CARE: Cache Guided Deterministic Replay for Concurrent Java Programs

Yanyan Jiang, Tianxiao Gu, Chang Xu*, Xiaoxing Ma*, Jian Lu
State Key Lab for Novel Software Technology, Nanjing University, 210023, Nanjing, China
Department of Computer Science and Technology, Nanjing University, 210023, Nanjing, China
jiangyy@outlook.com, tianxiaogu@gmail.com, {changxu,xxm,lj}@nju.edu.cn

ABSTRACT

Deterministic replay tools help programmers debug concurrent programs. However, for long-running programs, a replay tool may generate huge log of shared memory access dependences. In this paper, we present CARE, an application-level deterministic record and replay technique to reduce the log size. The key idea of CARE is logging read-write dependences only at per-thread value prediction cache misses. This strategy records only a subset of all exact read-write dependences, and reduces synchronizations protecting memory reads in the instrumented code. Realizing that such record strategy provides only value-deterministic replay, CARE also adopts variable grouping and action prioritization heuristics to synthesize sequentially consistent executions at replay in linear time. We implemented CARE in Java and experimentally evaluated it with recognized benchmarks. Results showed that CARE successfully resolved all missing read-write dependences, producing sequentially consistent replay for all benchmarks. CARE exhibited 1.7–40× (median 3.4×) smaller runtime overhead, and 1.1–309× (median 7.0×) smaller log size against state-of-the-art technique LEAP.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms, Performance, Reliability, Theory

Keywords

Replay, Debugging, Cache, Concurrency

1. INTRODUCTION

Concurrent programs are error-prone but difficult to debug. In order to alleviate such debug difficulties, it would be better to have cyclic debugging techniques such as breakpoint and time-traveling debugging. However, this is a nontrivial task requiring cost-effective execution record for *deterministic replay* [23, 24, 32]. By faithfully reproducing the progress of the recorded execution, deterministic replay tools help programmers better pinpoint the root causes of such concurrency bugs.

The major challenge of replaying a concurrent program execution comes from its non-determinism, which is naturally associated with concurrency. On a uniprocessor machine, providing all external inputs and thread-scheduling data of an execution necessarily eliminates all of its non-determinism [12]. However, such information is still insufficient for replaying a concurrent execution on a multiprocessor machine, due to non-deterministic outcomes of parallel shared memory accesses. According to the methodology of tackling shared memory non-determinism, existing multiprocessor deterministic replay work can be roughly classified into two categories: *search-based* [3, 19, 29, 33, 36] and *order-based* [9, 13, 18, 35, 37]. Search-based techniques emphasize reduced record cost, thereby their recorded information is typically incomplete for a faithful replay. To construct a valid execution for debugging, search-based techniques usually use the best-effort exhaustive state space search. On the other hand, order-based techniques record dependences among all key events (e.g., variable-read/write and object-lock/unlock events) at runtime. Such dependence information is usually sufficient for a faithful replay, but may at the same time incur intolerably large record cost, e.g., huge log.

State-of-the-art order-based deterministic replay techniques have reported encouraging results in reproducing concurrent program executions. However, their record cost is still unsatisfactory. For example, Stride [37] generates over 30MB/s (2.5TB/day) of log in its evaluation. For programs that contain mass of shared memory accesses, order-based replay technique can easily generate hundreds of megabytes of log per second, and nobody in practice can afford such performance degradation that may last unpredictably long.

In this paper, we present an order-based deterministic replay technique that is capable of reducing the record cost, specifically, reducing the log size. This is achieved by a trade-off in the replay guarantee: our approach in theory gives only value-deterministic replay. Fortunately, we carefully design our algorithms such that they produce sequentially consistent replay executions for almost all programs in practice (actually all evaluated benchmarks), making it practically useful for debugging.

Our key insight lies in the fact that a dominant percentage of memory accesses exhibit *thread locality*, i.e., for a specific variable, its successive accesses can very likely be performed by a single thread[1]. Combining successive local memory accesses into a single log entry can greatly compact the log [8, 37]. To identify

*Corresponding authors.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

[1]In our evaluated benchmarks, 77% to 96% of shared variable accesses exhibit such thread locality.

local memory accesses, we innovatively use light-weight *software value prediction cache* to optimistically *overestimate* such locality property (i.e., identifying a superset of all such memory accesses with an ideal cache). Such overestimation reduces the log size, but also makes replay difficult because some critical shared memory access dependences might be lost. Therefore, we also propose two heuristics that can resolve such missing dependences.

Such an idea leads to our cache guided deterministic replay approach, CARE. Being a typical order-based technique, CARE logs a fraction of actual precedence orders between reads and their associated writes (i.e., read-write dependences). CARE assigns each thread with a value prediction cache that keeps buffered variable values. Each time a read action is executed, the buffered value is compared with the one actually read. Only when a distinct comparison result occurs (i.e., cache miss), the corresponding read-write dependence is logged. Note that this is only a *sufficient but not necessary condition* indicating that a thread is reading a value written by another thread. Therefore, CARE logs only a *subset* of all exact read-write dependences. Another benefit of this treatment is that in case of cache hit, CARE does not need to keep the read action synchronized, further reducing the runtime overhead.

