquire** (Event \( e \)) do
28. \( \text{execute}(e); \)
29. \} 
30. 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
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 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 \( \sigma_2 \) 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 \( RC_{t_2} [t_2] \) is incremented by 1. The write-read dependency \( e_3 \rightarrow e_4 \) 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 \( RC_{t_2} [t_2] \) is incremented by 1. The write-read dependency \( e_6 \rightarrow e_7 \) is also recorded. Then \( RC_{t_2} [t_2] \) 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 \( RC_{t_1} [0,1,1] \), is constructed before event \( e_9 \) and saved in the record. The record output for the trace is shown in Figure 5.

### 4.2 AggrePlay Replay Phase

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

\[
\begin{align*}
\text{WH}_{\text{var}}(e) = & \{ (\text{var}(e), e) \} \\
\text{WH}_{x}(e) = & \{ (e_1 \rightarrow e_1) \} \\
\text{WH}_{y}(e) = & \{ (e_2 \rightarrow e_2) \} \\
\text{WH}_{z}(e) = & \{ (e_3 \rightarrow e_3) \}
\end{align*}
\]

Figure 5: AggrePlay output data for trace \( \sigma_2 \) in Figure 2

The \( \text{repW}_{\text{var}}(e) \) is the access list for the shared memory location \( \text{var}(e) \) 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 is \( \perp_{RC} \) 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 (\( W_{\text{var}}(e) \)) for that shared memory location \( \text{var}(e) \) 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 \( \text{repW}_{\text{var}}(e) \) is updated with the write event. The first element is removed from the \( W_{\text{var}}(e) \) input list (lines 10-12). Otherwise, if checkReadVector returns false, the thread yields the processor at line 13. Lines 17-27 show the onWrite 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 \( WR_{\text{var}}(e) \) 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 \( WR_{\text{var}}(e) \) input list is removed (lines 22-23). Otherwise, if the condition at line 19 is evaluated to false, 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 \( L_{\text{Lock}}(e) \), then the thread yields (lines 29-30). Otherwise, the lock acquisition event is executed and the first element of the input list \( L_{\text{Lock}}(e) \) 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 (\( \Theta \)) in the replay phase. If the first element of \( EX_i \) 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 \( EX_i \) 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 \( EX_i \) input list. The function then returns true at line 51.

AggrePlay replays the execution in Figure 2 as follows: \( t_5 \) 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
We evaluated our implementation using the Splash2x extension of PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

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 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 [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.

