Stride: Search-Based Deterministic Replay in Polynomial Time via Bounded Linkage

Jinguo Zhou Xiao Xiao Charles Zhang
The Prism Research Group
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{andyzhou,richardxx,charlesz}@cse.ust.hk

***Abstract*—Deterministic replay remains as one of the most effective ways to comprehend concurrent bugs. Existing approaches either maintain the *exact* shared read-write linkages with a large runtime overhead or use exponential off-line algorithms to search for a feasible interleaved execution. In this paper, we propose Stride, a hybrid solution that records the *bounded* shared memory access linkages at runtime and *infers* an equivalent interleaving in polynomial time, under the sequential consistency assumption. The recording scheme eliminates the need for synchronizing the shared read operations, which results in a significant overhead reduction. Comparing to the previous state-of-the-art approach of deterministic replay, Stride reduces, on average, 2.5 times of runtime overhead and produces, on average, 3.88 times smaller logs.**

***Keywords*-Concurrency; Replaying; Debugging**

I. INTRODUCTION

Deterministically replaying a concurrent multicore execution remains as one of the most effective ways to comprehend concurrency bugs([1]–[5]). A typical *deterministic* replayer must tame two sources of non-determinism: the *input non-determinism*, observing the randomness in the program input such as the user input, interrupts, signals, and the *scheduling non-determinism*, concerned with races to the shared memory locations caused by a random scheduler. While the input non-determinism can be effectively recorded with a low overhead([6], [7]), the scheduling non-determinism still poses tough challenges to making a record and replay technique attractive for the practical use.

Existing replay schemes that address memory races fall into two categories: *order-based* and *search-based*. For the order-based ones, we have come to know, in both theory [8] and practice([6], [9]–[15]), that tracking which write a read follows (the exact linkage), with respect to a particular shared memory location, can be used to efficiently reconstruct an equivalent interleaving, under the sequential consistency criterion [16]. A key drawback is that tracking the exact linkages requires adding additional locks to the program to ensure the recording operation and the observed read/write operations of the program happen together atomically, as illustrated in Figure 1(a). Consequently, recent deterministic replay techniques, such as Leap [9] and Order [15], essentially eliminate all low-level data races in a program, including many benign ones [17], and incur a significant runtime overhead. For Java programs on multi-processors, synchronization can significantly degrade the program performance for causing the chip-wide cache validation operations across all processors [18].

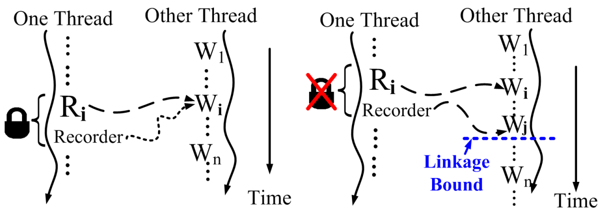
Recognizing this drawback, the search-based replaying techniques([7], [19]–[23]) do not record the exact RW-linkage and, instead, rely on the post-recording search to construct a feasible interleaving. The search-based replay techniques can incur a very low recording overhead[1] at the cost of losing the replay determinism. Gibbons *et al.* [8] proved that computing a feasible schedule with the value trace is NP-complete even with the help of *local write order* that defines a total order for the write operations to the same memory location. In practice, none of the existing search-based techniques guarantees to reproduce a concurrent multicore execution, essentially because the search space, without the exact linkage information, is exponential and cannot scale to large real systems.

It seems that we are faced with an unfortunate choice between losing the replay determinism and paying a severe performance cost for using synchronization. Towards alleviating this difficulty, we present a novel search-based deterministic replay technique that does not record the exact RW-linkages and yet still reconstructs the schedule in *polynomial time*. The "non-exactness" is a crucial relaxation that, for read operations on shared memory locations, *the recording operation and the read events are not required to happen atomically*. Hence, no synchronization is needed. As illustrated in Figure 1(b), for the read operation R_i, instead of observing its exact corresponding write W_i, our recorder observes a write operation, W_j, that happens sometime later than the matching write W_i. If we version all the write operations, the observed version of W_j can be used as a *linkage bound* in guiding the post-recording search to only focus on the writes of older versions, when reconstructing the original execution.

Compared to the pure order-based approaches, our technique dramatically reduces the need of synchronization. Since no atomic execution is required for reads, we essentially permit the *concurrent read exclusive write (CREW)* semantic where the read operations issued by one processor

---

[1] *E.g.* 1% presented by Lee *et al.* [7], and Weeratunge *et al.* [19] present a totally search based method with nothing recorded at all



(A) Recording Exact Linkage (B) Recording Bounded Linkage

Figure 1. Difference Between Recording Exact Linkage and Bounded Linkage

can happen in parallel with the writes from other processors. In most of the real world programs, the number of read operations is much larger than that of the writes. Our versioning of the writes does require adding locks to unprotected writes. We find that, in well engineered concurrent programs, most of the writes to shared locations are locked by the programmer already, which significantly limits the performance penalty of our technique. Since only a limited number of context switches get into the execution window between the read operation and the recording operation, the distance between the bounded linkage and the exact linkage is small. In fact, our evaluation of real programs shows that, for most of the cases, the two operations are not interleaved by other operations at all and, hence, the search can be done in almost $O(1)$ time in practice.

To the best of our knowledge, the only related approach that deterministically reproduces the interleaving without synchronizing the read operations is proposed as a theoretical possibility by Cantin *et al.* [24]. Their proposal requires the serialization of all the writes in the program by a global lock to establish the *global write order*. Serializing writes across cores incurs a significant slowdown for concurrent programs running many threads. Comparatively, our technique only requires locking writes locally for each shared memory location and incurs a limited penalty to the degree of concurrency.

To evaluate our technique, we have implemented a tool called Stride and used it to replay many large Java programs. Our experiment evaluates many widely cited programs including the Dacapo suite, the Derby database server, the ICE IPC middleware, and the specjbb2005 benchmark. The average recording slowdown incurred by Stride is 2 times for all subject programs and 1 time if we exclude special computationally intensive cases such as Avrora and Lusearch. We compare Stride against both our previous order-based replayer Leap and an implementation of Cantin *et al.*'s approach using the global write order. We show that, on average, Stride is faster than Leap by 2.5 times excluding our best cases, for which the gap can be up to 75 times. Stride is also faster than Cantin *et al.*'s global order approach [24] by 2.5 times on average. For all our subjects, the search time for the interleaving regeneration is negligible for all the subject programs. Also, compared to Leap, the log size of Stride is on average 3.88 times smaller excluding our best cases, which are up to 140 times smaller.