To understand how cache can reduce the log size, consider an example that thread T_1 initializes a large data structure consisting of many variables. Thread T_2 then copies all these variables into T_2's thread-local storage. Now all order-based deterministic replay techniques including CARE keep read-write dependence for each of these variables. Suppose that later T_1 once again writes to this data structure, but overwrites most of the variables with their original values. After that, all these variables are read by T_2 a second time. This time, however, in contrast to state-of-the-art techniques like LEAP [18] and Stride [37] that keep exact dependences for all of these variables, CARE only keeps those dependences for variables being overwritten with different values, reducing the log size.

However, CARE's capability of reducing log size is at the cost of providing only value-deterministic guarantee at replay. Though each thread's local behavior can be faithfully reproduced, the global order of replayed actions might not be sequentially consistent, causing trouble for the debugging procedure. Fortunately, we propose two heuristics to mitigate such worst-case possibility: one is grouping variable sequence numbers into coarse-granularity atomic counters at record, and the other is prioritizing candidate actions at replay. With these two heuristics, we resolved all lost read-write dependences in our evaluation.

We implemented CARE as a prototype tool in Java[2] and evaluated it on a set of benchmark programs. CARE realized cost-effective record and replay for Java programs by reducing runtime overhead and log size. Our experiments showed promising results that CARE reduced 2.6× of runtime overhead and 4.9× of log size compared with state-of-the-art deterministic replay work LEAP [18] on the Dacapo benchmark suite, and up to 40× runtime overhead reduction and 300× log size reduction on the scientific benchmarks. For another state-of-the-art replay work Stride[37], CARE also reduced record cost and realized 4×–20× log size reduction relatively on some scientific benchmarks. In summary, we make the following contributions in this paper:

1. We propose using value prediction cache to reduce the record cost in deterministic replay of concurrent programs. This strategy yields a new trade-off between the record cost and the replay fidelity.
2. We present two heuristics for synthesizing sequentially consistent execution at replay, realizing that our approach ensures only value-deterministic guarantee.
3. We implemented our CARE approach and evaluated it experimentally with real-world benchmarks.

The rest of this paper is organized as follows. Section 2 overviews CARE. Section 3 presents CARE algorithms for deterministic record and replay. Section 4 elaborates on some important issues including CARE's cache organization, replay guarantee, and heuristics for resolving unordered actions. Section 5 discusses CARE implementation and evaluation, which are followed by related work in Section 6, and finally Section 7 concludes this paper.

2. CARE OVERVIEW

Deterministic replay tools give developers the ability to time-traveling a past execution via logged information at runtime. In this paper, we focus on order-based deterministic replay for failure diagnosis, i.e., to reproduce execution states by topologically sorting partially-ordered actions observed at runtime.

To deterministically replay a concurrent Java program on a multiprocessor machine, we need happens-before orders about shared memory accesses, lock acquisitions/releases and thread synchronizations. As studied before [25], the latter two kinds of orders are relatively fewer and can be easily captured. However, shared memory accesses can be many, and a faithful replay requires preserving the relative order of any two conflicting variable accesses in the observed and replayed executions. Existing order-based work [9, 13, 18, 35] keeps version numbers of shared variables to identify such causal relationships. However, simultaneous accesses to the same version number can occur on a multiprocessor machine. Thus these accesses have to be synchronized explicitly by additional locks or implicitly by atomic operations, which can introduce large runtime overhead at record. Stride [37] addresses this issue by recording inexact version numbers without synchronization and inferring equivalent read-write dependences later at replay, but value fingerprints of variable reads have to be logged. On the contrary, our CARE takes a fundamentally different approach.

The first innovation of CARE is using cache to automatically identify the dependences between shared memory accesses. CARE assigns each thread with a software cache storing buffered variable values, which is updated each time this thread accesses a shared variable (either read or write). The cache is queried each time immediately after a thread reads a shared variable. Cache miss occurs if the read value differs from the one buffered in the cache, and CARE logs a read-write dependence corresponding to this variable access. Let us assume that the cache has an infinite capacity and never discards any buffered value. If a program is sequential (i.e., single-threaded), no cache miss would ever occur, and hence no log is kept. Otherwise, cache miss always indicates that a read value is written by another thread: cache misses successfully identify the read-write dependences. Note that the converse does not necessarily hold as cache hit might occur at reading an inter-thread value, as illustrated in (8)-(9) of Figure 1[3].

When cache miss occurs, one needs to log exact read-write dependence of memory accesses. However, exact dependences can be hard to determine because other writes to the same variable may have already occurred in a short earlier period. To ensure exact

[2]The idea of CARE is general and extensible to other programming languages.

[3]In this paper, we use the visual convention that actions from the same thread are grouped vertically by program order. We use R(x), W(x), ACQ(x), REL(x) to denote read, write, lock acquisition and release on x, respectively. Solid lines are CARE's recorded happens-before dependences, while dashed lines are actual read-write dependences omitted by CARE.