---

5.2 Benchmarks

We evaluated our implementation using the Splash2x extension of the PARSEC 3.1 benchmark suite [5], specifically barnes, ocean_cp, radiosity, raytrace, volrend, water_spatial, fmm, water_n_squared, water_spatial as well as kernel applications cholesky, fft, lu_c, lu_nb, 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 to test the robustness of our tool for emerging software technology.
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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th># of Events</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write</td>
<td>Read</td>
<td>Locks</td>
</tr>
<tr>
<td>cholesky</td>
<td>$1.16 \times 10^7$</td>
<td>$2.9 \times 10^6$</td>
<td>22,066</td>
</tr>
<tr>
<td>fft</td>
<td>$1.6 \times 10^9$</td>
<td>$3.0 \times 10^8$</td>
<td>66</td>
</tr>
<tr>
<td>fmm</td>
<td>$8.0 \times 10^8$</td>
<td>$6.2 \times 10^7$</td>
<td>$6.3 \times 10^5$</td>
</tr>
<tr>
<td>lu_cb</td>
<td>$2.9 \times 10^9$</td>
<td>$6.0 \times 10^8$</td>
<td>2,082</td>
</tr>
<tr>
<td>lu_tcb</td>
<td>$2.9 \times 10^9$</td>
<td>$6.0 \times 10^8$</td>
<td>2,082</td>
</tr>
<tr>
<td>ocean_ep</td>
<td>$1.6 \times 10^9$</td>
<td>$5.6 \times 10^8$</td>
<td>6,666</td>
</tr>
<tr>
<td>oceanhcp</td>
<td>$2.1 \times 10^9$</td>
<td>$5.6 \times 10^8$</td>
<td>6,506</td>
</tr>
<tr>
<td>raytrace</td>
<td>$1.0 \times 10^9$</td>
<td>$15.2 \times 10^7$</td>
<td>985,599</td>
</tr>
<tr>
<td>radiosity</td>
<td>$1.2 \times 10^9$</td>
<td>$1.3 \times 10^8$</td>
<td>481,866</td>
</tr>
<tr>
<td>radix</td>
<td>$6.7 \times 10^8$</td>
<td>$1.1 \times 10^7$</td>
<td>217</td>
</tr>
<tr>
<td>volrend</td>
<td>$8.6 \times 10^8$</td>
<td>$3.5 \times 10^7$</td>
<td>705,342</td>
</tr>
<tr>
<td>water_nrounded</td>
<td>$10.2 \times 10^9$</td>
<td>$26.0 \times 10^7$</td>
<td>160,298</td>
</tr>
<tr>
<td>water_spatial</td>
<td>$3.0 \times 10^9$</td>
<td>$8.8 \times 10^8$</td>
<td>315</td>
</tr>
<tr>
<td>barnes</td>
<td>$6.5 \times 10^9$</td>
<td>$11.5 \times 10^8$</td>
<td>$1.0 \times 10^8$</td>
</tr>
<tr>
<td>mysql</td>
<td>$5.1 \times 10^9$</td>
<td>$1.6 \times 10^8$</td>
<td>21</td>
</tr>
<tr>
<td>apache</td>
<td>$5.8 \times 10^5$</td>
<td>$6.5 \times 10^5$</td>
<td>6</td>
</tr>
<tr>
<td>Test Chain</td>
<td>$4.17 \times 10^{11}$</td>
<td>$8.14 \times 10^{11}$</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $(a, b)$ where $a$ represents the thread id and $b$ refers to the value for this 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 $(RC, write)$ 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

\[573\]
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.