In summary, our contributions are the follows:

1. We present Stride, a *bound-infer-replay* technique to deterministically replay concurrent programs on multi-cores. Stride is the first to record partial runtime information and to infer the deterministic execution in polynomial time.
2. Stride only concerns with the write-write race, a more relaxed race condition that favours a lot of well-engineered concurrent programs.
3. We extensively evaluate our algorithm and show that our new algorithm works well in practice, with the overhead orders of magnitude smaller than the state-of-the-art techniques.

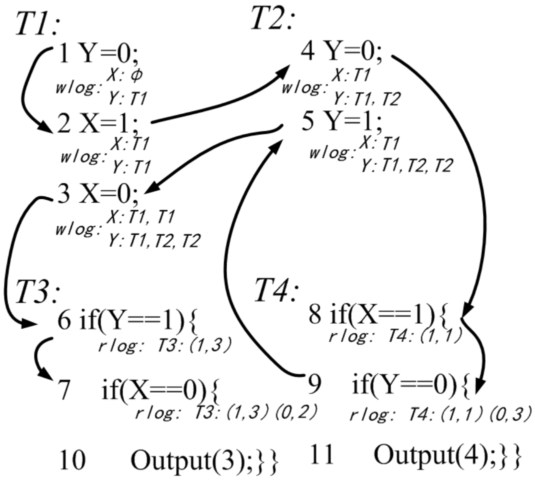
The rest of the paper is organized as follows. Section II provides an exemplified overview of Stride. The formal description and analysis of Stride is given in Section III and IV. In Section V, we discuss how to efficiently implement Stride. The evaluation result is given in Section VI. Finally, we discuss the related work in Section VII and conclude our work in Section VIII.

II. OVERVIEW OF OUR REPLAYING SCHEME

We first illustrate our technique with an example shown in Figure 2. This program has four threads with lines numbered following a total order. We are interested in replaying a special program state where both *output* statements (line 10 and line 11) are executed. The interleaving order, indicated by arrows, is one of the possible schedules to reach this program state. Recall that the order based technique can replay the program to this state by recording the *exact RW-linkages*, which, in the given schedule, include the following: $R6 \rightsquigarrow W5$, $R9 \rightsquigarrow W4$, $R7 \rightsquigarrow W3$, and $R8 \rightsquigarrow W2$. Here R and W stand for read or write operations and RX stands for reading at line X. We want to show that Stride does not record this information and, instead, computes these linkages to replay this program state.

Stride logs the information separately for the read operations, the write operations, and the lock operations. To simplify the example, let us consider only the read and write operations. For the read operations, Stride records a two-tuple representing the value returned by the read operation and the latest version of write that read can possibly link to (the bounded linkage). For example, the tuple (1,2) represents a read of value 1 from a write of version at most 2 for that variable. The read operations are logged separately for each thread. For the write operations, Stride records the thread access order on each variable. In the example in Figure 2, we embed what Stride logs at each statement, where **rlog** and **wlog** denote the logs for read and write operations, respectively.

Figure 3 presents how the Stride replayer uses the two logs to compute the exact linkages listed above. With no



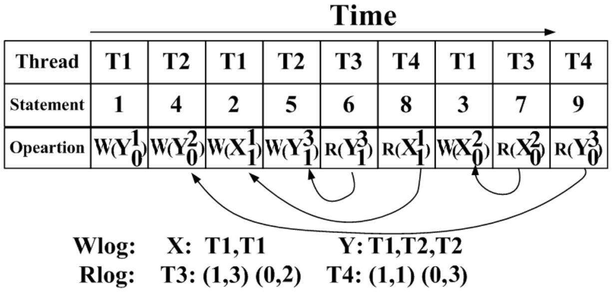
```
T1:                     T2:
1 Y=0;                  4 Y=0;
2 X=1;                  5 Y=1;
3 X=0;

T3:                     T4:
6 if(Y==1){             8 if(X==1){
7   if(X==0){           9   if(Y==0){
10    Output(3);}}      11    Output(4);}}
```

Figure 2. Example program

loss of generality, we assume the replayer uses a round-robin scheduler that executes the next statement selected from the four threads in a rotating fashion starting from the thread T_2. We denote the statement in line k as S_k. The replayer first tries to execute S_4 of thread T_2, a write to the variable Y. Since the **wlog** of Y indicates that the first write to Y is by thread T_1, T_2 is suspended. When the scheduler continues to execute S_6 of T_3, since it is a read, the replayer consults the **rlog** and obtains the tuple $(1, 3)$. This tuple means the value read from variable Y is 1, of which the write version is not larger than 3. Since the third version is not yet computed, it is not T_3's turn to execute and T_3 is also suspended. Similarly, T_4 is suspended. The replayer then executes S_1, writes value 0 to variable Y and updates its version as 1, denoted as Y_0^1 in Figure 3. At this point, S_4 as well as S_2 and S_5 can be executed, which produce the second version of Y, the first version of X, and the third version of Y, respectively. Consequently, the execution of S_6 of T_3, which is previously suspended, can finally be executed as follow. Since S_6 of T_3 reads value 1 of Y of version smaller than 3, we need to search all writes of Y of older versions that writes the value 1. In our example, the match is S_5 of T_2. An exact linkage $R6 \rightsquigarrow W5$ is computed, as shown by the arrow in Figure 3. Linkages $R7 \rightsquigarrow W3$ and $R8 \rightsquigarrow W2$ can be reasoned in the same way. The execution of the last statement S_9 particularly shows the strength of linkage bounding. The **rlog** indicates that we are reading 0 of Y no later than version 3. This means that we only look for writes that produce 0, with the associated versions not larger than 3. Through a simple linear scan, we can easily compute the last linkage: $R9 \rightsquigarrow W4$.

From this example, we can observe that, for the order-based replay technique, we need to insert nine synchronization operations in this short piece of code to protect nine shared variable accesses, whereas Stride only needs five.



Thread	T1	T2	T1	T2	T3	T4	T1	T3	T4
Statement	1	4	2	5	6	8	3	7	9
Opeartion	$W(Y_0^1)$	$W(Y_0^2)$	$W(X_1^1)$	$W(Y_1^3)$	$R(Y_1^3)$	$R(X_1^1)$	$W(X_0^2)$	$R(X_0^2)$	$R(Y_0^3)$

Wlog: X: T1,T1 Y: T1,T2,T2
Rlog: T3: (1,3) (0,2) T4: (1,1) (0,3)

Figure 3. Replaying the example program using Bounded Linkage

$$
\begin{aligned}
\textbf{Execution Log} &::= \textbf{LW}_x \;\; \textbf{LA}_l \;\; \textbf{TR}_i \\
\textbf{LW}_x(x \in \mathbb{SV}) &::= (\text{i of } W_x^i(v))^* \\
\textbf{LA}_l(l \in \mathbb{L}) &::= (\text{i of } L_l^i)^* \\
\textbf{TR}_i(i \in [1, K]) &::= (\text{v of } R_x^i(v), \textbf{BL}_x^i)^* \\
\textbf{BL}_x^i &::= [0-9]+
\end{aligned}
$$

Figure 4. Formalism of the concurrent program execution Log.