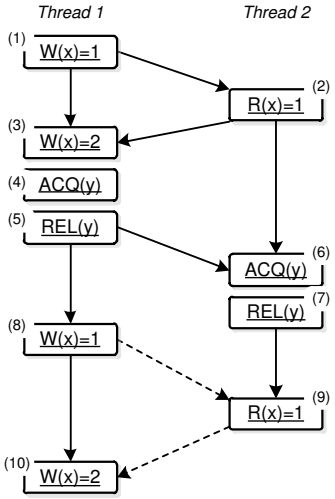


Figure 1: Illustration of missing dependences

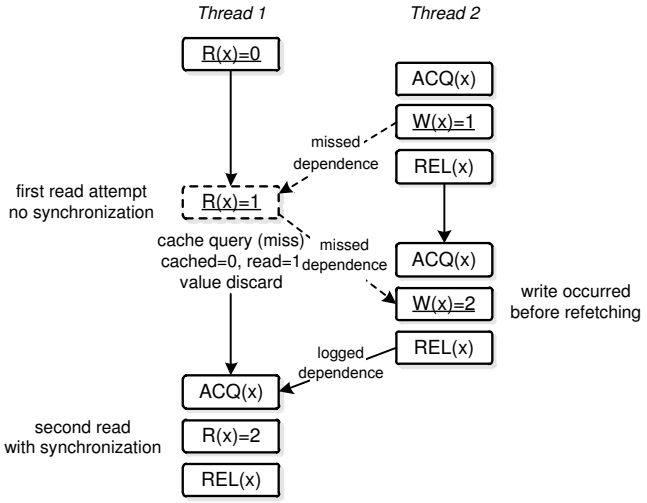


Figure 2: Illustration of refetching

dependences to be logged, CARE would perform the same read actions again (i.e., refetching) with synchronization to identify exact writers corresponding to these read actions, as illustrated in Figure 2. This trick favors programs that are dominated by the thread-local accesses because cache-hit queries incur only tiny overhead and require no synchronization.

The second innovation of CARE lies in its trade-off between log size and replayed trace fidelity guarantee. Recall that cache miss is only a sufficient but not necessary indicator of an exact read-write dependence. As such, not every linear extension of recorded partial orders is necessarily a valid execution. Figure 1 gives such an example. At record, CARE omits the dashed dependences because of cache hit at (9), hence possibly creating linear extension (1)-(2)-(3)-(4)-(5)-(6)-(7)-(8)-(10)-(9) at replay. Such a linear extension is invalid because placing (9) after (10) violates the requisite of sequential consistency. Actually, the CARE approach weakens the replay guarantee to be value-deterministic as the trade-off for reduced log size. At replay, CARE simulates the behavior of every thread's cache to reproduce its exact local behavior. These algorithms are formalized later in Section 3 and discussed in Section 4.

Realizing this drawback, the third innovation of CARE is two powerful heuristics to increase the chance of producing valid execution at replay for debugging. Both heuristics are based on the observation that such limitation is caused by missed read-write dependences. The first heuristic tries to guide the replay scheduler to stop before an unordered read action as early as possible, so that a thread can immediately continue once its target value is written by another thread. The second heuristic is adding extra ordering information to reduce non-determinism by grouping variables into bundles and recording ordering information at a granularity of bundle. Also intuitively, the situation shown in Figure 1 rarely occurs in the execution of real-world programs. For all evaluated benchmarks, CARE successfully resolved all missed dependences and obtained valid executions at replay. Besides, CARE also achieved a significant reduction of log size up to 300%, as compared with the state-of-the-art replay work LEAP [18].

3. CARE ALGORITHMS

In this section, we formalize the CARE instrumentation for collecting happens-before dependences at runtime, and the algorithm synthesizing a value-deterministic execution. We currently focus on the general description of algorithms explaining the principle of CARE. More specific issues such as cache organization, formal proof of replay guarantee and heuristics for execution synthesis are discussed in Section 4.

3.1 Notations

Our execution model is similar to the Java memory model [27]. The basic building block of an execution is an *action*. For action a of kind k performed by thread t accessing variable or monitor object v whose unique identifier is u, we denote it by a tuple $\langle t, k, v, u\rangle$ with the following conventions: (1) We only consider k to be read, write, lock acquisition or lock release, and assume that there is neither dynamic thread creation nor nested lock for brevity[4]. (2) A runtime variable v is uniquely identified by an object reference and a field name. A monitor object v is uniquely identified by its reference. We call both of them *variables* for short. (3) We use $a.t$, $a.k$, $a.v$ and $a.u$ to denote the thread identifier, action type, runtime variable and unique identifier of action a, respectively. (4) The shared memory is denoted by *heap*, which is a mapping from variables to their values. The mapping can be altered by write actions at runtime.

We assume that any execution of a Java program conforms the sequential consistency memory model[5]. Formally, we denote a Java program execution E by a 6-tuple $\langle P, A, V, \overset{po}{\rightarrow}, \overset{sw}{\rightarrow}, \overset{sc}{\rightarrow}\rangle$. P is the program. Note that we assume all threads to be statically created and their identifiers can be uniquely determined by P. A is the set of all actions executed in E. V is the value function indicating the value read or written by a variable access action. For any lock acquisition or release action a, we define $V(a) = \perp$. $\overset{po}{\rightarrow}$ denotes the program order of all actions in A. $\overset{sw}{\rightarrow}$ is the partial order indicating the actual ordering of lock acquisitions and releases happened in E. Every pair of lock actions performed on the same monitor variable is ordered in $\overset{sw}{\rightarrow}$. $\overset{sc}{\rightarrow}$ is a total order on A to define a sequentially consistent execution. We assume that any actual execution E produced by Java runtime is valid, i.e., E meets the following constraints:

[4]For thread fork/join and wait/notify actions in actual executions, we technically treat them as special kinds of lock acquisitions and releases and create corresponding dependences.

[5]Actually, CARE instrumentation enforces the sequential consistency memory model.