### 5.8 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 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application Domain</th>
<th>Log Size (MB)</th>
<th>BT</th>
<th>Slowdown factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>AggrePlay</strong></td>
<td><strong>Light</strong></td>
<td><strong>Stride</strong></td>
</tr>
<tr>
<td>cholesky</td>
<td>HPC</td>
<td>25.30</td>
<td>18.00</td>
<td>130</td>
</tr>
<tr>
<td>fft</td>
<td>Signal Processing</td>
<td>15.10</td>
<td>11.30</td>
<td>259.90</td>
</tr>
<tr>
<td>ffm</td>
<td>HPC</td>
<td>19.02</td>
<td>12.02</td>
<td>160.24</td>
</tr>
<tr>
<td>lu cb</td>
<td>HPC</td>
<td>17.37</td>
<td>11.37</td>
<td>9.80</td>
</tr>
<tr>
<td>lu ncb</td>
<td>HPC</td>
<td>14.60</td>
<td>9.60</td>
<td>8.70</td>
</tr>
<tr>
<td>ocean cp</td>
<td>HPC</td>
<td>52.30</td>
<td>41.70</td>
<td>215.10</td>
</tr>
<tr>
<td>ocean ncp</td>
<td>HPC</td>
<td>76.50</td>
<td>53.50</td>
<td>292.62</td>
</tr>
<tr>
<td>raytrace</td>
<td>Graphics</td>
<td>82.60</td>
<td>68.40</td>
<td>432.48</td>
</tr>
<tr>
<td>radiosity</td>
<td>Graphics</td>
<td>40.23</td>
<td>32.35</td>
<td>290.36</td>
</tr>
<tr>
<td>radix</td>
<td>General</td>
<td>31.39</td>
<td>24.25</td>
<td>253.12</td>
</tr>
<tr>
<td>volrend</td>
<td>Graphics</td>
<td>17.20</td>
<td>14.20</td>
<td>249.40</td>
</tr>
<tr>
<td>water nsquared</td>
<td>HPC</td>
<td>18.00</td>
<td>11.10</td>
<td>143.70</td>
</tr>
<tr>
<td>water spatial</td>
<td>HPC</td>
<td>18.80</td>
<td>11.60</td>
<td>170.90</td>
</tr>
<tr>
<td>barnes</td>
<td>HPC</td>
<td>172.40</td>
<td>120.40</td>
<td>864.58</td>
</tr>
<tr>
<td>mysql</td>
<td>Database</td>
<td>0.65</td>
<td>0.53</td>
<td>2.10</td>
</tr>
<tr>
<td>apache</td>
<td>webservew</td>
<td>1.20</td>
<td>0.90</td>
<td>3.50</td>
</tr>
<tr>
<td>Test Chain</td>
<td>Blockchain simulation</td>
<td>152.00</td>
<td>135.10</td>
<td>935.12</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td></td>
<td><strong>44.39</strong></td>
<td><strong>33.90</strong></td>
<td><strong>260.10</strong></td>
</tr>
</tbody>
</table>
Table 4: Comparison on thread interleaving sets recorded by AggrePlay (AP), Light (LI), and Stride (ST).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AP</th>
<th>LI</th>
<th>ST</th>
<th>AP</th>
<th>LI</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write-read</td>
<td></td>
<td></td>
<td>write-read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cholesky</td>
<td>16,835,223</td>
<td>5,244,680</td>
<td>211,316,561</td>
<td>116,454,698</td>
<td>-</td>
<td>116,455,141</td>
</tr>
<tr>
<td>fft</td>
<td>37,755,408</td>
<td>20,354,645</td>
<td>1,719,517,629</td>
<td>1,589,940,873</td>
<td>-</td>
<td>1,630,086,251</td>
</tr>
<tr>
<td>fnm</td>
<td>86,480,840</td>
<td>59,944,785</td>
<td>4,992,290,167</td>
<td>802,102,093</td>
<td>-</td>
<td>802,509,215</td>
</tr>
<tr>
<td>lu_cb</td>
<td>1,147,123,333</td>
<td>453,176,574</td>
<td>420,449,530</td>
<td>2,906,631,297</td>
<td>-</td>
<td>2,918,351,356</td>
</tr>
<tr>
<td>lu_ncb</td>
<td>1,118,695,121</td>
<td>582,823,435</td>
<td>414,739,444</td>
<td>3,022,418,803</td>
<td>-</td>
<td>3,000,325,289</td>
</tr>
<tr>
<td>ocean_cp</td>
<td>23,393,772</td>
<td>626,687,427</td>
<td>3,092,664,375</td>
<td>1,559,238,042</td>
<td>-</td>
<td>1,588,397,241</td>
</tr>
<tr>
<td>ocean_ncp</td>
<td>22,176,838</td>
<td>522,651,502</td>
<td>3,035,684,527</td>
<td>1,309,305,136</td>
<td>-</td>
<td>1,329,918,835</td>
</tr>
<tr>
<td>raytrace</td>
<td>1,755,124</td>
<td>7,852,866,222</td>
<td>8,497,788,518</td>
<td>1,002,209,716</td>
<td>-</td>
<td>1,023,168,563</td>
</tr>
<tr>
<td>radiosity</td>
<td>3,072,561,026</td>
<td>1,877,512,232</td>
<td>9,542,258,648</td>
<td>3,980,326,223</td>
<td>-</td>
<td>4,195,687,007</td>
</tr>
<tr>
<td>radix</td>
<td>169,934,127</td>
<td>123,660,735</td>
<td>68,189,344</td>
<td>638,572,953</td>
<td>-</td>
<td>671,870,742</td>
</tr>
<tr>
<td>volrend</td>
<td>167,140,775</td>
<td>104,818,614</td>
<td>2,244,606,453</td>
<td>861,000,945</td>
<td>-</td>
<td>860,001,217</td>
</tr>
<tr>
<td>water_rnsquared</td>
<td>2,650,563,143</td>
<td>1,896,254,028</td>
<td>12,309,100,041</td>
<td>10,045,536,823</td>
<td>-</td>
<td>10,197,195,505</td>
</tr>
<tr>
<td>barnes</td>
<td>847,372,502</td>
<td>379,742,097</td>
<td>9,778,880,554</td>
<td>5,972,324,278</td>
<td>-</td>
<td>6,467,986,600</td>
</tr>
<tr>
<td>mysql</td>
<td>426</td>
<td>352</td>
<td>2,399,719</td>
<td>15,087,657</td>
<td>-</td>
<td>19,387,551</td>
</tr>
<tr>
<td>apache</td>
<td>6,123</td>
<td>5,235</td>
<td>6,389,212</td>
<td>4,802,304</td>
<td>-</td>
<td>4,982,341</td>
</tr>
<tr>
<td>Test Chain</td>
<td>95,275,694,122</td>
<td>62,921,563,148</td>
<td>379,170,163,729</td>
<td>265,504,242,072</td>
<td>-</td>
<td>269,831,003,128</td>
</tr>
</tbody>
</table>