More importantly, since Stride allows the CREW semantic, the execution of threads T_3 and T_4 can be completely in parallel, leading to the more efficient recording run. In the following sections, we will describe how Stride works, why it is correct, as well as the engineering challenges that we have encountered.

III. PRELIMINARIES

In this section, we formalize the essential concepts as well as the problem addressed in this paper.

A. Execution Log of Concurrent Program

We adopt the notations of a previous work [25] to define the concurrent program as a set of threads $\mathbb{T}$: T_1, T_2, ..., T_K, communicating through a set of shared variables, $\mathbb{SV}$, residing in a single shared memory protected by a set of locks $\mathbb{L}$. The thread T_1 is the main thread that *forks* other threads at runtime. All the operations executed by thread T_i can be numbered in order and we use PC_a^i to denote the execution number of an operation a.

Formally, Figure 4 gives the definition of our execution log for a concurrent program. The symbols (*e.g.* R_x^i) define the following operations:

- $R_x^i(v)$: read value v of variable x by thread T_i.
- $W_x^i(v)$: write value v to variable x by thread T_i.
- L_l^i: acquire lock l by thread T_i.
- U_l^i: release lock l by thread T_i[2].
- F_j^i: fork a new thread T_j by thread T_i.
- J_j^i: join the thread T_j to thread T_i.

---

[2] U_l^i, F_j^i, and J_j^i is not used in the execution log. We define them here to describe all the operations concerned by Stride

/* Program Order */

$$\pi_1 := \forall a, b \in TE_i : (PC^i_a < PC^i_b) \Rightarrow a <_{sc} b$$

/* Weak Total Order */

$$\pi_2 := \forall a, b \in \mathit{ops} : a \neq b \Rightarrow (a <_{sc} b) \vee (b <_{sc} a)$$

/* Asymmetric Order */

$$\pi_3 := \forall a, b \in \mathit{ops} : a \neq b \wedge (a <_{sc} b) \Rightarrow \neg(b <_{sc} a)$$

/* Exact Read Write Linkage */

$$\pi_4 := \forall a \in \mathit{ops}_R : \exists b \in \mathit{ops}_W (\mathit{var(a)} = \mathit{var(b)}) \wedge (b <_{sc} a) \wedge$$
$$\neg(\exists c \in \mathit{ops}_W, \mathit{var(c)} = \mathit{var(b)} \wedge b <_{sc} c <_{sc} a)$$

Figure 5. Sequential Consistency Specification. (ops_R and ops_W denote all the reads and writes respectively, and *ops* denotes all the concerned operations. *var(a)* is the variable of the operation a accessed.)

An execution log is divided into three disjoint parts. LW$_x$ stands for the *local total order* of the writes to a shared variable x. Specially, we say the k^{th} write in LW$_x$ is of *version* k. LA$_l$ is the *lock acquisition* log recording the lock/unlock order for the lock l for reproducing dead locks. TR$_i$ is the *read log* of thread T_i. Each item in a read log is a two-tuple representing the value returned by the read operation and the latest version of write that read can possibly link to (the bounded linkage). The read value can be used to faithfully replay the thread-local execution trace for each thread (see Section IV-A), while the bounded linkages are used to guide the search for the exact read-write linkages (see Section IV-B).

B. Memory Model and Legal Schedule

A *memory model* defines the set of values committed by writes that are allowed to be returned by a read [26]. The most strict memory model for concurrent programs is *sequential consistency* (SC). Lamport defines sequential consistency as: the result of any concurrent execution is the same as that the operations on all the processors are executed in some sequential order and the operations of each individual thread appear in the program order [16]. Axiomatically, we define a *legal schedule* under SC to be a *total order* ($<_{sc}$) of the read write operations that conform to the memory behaviour rules given in Figure 5. Among the rules, π_1 reflects the program order, π_2 and π_3 restrict the shared memory operations to be executed sequentially, and π_4 mandates the read can only return the most recent value written to the same memory location. If more than one legal schedule can be found, we say they are *equivalent* to each other.

C. Problem Definition

/* Lock and Unlock Matching */

$$\pi_5 := \forall a = L^{i_1}_{l_1}, b = L^{i_2}_{l_2} : ((l_1 = l_2) \wedge (a <_{sc} b)) \rightarrow$$
$$(\exists c = U^{i_3}_{l_3}, (l_3 = l_1) \wedge (i_3 = i_1) \wedge (a <_{sc} c <_{sc} b))$$
$$\pi_6 := \forall a = U^{i_1}_{l_1} : \exists b = L^{i_1}_{l_1}, (b <_{sc} a) \wedge$$
$$\neg(\exists c = U^{i_1}_{l_1}, b <_{sc} c <_{sc} < a)$$

/* Fork and Join Constraints */

$$\pi_7 := \forall a = F^i_j, \forall b \in T_j, a <_{sc} b$$
$$\pi_8 := \forall a = J^i_j, \forall b \in T_j, b <_{sc} a$$

Figure 6. Thread control axioms for lock/unlock and fork/join.

Given an execution log, the task of replay, or of the execution composition is to generate a total order of all operations such that the reads and writes conform to the sequential consistency memory model and, meanwhile, the lock/unlock, as well as the fork/join operations, conform to the thread control axioms described in Figure 6. Rules π_5 and π_6 define a lock that can only be held by one thread at a time. Rules π_7 and π_8 guarantee a thread must be executed after the fork operation and before the join operation. Similar to the previous work [8], [16], we synthesize a valid execution by sorting the happens-before graph topologically (see Section IV).

Our core research question is how to rediscover the exact RW-linkages via the read logs (TR$_i$) and the write sequences (LW$_x$). The bounded linkages in the read log is a number describing a *bounded write version* for a read R^i_x. The bounded write version is used as an upper bound to search for the matched write for any read. If, for example, a read R^i_x has a bounded linkage 9, it means the matched write W^j_x of this read is placed before or equal to the position 9 (starting from 1) in LW$_x$. In the next section, we will show how to instrument the program, how to infer the exact RW-linkage, as well as the proof of why this algorithm can compute an execution equivalent to the original execution.