Algorithm 1: Instrumentation for $\underline{d \leftarrow heap(v)}$ generating $r = \langle t, \text{read}, v, u\rangle$

```
1 d ← heap(v);
2 if cache_t(v) ≠ d then
3     synchronized v
4         d ← heap(v);
5         H ← H ∪ (last(v), r);
6         G ← G ∪ {r};
7         last(v) ← r;
8     cache_t(v) ← d;
```

Algorithm 2: Instrumentation for $\underline{heap(v) \leftarrow d}$ generating $w = \langle t, \text{write}, v, u\rangle$

```
1 synchronized v
2     heap(v) ← d;
3     if last(v).t ≠ t then
4         H ← H ∪ (last(v), w);
5         last(v) ← w;
6 cache_t(v) ← d;
```

Algorithm 3: Instrumentation for $\underline{acquire(v)}$ generating $\ell = \langle t, \text{acquire}, v, u\rangle$

```
1 acquire(v);
2 if last(v).t ≠ t then
3     H ← H ∪ (last(v), ℓ);
4     last(v) ← ℓ;
```

1. $\overset{sw}{\rightarrow}$ conforms the lock semantic, i.e., for any two paired lock acquisition and release actions on the same monitor variable v, say, (acq_1, rel_1) and (acq_2, rel_2), either $rel_1 \overset{sw}{\rightarrow} acq_2$ or $rel_2 \overset{sw}{\rightarrow} acq_1$ holds.
2. $\overset{sc}{\rightarrow}$ conforms the program order and lock order, i.e., $\overset{sw}{\rightarrow} \subseteq \overset{sc}{\rightarrow}$ and $\overset{po}{\rightarrow} \subseteq \overset{sc}{\rightarrow}$ hold.
3. $\overset{sc}{\rightarrow}$ is sequentially consistent, i.e., for every read action $r \in A$, there is a unique write action $w \in A$ such that $(w \overset{sc}{\rightarrow} r) \wedge (w.v = r.v) \wedge (V(r) = V(w))$, and there is no $w \overset{sc}{\rightarrow} w' \overset{sc}{\rightarrow} r$ for any write action w'.

3.2 CARE Record

At record, CARE instruments a given program's bytecode at class-loading time to collect dependences of its actions. During the program's execution, log $\langle \mathcal{G}, \mathcal{H}, \mathcal{P}\rangle$ is collected for later deterministic replay[6]. $\mathcal{G} \subseteq A$ is the set of all read actions that have ever encountered cache miss in the execution, $\mathcal{H} \subseteq A \times A$ is the inter-thread causal dependences, and $\mathcal{P} \subseteq A \times A$ is the program order.

In order to collect such ordering information, CARE maintains several data structures at runtime: (1) Each thread t is associated with a value prediction cache $cache_t$ keeping buffered values of variables. Formally, $cache_t$ is a mapping from variables to their values. Initially, $cache_t(v) = \perp$ for all v, and $cache_t(v)$ can be reset to $\perp$ before any cache update for arbitrary v. The cache resides in each thread's local storage and is invisible to other threads. Caches of different threads are allowed to have inconsistent values for a variable. (2) Each variable v is assigned with $last(v)$ denoting the unique identifier of the last thread that performs an action on v.

For every Java virtual machine instruction that can generate a non-deterministic action, CARE instruments it to collect dependences. For instructions in thread t generating heap read action $\underline{d \leftarrow heap(v)}$, heap write action $\underline{heap(v) \leftarrow d}$ and lock acquisition action $\underline{acquire(v)}$, their instrumentations are shown in Algorithms 1–3, respectively. In each algorithm, the underlined action is the one being instrumented. These algorithms are explained in detail below.

In Algorithm 1, for each instruction generating read action $\langle t, \text{read}, v, u\rangle$, CARE first reads the value d from $heap$ (Line 1). Then the buffered value $cache_t(v)$ in thread t is fetched from t's value prediction cache for a comparison with d. If $cache_t(v) = d$, CARE optimistically assumes that there is no write action on v since the last update of $cache_t(v)$ in t, and hence no log other than the program order is kept. Otherwise, we have $cache_t(v) \neq d$, indicating that the read value d must be written by another thread. At this time, however, it is already too late to exactly determine which write action is responsible for writing this value, because other threads might write to v during this short period. Instead of recording an inexact dependence and value fingerprint for replay-time inference as Stride [37] did, CARE logs exact read-write dependence by redoing the read action again with the lock of v held (Line 4). Since in Algorithm 2, all writes to variable v are synchronized on the monitor variable v, the second read is serialized with any write to v, and hence its depending write action can be correctly identified and logged in $\mathcal{H}$ (Line 5). Finally in Line 7–8, $last(v)$ and $cache_t(v)$ are correspondingly updated.

In Algorithm 2, for each instruction generating write action $\langle t, \text{write}, v, u\rangle$, CARE wraps it with a synchronization block on the monitor variable v. Log is kept only if $last(v)$ does not happen in thread t, because successive writes in the same thread can be inferred by program order $\mathcal{P}$. $last(v)$ and $cache_t(v)$ are subsequently updated as in the instrumentation of read actions.

In Algorithm 3, for each instruction generating lock acquisition action $\langle t, \text{acquire}, v, u\rangle$, CARE maintains $last(v)$ keeping the latest acquisition action that has successfully entered the monitor. Each time right after v is acquired, CARE checks whether the last acquisition of v is previously performed by another thread. If so, this inter-thread dependence is kept in $\mathcal{H}$.