| Mean       | 32.56       | 43.39       | 39           | 22.61       | 2.85       | 4.45       |

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># COE</th>
<th># CIE</th>
<th>Mean Slowdown factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ST</td>
<td>AP</td>
<td>LI</td>
</tr>
<tr>
<td>cholesky</td>
<td>36</td>
<td>48</td>
<td>-</td>
</tr>
<tr>
<td>fft</td>
<td>28</td>
<td>43</td>
<td>-</td>
</tr>
<tr>
<td>fnm</td>
<td>26</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td>lu_cb</td>
<td>27</td>
<td>41</td>
<td>-</td>
</tr>
<tr>
<td>lu_ncb</td>
<td>25</td>
<td>44</td>
<td>-</td>
</tr>
<tr>
<td>ocean_cp</td>
<td>36</td>
<td>41</td>
<td>-</td>
</tr>
<tr>
<td>ocean_ncp</td>
<td>32</td>
<td>39</td>
<td>-</td>
</tr>
<tr>
<td>raytrace</td>
<td>29</td>
<td>37</td>
<td>-</td>
</tr>
<tr>
<td>radiosity</td>
<td>32</td>
<td>42</td>
<td>-</td>
</tr>
<tr>
<td>radix</td>
<td>32</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td>volrend</td>
<td>24</td>
<td>42</td>
<td>-</td>
</tr>
<tr>
<td>water_rnsquared</td>
<td>36</td>
<td>48</td>
<td>-</td>
</tr>
<tr>
<td>water_spatial</td>
<td>37</td>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>barnes</td>
<td>31</td>
<td>39</td>
<td>-</td>
</tr>
<tr>
<td>mysql</td>
<td>40</td>
<td>48</td>
<td>40</td>
</tr>
<tr>
<td>apache</td>
<td>40</td>
<td>46</td>
<td>38</td>
</tr>
<tr>
<td>Test Chain</td>
<td>30</td>
<td>48</td>
<td>-</td>
</tr>
</tbody>
</table>

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 fnm.
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 evaluate to false otherwise) evaluates to true, then $EX_{Tid(e)} \triangleq EX_{Tid(e)} \cdot \left( RC_{Tid(e)}(e), e \right)$. If $RC_{Tid(e)}$ evaluates to false then a null line $\Theta$ is appended to $EX_{Tid(e)}$.

In algorithm 2, lines 41-42 show how the records missed by O1 and O2 are recovered. If the first element of $EX_1$ matches a null line ($\Theta$), 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 $EX_1$ is removed and the checkReads function returns true. For O3 in algorithm 2, the init($EX_1$) function recovers all pairs recorded and resolves each pair to a $RC_{Tid}$ 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 mysql as 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 unscaleable, 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 execution 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