IV. A THEORY OF EXECUTION COMPOSITION BY BOUNDED LINKAGES

A. Program Instrumentation

We first perform a thread escape analysis [27] to identify all the shared variables ($\mathbb{SV}$). Next, we normalize the program so that the result of reading a shared variable is first stored in a local variable and use that local variable in the subsequent computation. For example, if a statement, $x = y + z$, involves three shared variables x, y, and z, we change the code into three statements: $a = y$, $b = z$, $x = a + b$. After the transformation, each statement can access at most one shared variable.

To collect the execution log, we instrument the program as shown in Table IV-B. For each shared variable x, we maintain a version value V_x. The statements labelled with W_c (version update) and R_c (version snapshot) in the instrumented code for the shared write and read guarantee any bounded linkage is a searching upper bound. This is because, since W_c must execute before the write to x, and R_c must execute after the read from x, the matched write of R_x^j is always positioned before or equal to its bounding write W_x^i.

The full details of the thread execution as well as the unlock operations can be reconstructed during replay. When replaying, since the only way for one thread to be affected by another thread is by reading a value[3], the values in the read log can help faithfully reproduce a thread's local behaviour. For reproducing the orders of write and lock operations, logging the execution as a sequence of thread IDs is also sufficient since the program order is available in the replaying run. Since a lock operation must be followed by a corresponding unlock operation, the sequence of unlock information is also available. Thus, in the rest of this section, we assume the full details of each thread's execution and the lock/unlock sequence are already obtained in the replay run.

Table I
PROGRAM INSTRUMENTATION ILLUSTRATION. ALL CODE IS EXECUTED IN THREAD T_i, AND THE UNDERLINED STATEMENTS ARE OUR INSTRUMENTED CODE.

Write	Read	Lock/Unlock

$\underline{Synchronized(l_x)}$ {
$W_c : \underline{V_x{+}{+}}$;
$\mathbf{x} = \mathbf{a}$;
$\underline{LW_x.add(i)}$;
}
	$\mathbf{a} = \mathbf{x}$;
	$R_c : \underline{v = V_x}$;
	$\underline{BL_i.add(v)}$;
	$\underline{TE_i.add(a)}$;
		$\mathbf{l.lock}()$;
		$\underline{LA_l.add(i)}$;
		$<$ **code** $>$
		$\mathbf{l.unlock}()$;

Algorithm 1 Infer the exact linkages for all reads

```
 1: procedure LinkageInfer
 2:     for all thread $T_i$, $i \in [1, K]$ do
 3:         for all $R_i^x(v)$ in $T_i$ with bounded linkage $bl$ do
 4:             SearchForMatch($R_i^x(v)$, $bl$)
 5:         end for
 6:     end for
 7: end procedure
 8:
 9: procedure SearchForMatch($R_i^x(v)$, $bl$)
10:     for ($k = bl$; $k > 0$; $k$- -) do
11:         if WriteValueOf($LW_x[k]$) == $v$ then
12:             return $LW_x[k]$        ▷ Found the exact linkage
13:             exit for
14:         end if
15:     end for
16: end procedure
```

B. Inferring Exact Read Write Linkage

Composing a feasible execution requires a happens-before graph that encodes the legal schedule constraints, which, in turn, needs the exact read write linkages. Fortunately, turning our bounded linkages to exact linkages can be achieved by a simple linear scan, which is given in Algorithm 1.

The core of Algorithm 1 is the SearchForMatch procedure. For each read operation (we suppose it reads variable x), we search from the upper bound bl backward to index 1 in the local write log (LW_x) and stop at the first write that writes the value returned by this read.

The time complexity of Algorithm 1 is $O(Kn)$, where n is the total length of the execution log, and K is the number of threads. This is because, although the lower bound for the search in Line 10 is 0, the j^{th} read in thread T_i cannot match a write of an older version than the bounded linkage of the $(j-1)^{th}$ read. Therefore, the loop from the Line 3 to Line 5 in the worst case examines $O(n)$ operations. Since we only query $O(n)$ times for the exact linkages, the average execution time of SearchForMatch is $O(Kn/n) = O(k)$, which is extremely fast if only a small number of exact RW-linkages are to be recovered.

The last question is why the first matched write guarantees the legal schedule. Recall that a legal schedule is obtained by sorting the happens-before graph topologically. Hence, it is essential to prove that graph has no cycle. Formally, a happens-before graph is constructed as follows:

Definition 4.1: A happens-before graph has all the executed statements as its nodes. The edges are built by:
(a). If R_x^i reads the value written by W_x^j, we add edges $W_x^j \rightarrow R_x^i$ and $R_x^i \rightarrow W_c$ where W_c is the version-update statement for the next write W_x^j in LW_x;
(b). For any two adjacent writes W_x^{i1} and W_x^{i2} in LW_x, we add $W_x^{i1} \rightarrow W_c$, where W_c is the version-update statement for W_x^{i2};
(c). If statements a and b are both executed in T_i and $PC_a^i <_{sc} PC_b^i$, we add $a \rightarrow b$;
(d). For an unlock operation U_l^i, we add an edge to the next lock operation L_l^j for the same lock l;
(e). For any fork operation F_j^i, we add an edge from F_j^i to the first operation of thread T_j;
(f). For any join operation J_j^i, we add an edge from the last operation of thread T_j to J_j^i.

It is straightforward to validate that the happens-before graph constructed by Definition 4.1 satisfies all the axioms in Figure 5 and Figure 6. Therefore, a topological sort on this graph gives a legal execution. Since the original execution is a legal execution, by the definition of equivalent execution stated in Section III-B, the computed result is equivalent to the original execution if and only if there exists a topological sort in the happens-before graph, or in other words, the happens-before graph has no cycle. Since only rule(a) uses the inferred result, we only need to prove that rule(a) does not incur any cycle.

Theorem 4.1: The exact read write linkages computed by Algorithm 1 leads to an acyclic happens-before graph.

---

[3] none memory access operations cannot affect the execution path of a thread, thus will not affect a thread's local behaviour

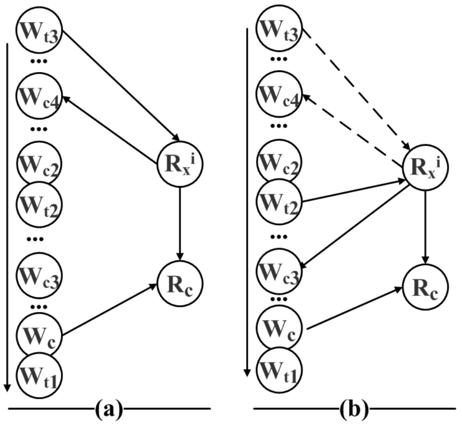


Figure 7. Happens-before graphs with different RW-linkages.

Proof: Suppose the bounded linkage for a read R_x^i is $t1$, and the matched write found by Algorithm 1 is positioned at $t2$ ($t2 \leq t1$). We first prove that, if there is another match $t3$ ($t3 < t2$) that forms a happens-before graph with no cycle, so does the match $t2$.