In summary, the CARE record algorithm logs the write total order, synchronization order and read-write dependences at the cache miss. We discuss several remaining issues below.

First, the record algorithm does not require a specific cache organization. Using an infinite cache that never discards any buffered value is both impractical and unnecessary. Note that using a finite cache does not affect the correctness of the replay algorithm: reducing the cache size only makes cache miss more frequent, incurring extra synchronizations and more detailed log. We further discuss cache organization related issues later in Section 4.1.

Second, the CARE instrumentation keeps actual orders that happen in a recorded execution. Assume that the actual execution is $E = \langle P, A, V, \overset{po}{\rightarrow}, \overset{sw}{\rightarrow}, \overset{sc}{\rightarrow}\rangle$. In the CARE implementation, we use the thread identifier and the sequence number in each thread to uniquely identify an action, thus $\overset{po}{\rightarrow} = \mathcal{P}$ is confirmed. Also since every logged action is correctly synchronized if necessary, $\mathcal{H}$ must conform the actual execution. Particularly, we have $\overset{sw}{\rightarrow} \subseteq \mathcal{H}$ and $\mathcal{H} \subseteq \overset{sc}{\rightarrow}$ guaranteed. Let $\overset{rec}{\rightarrow}$ be all the information of $\mathcal{H}$ and $\mathcal{P}$, i.e.,

[6]The claim that CARE keeps log of $\langle \mathcal{G}, \mathcal{H}, \mathcal{P}\rangle$ is for brevity of presenting our algorithms and proofs. As shown in Section 5.1, we store such logs compactly in each thread's local storage in the implementation of CARE.

Algorithm 4: CARE replay

```
1  S ← A; T ← ∅;
2  while S ≠ ∅ do
3      find a ∈ S such that ∄a' ∈ S and a' -rec-> a;
4      execute thread a.t for one action, obtaining action a';
5      M ← M ∪ (a, a');
6      S ← S \ {a}; T ← T ∪ {a};
7      switch a.k do
8          case read (d ← heap(a'.v))
9              if a ∉ G then
10                 d ← cache_t(a'.v);
11             else
12                 cache_t(a'.v) ← d;
13         case write (heap(a'.v) ← d)
14             cache_t(a'.v) ← d;
```

$\xrightarrow{rec} = tr(\mathcal{H} \cup \mathcal{P})$. Since both $\mathcal{P} \subseteq \xrightarrow{sc}$ and $\mathcal{H} \subseteq \xrightarrow{sc}$ hold, it is clear that $\xrightarrow{rec} \subseteq \xrightarrow{sc}$ is a valid partial order on A.

3.3 CARE Replay

The CARE replayer is a scheduler that controls a program's execution. The replay algorithm suspends a thread when it is about to execute an instruction that can generate an action. When all threads are suspended, the scheduler decides which thread to resume according to $\xrightarrow{rec}$. The algorithm also generates a mapping $\mathcal{M}$. $\mathcal{M}$ maps actions in a recorded execution to the ones in its corresponding replayed execution. $\mathcal{M}$ is only used for proving properties of CARE, and hence is not kept in the actual implementation. We formalize the replay algorithm in Algorithm 4.

To better illustrate the intuition behind Algorithm 4, we consider a special case that every read action r satisfies $r \in \mathcal{G}$. In this case, all reads trigger cache miss and every read on t will be ordered with every write on t in $\xrightarrow{rec}$. Existing work [14] has shown that any linear extension of $\xrightarrow{rec}$ is a valid, failure-reproductive and sequentially consistent execution. Then for this case, Line 7–14 in the algorithm would be totally redundant.

For those $r \notin \mathcal{G}$, no ordering information about r other than its predecessor and successor in thread $r.t$ is kept. There can be interleaving read action $r \notin \mathcal{G}$ and write action w accessing the same variable $r.v = w.v$ such that neither $r \xrightarrow{rec} w$ nor $w \xrightarrow{rec} r$ holds. These unordered actions make obtaining a sequentially consistent linear extension of $\xrightarrow{rec}$ difficult. Nevertheless, CARE is still able to faithfully restore such read values by simulating the cache behavior at replay. Note that Line 6 in Algorithm 1 is guarded by the path condition of $(v \notin cache_t \vee cache_t(v) \neq d)$. Therefore, $r \notin \mathcal{G}$ is sufficient for conducting $cache_t(v) = d$. For a thread's every read action r, making $V(r) = cache_t(v)$ at replay essentially reproduces that thread's execution path. Therefore, CARE simulates the cache behavior, and replaces the return value of every $r \notin \mathcal{G}$ to be $cache_t(v)$.

Observing that $V(r)$ for $r \notin \mathcal{G}$ only depends on $r.t$'s thread-local state: either an earlier read action in $\mathcal{G}$, or a write action. Ordering information for such an action is faithfully kept in $\xrightarrow{rec}$. Executing the program by the order of $\xrightarrow{rec}$ yields the identical cache state in the recorded and replayed executions, and hence CARE is able to provide value-deterministic replay.

However, this replay technique alters the semantic of the program being executed at replay: in a valid Java implementation, any shared memory read should always be fetched from the heap rather than from a thread-local cache. Although the replayed execution's thread-local behavior is identical with the one recorded, the total order of these actions might not satisfy the sequential consistency criteria, causing trouble in figuring out how failures happened and what is the root cause.