We use Figure 7(a) to show the part of the happens-before graph around the RW-linkage W_{t3}–R_x^i. W_c and W_{c2} are the version-update statements corresponding to W_{t1} and W_{t2}. W_{c3} and W_{c4} are the version-update statements corresponding to the write operations next to W_{t2} and W_{t3} in LW_x, respectively. The graph on the right (Figure 7(b)) is a modified version of Figure 7(a), in which the edges $W_{t3} \rightarrow R_x^i$ and $R_x^i \rightarrow W_{c4}$ are replaced by $W_{t2} \rightarrow R_x^i$ and $R_x^i \rightarrow W_{c3}$. Our aim is to show, if Figure 7(a) has no cycle, Figure 7(b) is also acyclic.

Because we only add two edges in Figure 7(b), there are only two chances to form a cycle:

(I) *The circle formed by the edge $R_x^i \rightarrow W_{c3}$ and the path $W_{c3} \rightsquigarrow R_x^i$.* The path $W_{c3} \rightsquigarrow R_x^i$ does not exist because, otherwise, there is a path $W_{c4} \rightsquigarrow R_x^i$ and Figure 7(a) has a cycle, which contradicts our assumption.

(II) *The circle formed by the edge $W_{t2} \rightarrow R_x^i$ and the path $R_x^i \rightsquigarrow W_{t2}$.* Because R_x^i has only two outgoing edges, the path $R_x^i \rightsquigarrow W_{t2}$ must start with one of them. $R_x^i \rightarrow W_{c3}$ cannot be picked because, otherwise, there is a path $W_{c3} \rightsquigarrow W_{t2}$. Since W_{t2} must be executed before W_{c3} according to our local write order constraint, there is a path $W_{t2} \rightsquigarrow W_{c3}$ in Figure 7(a). Therefore, together with the path $W_{c3} \rightsquigarrow W_{t2}$, we have a cycle in Figure 7(a), which is a contradiction. If we pick $R_x^i \rightarrow R_c$ as the first edge, it implies that there is a path $R_c \rightsquigarrow W_{t2}$. Also, since there is only one incoming edge of W_{t2} from W_{c2}, there should be a path from R_c to W_{c2}. Since there is a path $W_{c2} \rightsquigarrow W_c$[4] and an edge $W_c \rightarrow R_c$, there must be a cycle in Figure 7(a), which again contradicts our assumption.

Now, we have proved Figure 7(b) also has no cycle. Since we know R_x^i must read from some write W_{real}, and W_{real} is either W_{t2} or the one that is placed preceding W_{t2} in LW_x, it immediately follows that the exact RW-linkage W_{t2}–R_x^i cannot form a cycle. Since R_x^i is chosen arbitrarily, we can conclude that the happens-before graph has no cycle. ■

V. FROM THEORY TO ENGINEERING

In the previous section, we have discussed our core contribution of inferring a legal execution with bounded linkages. A few engineering challenges still remain.

A. Execution Log Compression

For the read logs (TR$_i$), we compress the read values and the bounded linkages separately. The common way of compressing the read values is using the *last one value predictor* [28], which is also adopted by the tracing tool iDNA [29]. Specially, for each shared variable x, we maintain a shadow memory in each thread T_i to record its last accessed value and a counter to record the prediction hits rate. When $R_x^i(v)$ is executed, we compare the value v to its current shadowed value v'. If they are equal, we increment the corresponding counter by one. Otherwise, we output an entry (value, counter) to the log and update the shadow memory using v and reset the counter to 1. For a write $W_x^i(v)$, we only update the corresponding shadow memory to v and reset the counter to 1.

The memory footprint can be very large if we create a shadow memory for every shared variable at runtime. To limit the memory usage, we use a hash function so that two different variables can share a shadow memory if they have the same hash value. According to our experiment, a 10MB shadow memory for each thread is very effective for log compression.

We compress the bounded linkages in the read logs, the local write logs (LW$_x$), and the lock acquisition logs (LA$_l$), by replacing the consecutive n elements with the same value t with an 2-tuple (t, n) (a form of run length encoding). For example, we merge the sequence 1, 1, 1 into (1, 3).

B. Variable Grouping

Maintaining the order and the version for each variable is costly due to the large amount of memory used in the execution of the original program. Stride uses the *context insensitive* and the *field based* model [30] to abstract the program and map the runtime shared variables to the symbolic variables, also adopted by Leap [9]. Supposing a and b are two runtime instances of class C, the runtime variables $a.f$ and $b.f$ are treated as the same variable f that share the same local write log LW$_f$ and the same version value.

When a program has strong locality and a small number of context switches, a group of variables may be accessed by the same thread for a period of time. Such property results in a lot of adjacent log entries having the same value in both

---

[4] Particularly, if $t_1 = t_2$, W_{c2} is essentially the W_c.

the read and the write logs. This can be used to improve the compression rate of the run length encoding. The *last one value predictor* for logging the read values, however, cannot benefit from the grouping of the variables, since the value in each memory unit is supposed to be different. For example, thread T_1 updates $x_1, x_2 ... x_n$ and then thread T_2 reads $x_1, x_2 ... x_n$. If we group $x_1, x_2 ... x_n$ together as variable x, recording (1,n)[5] for the write order log and (n,n)[6] for the bounded linkages is enough. However, we have to record all the values of $x_1, x_2 ... x_n$, since the value of x_1, x_2 ... x_n are supposed to be different from each other.

We have designed a novel compression technique to deal with this problem. If we can confirm the version value is the exact linkage but not merely a bounded linkage, the read value need not be logged. This is because the read value can be recovered by loading the write value of its exact linkage write in the replaying run. To implement our idea, we update the version value twice for each write operation instead of once in the original algorithm. One is put before the write and the other is put after the write. If a version value is even and it is the same as the last version recorded, the version recorded is actually an exact linkage, since under this condition, no new value is written. Thus, the read value need not be recorded. By this means, we can achieve similar compression rate as other logs for the read values in programs with strong locality and infrequent context switches.

C. Optimization for Race-free Programs

If the read and the write operations to a variable are all protected by a lock, logging the acquisition order of the lock can regenerate the shared access orders for the variable and, thus, deterministically reproduce the execution [31]. More precisely, if a variable is protected by a lock for both read and write operations, we insert no instrumentation for this variable. If a variable is protected by a lock for all the write operations, we only record the read logs for the variable, since under this condition, the write order can be deduced from the lock order. This treatment leads to a great runtime overhead reduction. The experimental details are given in Section VI-E.