In practice, however, this issue is not that serious. With an efficient heuristic of variable grouping at nearly zero cost, the amount of unordered actions in the log of CARE can be largely reduced. We also propose another heuristic to increase the probability of synthesizing a valid execution even if unordered actions exist. These two heuristics are to be discussed in the next section. Actually, this is exactly the key insight of the trade-off in CARE: to have an acceptable relaxation of the replay fidelity guarantee for much lessened read synchronization, reduced log size, linear replay time, as well as synthesizing sequentially consistent executions for all evaluated benchmarks.

4. DISCUSSIONS

Last section gives the algorithms of CARE. We still have some remaining issues with CARE algorithms. First, we have not discussed how the cache organization may impact the performance of logging. Second, CARE keeps log for a read action only if cache miss is encountered. This, however, is insufficient for easily restoring a sequentially consistent execution at replay. In this section, we discuss issues about CARE's cache organization, value-deterministic replay guarantee, sequentially consistent execution synthesis, and performance characteristics.

4.1 Cache Organization

Cache organization may impact CARE's runtime overhead because every shared memory access is associated with at least one cache query or update operation. Using a simple cache can reduce such overhead, however, on the other hand, can incur a higher cache miss rate. Frequent cache misses would lead to redundant read synchronizations as well as redundant logs, which are undesirable. Therefore, cache organization is a trade-off issue.

At one end, cache can be of no use at all: every read action triggers a cache miss. In this case, every shared variable keeps an individual log of all actions accessing it. This becomes exactly the algorithm described in [18]. At another end, we can also use an ideal cache of infinite capacity that never discards any buffered value. Read synchronizations and log size can be reduced to minimal in this setting. However, operations on such a cache is considerably slower than those on simple ones. The most serious issue of such a cache organization strategy is that it virtually disables the garbage collection mechanism, which can quickly drain all memory available for long-running programs. Therefore, we need to set up a cache organization that satisfies: (1) queries and updates are efficient, (2) memory consumption is moderate, and (3) its cache hit rate is comparable to that of an infinite cache. At a first glance, these requirements are conflicting. Fortunately for most real-word programs, memory accesses exhibit very strong locality. A moderate-sized cache readily brings satisfactory performance. We study cache organization and its impact on CARE performance later in the evaluation.

4.2 Value-deterministic Replay Guarantee

As mentioned in Section 3.3, there can be read action r and write action w on the same variable that are not ordered in $\xrightarrow{rec}$. Resolving these unordered actions to obtain a sequentially consistent execution would be very costly. Instead of determining all exact read-write dependences, the CARE replay algorithm simulates the

cache behavior and overrides the read values from heap by values buffered in cache at replay. To prove that such strategy provides value-deterministic executions, we first introduce several notations and definitions. As defined in Section 3.1, an execution is defined by a program, the actions performed, a value function, a program order, a lock order and a total order of actions. Let a recorded execution be $E = \langle P, A, V, \overset{po}{\rightarrow}, \overset{sw}{\rightarrow}, \overset{sc}{\rightarrow} \rangle$ which is sequentially consistent. CARE replay algorithm creates another execution $E' = \langle P, A', V', \overset{po'}{\rightarrow}, \overset{sw'}{\rightarrow}, \overset{sc'}{\rightarrow} \rangle$, but not necessarily sequentially consistent. For two actions $a \in E$ and $a' \in E'$, we define $a \equiv a'$ indicating their equivalence by

$$(a.t = a'.t) \wedge (a.k = a'.k) \wedge (a.v = a'.v) \wedge (V(a) = V'(a')).$$

For actions $a, b \in E$, we define $a \parallel b$ indicating that a and b are interleaving in the logged partial order $\overset{rec}{\rightarrow}$ by

$$\neg(a \overset{rec}{\rightarrow} b) \wedge \neg(b \overset{rec}{\rightarrow} a).$$

For read action $r \in A$, we define its interleaving write set $W(r)$ by

$$\left\{ w \in A \;\middle|\; (w.k = \text{write}) \wedge (w.v = r.v) \wedge (r \parallel w) \right\}.$$

Value-deterministic replay guarantee is reflected by reproducing each thread's local behavior in E'. In Algorithm 4, we have already created mapping $\mathcal{M}$, which associates the actions in E to those in E'. The replay guarantee is characterized by the following theorem:

THEOREM 1. *(1) For all actions $a \in A$, $a \equiv \mathcal{M}(a)$. (2) For all pairs of actions $a, b \in A$, $a \overset{po}{\rightarrow} b$ if and only if $\mathcal{M}(a) \overset{po'}{\rightarrow} \mathcal{M}(b)$. (3) $\overset{sw'}{\rightarrow}$ satisfies the lock semantics.*

PROOF. (Sketch) Prove by induction. Assume that each time at the beginning of an iteration in Algorithm 4, (1) through (3) holds concerning actions in T. Specifically, for all action $a \in T$, $a \equiv \mathcal{M}(a)$, program order $\overset{po}{\rightarrow}$ on T is reflected in $\overset{po'}{\rightarrow}$, and all lock actions in T do not violate the lock semantics.

The inductive hypothesis trivially holds before the first iteration because $T = \emptyset$.

Each time during an iteration, an action $a \in S$ is chosen, and the thread with identifier $a.t$ is executed for one action $a' = \mathcal{M}(a)$. Hence $a.t = a'.t$. The one-one correspondence between threads ensures that program order $\overset{po}{\rightarrow}$ is reflected in $\overset{po'}{\rightarrow}$. Also note that the state of a thread (i.e., stack, registers, thread-local storages, etc.) is deterministic if values of read actions previously performed by the thread are known. According to our inductive hypothesis that for all read actions $r \in T$, $V(r) = V(\mathcal{M}(r))$, we have $a.k = a'.k$ and $a.v = a'.v$. We enumerate the action type $a.k$ to show that the inductive hypothesis still holds after this iteration.

Case 1: a is a lock acquisition or release action. This would never violate the lock semantics because $\overset{sw}{\rightarrow} \subseteq \overset{rec}{\rightarrow}$ and Line 3 in Algorithm 4 ensures that for any actions p and q that $p \overset{rec}{\rightarrow} q$, we have $\mathcal{M}(p) \overset{rec}{\rightarrow} \mathcal{M}(q)$.

Case 2: a is a write. $V(a)$ must be equal to $V'(a')$ according to the inductive hypothesis that each thread's local state is identical in the recorded and replayed executions.

Case 3: a is a read and $a \in \mathcal{G}$. Since all writes on $a.v$ are serialized by lock of $a.v$, the interleaving write set $W(a) = \emptyset$. Let the latest write to $a.v$ before a in $\overset{rec}{\rightarrow}$ be w, $\mathcal{M}(w)$ must also be the latest write to $a.v$ in E'. According to the inductive hypothesis, $V'(a') = V(\mathcal{M}(w)) = V(w)$. By the sequentially consistency assumption of E, we have $V(w) = V(a)$ and hence $V(a) = V'(a')$.

Case 4: a is a read and a $\notin \mathcal{G}$. According to Line 2 of Algorithm 1, We know that $cache_t(a.v) = V(a)$. Since a cached value depends only on the cache's associated thread state, the inductive hypothesis tells that we can easily construct the value of $cache_t(a.v)$ to be $V(a)$, and this value is overridden as $V'(a') = V(a)$.

All possible configurations are enumerated and we thus conclude that $a \equiv a'$ always holds at the end of each iteration. The algorithm terminates when $S = \emptyset$ and $T = A$. At this time, $T = A$ indicates that our theorem is proved. □

The theorem above tells that CARE is able to correctly reproduce all read and write actions, their values and their program order in E. Observable failures such as wrong outputs, assertion violations and uncaught exceptions are guaranteed to be triggered in E'. Still, this is not sufficient for cyclic debugging. Value-deterministic replay can be achieved by logging every read value of the failed thread and feeding them in the replay. However, this gives not enough information about how the failure has happened, especially in the case that the failure is triggered by a sophisticated thread interleaving.

4.3 Synthesizing Sequentially Consistent Executions

It is interesting that CARE records dependences of actions, but only gives value-deterministic replay, while traditional order-based approaches [13, 18] give provable replay guarantee. Intuitively, CARE captures most of "critical" dependences for synthesizing a sequentially consistent execution. An example is replaying data-race-free programs. As shown in [24], for such programs, any linear extension of $\overset{sw}{\rightarrow}$ is sequentially consistent. Since $\overset{sw}{\rightarrow} \subseteq \mathcal{H}$, CARE restores sequentially consistent executions for such programs. Another example is programs that for every variable, all write values to it are distinct. Cache miss now becomes a sufficient and necessary condition indicating that a value being read is certainly from another thread. For such programs, the log collected by CARE contains all exact read-write dependences. Results in [14] suggest that CARE would always restore sequentially consistent executions for such programs.

The intuition above suggests that CARE would mostly replay a sequentially consistent execution. Observing that read-write dependences are hard to determine only when a specific value is back and forth written to the same variable together with writes writing other values in between, at the same time being fetched by a racing memory read (as shown in Figure 1). One can reasonably expect that such hard-to-determine memory accesses are very rare in practice because breaking any part of this condition can make CARE easily synthesize a sequentially consistent execution. This motivates us to formulate an enhanced replay guarantee statement:

THEOREM 2. *CARE can determine orders of interleaving read and write actions to obtain sequentially consistent executions in replay if for any read action $r \in A$, either of the following two conditions holds:*

(1) $\forall w \in W(r)$, $V(w) \neq V(r)$;

(2) $\forall w_1, w_2 \in W(r)$, $V(w_1) = V(w_2) = V(r)$.

PROOF. (Sketch) Since write total order of each variable is contained in $\overset{rec}{\rightarrow}$, we can always find the latest write action w_ℓ writing variable $r.v$ such that $w_\ell \overset{rec}{\rightarrow} r$.

In Case (1), we manually add $r \overset{rec}{\rightarrow} w$ for all $w \in W(r)$. This extra dependence conforms $\overset{sc}{\rightarrow}$ because for any $w \in W(r)$, $w \overset{sc}{\rightarrow} r$ would never be the case, otherwise there must be $w' \in W(r)$ such that $V(w') = V(r)$ according to sequential consistency of $\overset{sc}{\rightarrow}$. This contradicts our assumption that $V(w') \neq V(r)$.