D. Objects Correlation in Different Executions

In Java, the address of an object is represented by a hash code. As the hash code is dynamically assigned to an object, two executions of the same program of the same allocation statement may return different hash codes. To correlate the same objects created in different executions, we assign a *birthday* to every object and maintain a *hashcode-birthday* map. More precisely, for each thread, we maintain a counter C_{birth}. When an object is created, we map the hash code of that object to C_{birth} and increment the counter by one. After the execution ends, we dump the map between the hash code and birthday counter. During the replay run, we assign the birthday to every object in the same way as above. But this time, we maintain a *birthday-object* map. If the logged value of a pointer variable is t, we immediately translate t to the birthday using the hashcode-birthday map, obtained in the recording run, to lookup the referred object. In this way, the object correlation is easily achieved with low performance penalty. Since the execution control flow for each thread is guaranteed to be same in two runs, the birthday method is sound.

VI. EVALUATION

We assess the quality of Stride by quantifying its recording overhead, its log size, and the inference cost. We have implemented Stride for Java using the Soot framework[7]. We compare our approach to our earlier work Leap [9], a representative approach[8] in using the exact linkage to deterministically replay concurrent Java programs. To conduct a fair comparison, we group the variables for Stride in the same granularity as Leap. We have also implemented the work of Cantin *et al.* [24], referred to as Global in the rest of the paper, that maintains a global write order in order to deterministically replay. For Global, there is no need of grouping since we must maintain the global order of all the write operations accessing each shared variable. We do not compare Stride to the search-based techniques, because, unlike Stride, the search-based techniques are not deterministic.

All experiments are conducted on a 8-core 3.00GHz Intel Xeon machine with 16GB memory and Linux version 2.6.22. We selected a wide range of benchmarks to evaluate our approach. Avrora, Batik, H2, Lusearch, Sunflow, Tomcat, and Xalan are from the *Dacapo* suite[9]. Moldyn is a scientific computation program from the *Java Grande* benchmark suite. Tsp is a parallel algorithm solving the Travelling Salesman Problem. We also include Derby, a widely used database engine, SpecJBB2005, a benchmark for parallel business transactions, and ICE, a high performance implementation of the protocol buffer[10] IPC specification.

A. The Study of Recording Overhead

Table II presents the experimental results for the selected benchmarks. The column *Read Percentage* presents the percentage of read operations among all concerned operations (described in Section III-A) during the execution. The third column reports the average comparison time during the *infer* stage. The 4^{th} to 6^{th} columns report the runtime overhead,

---

[5] (1,n) stands for the next n writes are issued by thread T_1.

[6] (n,n) stands for the next n read operations reads version n.

[7] http://www.sable.mcgill.ca/soot/

[8] A more recent work [15] successfully applies our technique in the JVM.

[9] The reflections in *Dacapo* suite are solved using *tamiflex* (http://code.google.com/p/tamiflex/)

[10] http://www.zeroc.com/labs/protobuf/index.html

Table II
PERFORMANCE FOR REAL APPLICATIONS

		Infer Efficiency	*Overhead (X)*			*Log Size(/s)*		
Benchmark	*Read Percentage*	*Avg compare time*	*Stride*	*Leap*	*Global*	*Stride*	*Leap*	*Global*
Avrora	70.45%	1.00094	10.58	19.61	18.65	257.4MB	707.5MB	87.1MB
Batik	84.02%	1.00002	0.08	0.16	0.21	1.5KB	4.3KB	691.7KB
H2	93.06%	1.00000	0.62	2.08	2.12	0.569MB	2.382MB	51.353MB
Lusearch	79.90%	1.00076	7.46	21.47	19.20	205.8MB	685.7MB	146.0MB
Sunflow	92.20%	1.00007	2.55	6.62	4.62	27.2KB	296.6KB	52758KB
Tomcat	77.18%	1.00685	0.09	0.14	0.15	133.6KB	385.7KB	105.1KB
Xalan	87.92%	1.00428	0.81	4.26	4.87	30.8MB	133.1MB	36.9MB
Tsp	89.54%	1.00216	1.54	16.46	4.03	39.8MB	554.7MB	12.6MB
Moldyn	99.40%	1.00027	1.50	113.5	4.99	27.3MB	3834MB	37.2MB
Derby	83.18%	1.00008	0.05	0.10	0.05	2.1KB	4.2KB	2.1KB
SpecJBB	95.46%	1.00000	0.11	0.13	0.12	2.9KB	5.1KB	1.5KB
ICE	95.46%	1.00005	2.06	7.26	1.93	5.57MB	21.21MB	6.14MB

which is the gap of the execution time between instrumented code and the original code, normalized based on the original execution time. The last three columns report the log size for one second of execution.

Our first study looks at the most important characteristic of a replay technique, the recording overhead. Compared to the original programs, the overhead of Stride is below 1X in 6 of the 12 subjects and below 2X for the two evaluated scientific computation benchmarks(TSP and Moldyn) that intensively access shared variables. For Tomcat, Derby, and Batik, the overhead is less than 10% which is attractive even for the production usage.

Compared to Leap, our measurements show that Stride incurs on average of 2.5X smaller runtime slowdown if we consider the subjects Moldyn and Tsp as special cases, where Stride is 11X and 75X better, respectively. Stride only incurs a 5% slowdown on Derby because Derby rarely accesses shared variables. Although the write operations on the same variable cannot execute in parallel, the number of such operations is small and most of them have already been protected by locks. Therefore, there is no need for Stride to insert locks. For Moldyn, despite that the program accesses shared memory very frequently, 99.4% of the operations are read operations. Under this condition, tracking the exact read-write linkages is very expensive due to the large amount of additional locks.

Stride also performs better than Global for 11 out of the 12 subjects. Global requires a global lock for all of the write operations to shared variables, such that any two write operations, whether they access the same memory location or not, can not execute in parallel. This increases the lock contention drastically if the thread number gets large. For ICE, the performance of Global is slightly better because ICE frequently accesses the same shared variable. Stride and Global incur a similar degree of lock contention in this case. Since Global does not maintain the write version, it performs better than Stride. However, this case shows that maintaining and logging the write versions incur very small overhead because the performance gap between Global(1.93X) and Stride(2.06X) is small.

An interesting finding is that Global, which is assumed not practical, performs better than Leap for 8 out of the 12 subjects due to the removal of the lock contention for read operations. Since the read operation contributes 70% to 99% of the total amount of operations on the shared variables, Global has the comparable performance with respect to Leap.

B. The Study of Log Size

For the log size, Table II shows that Stride performs better than Leap for all of the 12 subjects. Leap produces, on average, 3.88X of the log size of Stride without counting our best cases Tsp and Moldyn. Compared to Leap, Stride only tracks the write operations which is fewer in number and easier to compress. In addition, the read operations usually read a value written by the same thread which need not be recorded. In the subjects Derby and SpecJBB, the gap on log size between Leap and Stride is less than 2X, due to the fact that the interleaving is not very frequent, making the compression algorithm of Leap very effective. However, for Moldyn, which intensively accesses shared memory, the log size of Stride is only 27.3MB per second, which is more than 140X smaller than that of Leap. One reason is that 99% of the operations in Moldyn are reads, for which Leap needs to insert locks for recording the thread access order. Besides, in Moldyn, the value updated by write operations are very frequently checked by most of the threads, making it very easy for Stride to reduce the log size but quite hard for Leap.

The log size of Global is even smaller than Stride in 4 of the 12 benchmarks. This is because, in these four subjects, the write operations rarely update new values and the reads mostly return the same value. The entropy of the log files is low, which favours compression algorithms a lot. On the contrary, for Sunflow, H2 and Batik, Global

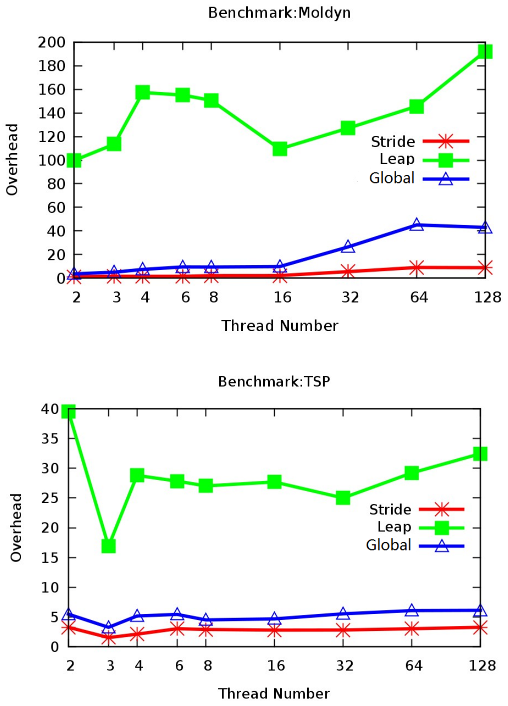


Figure 8. Overhead VS Thread Number(X-coordinate specifies the amount of thread, y-coordinate specifies the overhead normalized based on the original execution time)

incurs very large log sizes because of the opposite reasons: the writes often update the values of shared variables and these updates are checked by other reading threads, causing a lot of recording of read values. Stride encounters similar problems. But our double versioning technique can provide an optimization (see Section V-B) to solve this problem. Therefore, the log size of Stride is also small under such conditions.

C. The Thread Scalability Study

We are also interested in investigating how the recording overhead and the log size scale with respect to the increasing number of threads used. Since Dacapo has self-configured thread numbers, we select two benchmarks: Moldyn, where almost all the operations accessing shared memory are read operations, and Tsp, a subject that has the normal percentage of reads and writes to the shared memory. The observed overhead is shown in Figure 8 and the log sizes are shown in Figure 9. We can see that, for Stride, the overhead increases from 1.5X to 8.81X for Moldyn and from 1.54X to 3.27X for TSP, when the number of threads increases from 3 to 128. When the number of threads increases from 3 to 128, the log size for Stride also increases from 27.3MB/s to 325.4MB/s for Moldyn and from 39.8MB/s to 60.3MB/s for TSP. Also, we find that when thread number increases, the recording overhead of Global increases 5X faster than Stride for Moldyn and 2X faster for TSP. This is consistent with the theoretical conclusion that Global does not fit for highly parallelized executions.

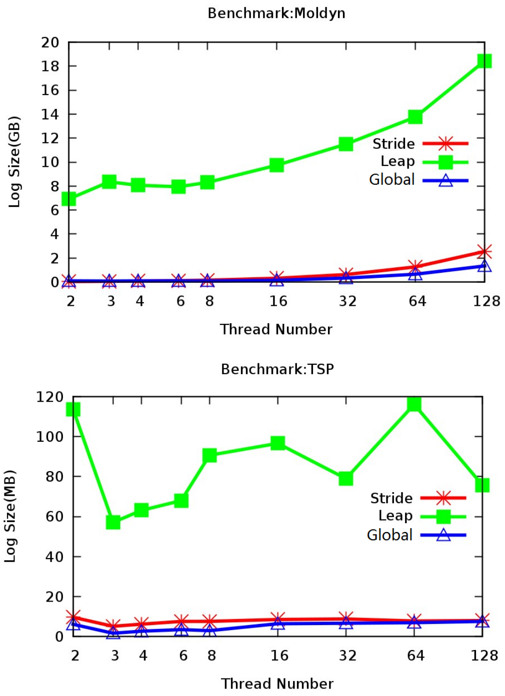


Figure 9. Log Size VS Thread Number(X-coordinate specifies the amount of thread, y-coordinate specifies the exact log size)

D. The Cost of Inferring the Exact Linkage

In this study, we quantify the inference cost of Stride since we only record a bound for the exact linkage in the log. For each read operation in the log, we need to linearly scan all the write operations that have smaller version numbers than the bound. Given the huge amount of read operations, it is crucial that the scan needs to be very fast. The *Avg Compare Time* column of Table II shows the average number of lookups during the scan is very close to 1 for all of the 12 subjects. This shows that the number of preemption between the read operation and the following read of the bound is very small in practice. For Avrora, where interleaving frequently happens, there are 445.8 million out of 446.2 million read operations to shared memory can be solved in the first comparison, 403923 (0.4 million) in the second, 249 in the third. Only 58 read operations requires 4 or more comparison. We have similar findings for the other 11 subjects. In the subject Xalan, we detected two cases that the scan requires more than 7000 lookups. Overall, we conclude that, although the complexity of inferring an exact linkage is on average $O(k)$ in theory, the average complexity in practice is almost $O(1)$.

E. Race-free Condition

Our final study explores the optimal recording overhead of Stride assuming that, in well engineered programs, the

Table III
RACE-FREE OPTIMIZATION

	Overhead(X)	*ProtectedRW*	*ProtectedW*
Avrora	3.54	0.60%	70.21%
Batik	0.05	1.47%	64.06%
H2	0.48	2.58%	29.10%
Lusearch	3.10	5.12%	63.57%
Sunflow	1.35	3.08%	50.54%
Tomcat	0.05	6.77%	56.77%
Xalan	0.45	1.28%	47.53%
Tsp	0.78	41.37%	89.66%
Moldyn	1.16	9.30%	36.05%
Derby	0.02	2.54%	84.43%
SpecJBB	0.10	0.78%	47.24%
ICE	1.55	19.57%	79.80%

unprotected writes are intentional, i.e., the write-write race is benign. In this case, Stride does not need to add any additional locks to the program and is still able to deterministically replay it. Table III reports the overhead normalized against the original execution time. We find that the overhead is on average only 1X and even less than 4X for Avrora where there are lots of hot loops accessing the shared memory. This result is significant because all of the order based techniques, such as Leap [9], Order [15], and Recplay [31], requires the program to be both Read-Write and Write-Write race free if no locks are to be added. Also as reported in Table III, the percentage of variables that both reads and writes are protected (*ProtectedRW*) is much smaller than those to which writes are protected (*ProtectedW*). Stride is much more efficient if this assumption holds in practice.

VII. RELATED WORK

PRES [23] and ODR [21] are two recent search-based projects. PRES uses a feedback replayer to explore the thread interleaving space. It reduces the overhead by adding more replay attempts. ODR focuses on reproducing the same output and reason a possible execution with the offline inference in order to alleviate the online recording overhead. Weeratunge *et al.* [19] presents a way to guide the offline inference based on the core dump without any online overhead. These approaches provide no guarantee of reproducing a feasible execution trace and they all report the cases that they fail to reproduce a run in several hours.

LEAP [9] and Order [15] are two state of the art order based techniques that directly record the order of shared memory accesses. They carefully adjust the granularity of how the shared memory cells are grouped to avoid the contentions caused by additional synchronizations. Netzer [32] presents a method on minimizing the amount of logged exact RW-linkages in recovering the same execution trace, which make the further reduction of the runtime cost hard for the order-based techniques. DoublePlay [33] breaks this bound by executing the program twice using two different parallel strategies and comparing the effect of the executions. Instead of maintaining the exact linkage, DoublePlay link the read and write operations by value. DoublePlay can achieve a lower recording overhead. But the change of the parallel strategy requires the low-level control permission and the hardware support. Our work, however, provides a general theory on how to perform the read-write mapping in polynomial time.

To avoid the overhead of recording memory races, RecPlay [31] and Kendo [34] replay race-free multithread programs by logging lock sequences. Both the approaches use a data race detector during replay to ensure the replay determinism until the first race. However, they suffer from the limitation that they cannot replay past the data race. Unfortunately, most real world concurrent applications contain low-level data races. Our work relaxes the the race free requirement to be the write-write race free, which favours many well-engineered concurrent programs.

Bhansali *et al.* [29] presents iDNA, an instruction level tracing framework. Their work records all the values read from or written to a memory cell. They use a memory predictor to compress the value trace. iDNA incurs on average 11X runtime overhead and the trace size of tens of mega-bytes per second, by recording all the values from memory access operations. Unlike tracing techniques, our replay technique requires logging only the memory access to the shared memory, for which only the read value written by a different thread is required to be recorded. Thus the recording overhead and the log size for Stride can be much smaller than that of iDNA.

VIII. CONCLUSION

We have presented Stride, a deterministic replay technique for multi-thread programs by *recording* the *bounded linkages* of read and write operations and then *inferring* an equivalent execution in almost linear time. Our method achieves a low runtime overhead by removing the additional synchronizations on read operations and allows the *concurrent read exclusive write* semantics. Our experiments show that, compared to the state-of-the-art, Stride incurs 2.5 times smaller runtime slowdown excluding our best cases for which the gap can be up to 75 times. The log size is also on average 3.88 times smaller excluding our best cases, for which our log size is 140 times smaller. Besides, our work makes more space for further optimization by leveraging the restriction of being low level race free to write-write race free. Since our technique focuses on the problem of what to record but not how to record, it can also be directly applied for many order-based techniques as an optimization.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive comments. This research is supported by RGC GRF grants 622208 and 622909.

REFERENCES

[1] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," ser. ATEC '05, 2005.

[2] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," ser. ATEC '04, 2004.

[3] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," ser. SOSP '07, 2007.

[4] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, vol. 14, February 1996.

[5] S. Medini, P. Galinier, M. D. Penta, Y.-G. Gueheneuc, and G. Antoniol, "A fast algorithm to locate concepts in execution traces," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, 2011, vol. 6956, pp. 252–266.

[6] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," ser. ISCA '05, 2005.

[7] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira, "Offline symbolic analysis for multi-processor execution replay," ser. MICRO 42, 2009.

[8] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, August 1997.

[9] J. Huang, P. Liu, and C. Zhang, "Leap: lightweight deterministic multi-processor replay of concurrent java programs," ser. FSE '10, 2010.

[10] D. R. Hower and M. D. Hill, "Rerun: Exploiting episodes for lightweight memory race recording," ser. ISCA '08, 2008.

[11] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," ser. ISCA '08, 2008.

[12] S. Narayanasamy, C. Pereira, and B. Calder, "Recording shared memory dependencies using strata," ser. ASPLOS-XII, 2006.

[13] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th annual international symposium on Computer architecture*, ser. ISCA '03, 2003.

[14] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: efficient online multiprocessor replayvia speculation and external determinism," ser. ASPLOS '10, 2010.

[15] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang, "Order: object centric deterministic replay for java," ser. USENIX-ATC'11, 2011.

[16] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, September 1979.

[17] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," ser. PLDI '07, 2007.

[18] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient smith-waterman on multi-core with fastflow," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, 2010.

[19] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," ser. ASPLOS '10, 2010.

[20] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," ser. EuroSys '10, 2010.

[21] G. Altekar and I. Stoica, "Odr: output-deterministic replay for multicore debugging," ser. SOSP '09, 2009.

[22] N. Sinha and C. Wang, "On interference abstractions," ser. POPL '11, 2011.

[23] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: probabilistic replay with execution sketching on multiprocessors," ser. SOSP '09, 2009.

[24] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "The complexity of verifying memory coherence and consistency," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, July 2005.

[25] C. Flanagan and S. N. Freund, "Adversarial memory for detecting destructive races," ser. PLDI '10, 2010.

[26] S. V. Adve and H.-J. Boehm, "Memory models: a case for rethinking parallel languages and hardware," *Commun. ACM*, vol. 53, August 2010.

[27] R. L. Halpert, "Static lock allocation," Master's thesis, McGill University, April 2008.

[28] M. Burtscher and B. G. Zorn, "Exploring last n value prediction," in *IEEE PACT*.

[29] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," ser. VEE '06, 2006.

[30] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," ser. CC'03, 2003.

[31] M. Ronsse and K. De Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, May 1999.

[32] R. H. B. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," ser. PADD '93, 1993.

[33] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: parallelizing sequential logging and replay," ser. ASPLOS '11, 2011.

[34] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," ser. ASPLOS '09, 2009.