In Case (2), if $V(w_\ell) = V(r)$, r never violates the sequential consistency assumption in any linear extension of $\overset{rec}{\rightarrow}$. In case that

$V(w_\ell) \neq V(r)$, there must exist a write action that writes $V(r)$ between w_ℓ and r in $\xrightarrow{sc}$ and such an action must be in $W(r)$. Let the earliest write in $W(r)$ be w_e, $w_e \xrightarrow{sc} r$ must be the case because $w_\ell \xrightarrow{sc} r$. Therefore we manually add $w_e \xrightarrow{rec} r$. Since all writes in $W(r)$ have the same value written, the ordering issue is resolved. □

The above analysis inspires us to use a heuristic rule giving those reads $r \notin \mathcal{G}$ higher priority so that they can be executed early at replay: if a pair of interleaving read and write action could both be executed at a time, one should try to schedule the read action first because executing the write action might prevent the read from returning the desired value. Therefore to synthesize a sequentially consistent execution, CARE tries to schedule read actions interleaving with other write actions as early as possible. Only in case that the desired value of a read action is inconsistent with the one in the heap, CARE suspends the reading thread and registers a listener on the desired value. Immediately after the desired value is written by another thread, the reading thread is resumed.

Another heuristic used in CARE is based on the observation that interleaving read and write actions can be reduced by recording more artificial dependences in $\xrightarrow{rec}$. This leads to a straightforward strategy that randomly groups variables together into bundles, and collects dependence log at a granularity of bundle. In the actual implementation of CARE, the precedence order is represented by sequence numbers, and variable grouping is equivalent to binding several separate sequence numbers into a single atomic counter. In this setting, the heuristic neither increases the log size nor combines fine-granularity locks into coarse ones.

We evaluate the usefulness of the two heuristics in Section 5. The results show that CARE's practical replay guarantee is quite satisfactory that it successfully produced sequentially consistent executions for all of our benchmarks, including those who contain extensive benign data races.

4.4 Performance

CARE is efficient both at record and at replay. At record, a vast majority of read actions incur little overhead as they require CARE to perform only thread-local cache queries without synchronization. Besides, only a small portion of all inter-thread dependences have to be logged. Now we present a theorem and a series of facts to explain the intuition why CARE is able to generate small log size with low runtime overhead.

THEOREM 3. *Suppose that the cache capacity is infinite. If a variable is accessed by only one thread during a time period, at most one memory access log entry for this variable is kept.*

PROOF. (Sketch) Suppose that variable v does not escape the thread scope during the time period. The first access to v might generate a log entry because $last(v)$ may possibly not refer to the current thread, and $cache_t(v)$ is immediately updated. Since $cache_t(v)$ and $last(v)$ are not altered by other threads, none of subsequent memory accesses needs any log. □

There are also intuitive facts below explaining why CARE achieves both log size and runtime efficiency. Their impacts are presented in the following section of evaluation.

1. Memory accesses exhibit both temporal and spatial locality. Cache precisely identifies successive memory accesses happening in the same thread, reducing both the log size and runtime overhead.
2. A write action causes at most $T - 1$ cache misses, assuming that there are T threads in a program's execution. However in practice, such kind of "broadcasting" is quite rare: most data are shared among only two threads. Together with the phenomenon of memory access locality, cache would be very efficient in reducing synchronization needed for read actions.
3. The CARE algorithms are orthogonal to many of existing widely-used optimization techniques such as static analysis (e.g., escape analysis and race-free analysis), lock grouping, etc. These techniques can be applied for further reducing the runtime overhead and log size.
4. CARE records dependences of actions by logging sequence numbers. These numbers typically follow specific patterns, leading to a low-entropy log. If a log is needed to be archived, one can further reduce the log size by compressing it with a standard tool (e.g., gzip).

Finally, note that finding a linear extension of a partial order can easily be done in linear time. When cache size is fixed, any operation performed on it takes only $O(1)$ time. Moreover, the first heuristic requires only value-listener registration which can also be implemented by hashing in $O(1)$ time. It is thus straightforward that the time complexity of CARE replay algorithm is linear of the trace size.

5. EVALUATION

In this section, we present our CARE implementation and evaluate it with real-world benchmark programs.

5.1 Implementation

We implemented CARE on the basis of our DPAC tool [21], which is an encapsulation of JVM Tool Interface [2] and ASM bytecode transformation libraries [1]. CARE instruments Java bytecode at class loading to collect log at record and to insert breakpoints at replay.

According to Section 3, CARE virtually logs $\langle \mathcal{G}, \mathcal{H}, \mathcal{P} \rangle$ by maintaining each variable's access sequence numbers. An action a of sequence number s means that it is the s-th action performed on variable $a.v$. We store such sequence numbers compactly in each thread's local storage: each time when thread t is about to log a dependence on variable v, v's sequence number is increased and logged in t's local log. Non-conflicting actions are logged in parallel. Each thread's log is a list of tuples $\langle c, s \rangle$. Each tuple denotes a logged action of sequence number s, with c unlogged actions skipped before. Our variable grouping heuristic combines many separate sequence numbers into a single atomic counter. Since updates to sequence numbers are synchronized by CARE's instrumentation, our atomic counters can guarantee to restore actual orders for these separate variables.

At replay, the instrumented program generates a breakpoint each time before an action is actually executed. Breakpoint suspends the current thread. According to Algorithm 4, when all threads are suspended, our scheduler selects an eligible thread to execute for one action. If multiple threads are eligible, the heuristic of action prioritization favors such a thread whose next action r is read and $cache_{r.t}(r.v)$ is identical to the value of $r.v$ in the heap.

We also implemented state-of-the-art deterministic replay work LEAP [18] on the same base as our CARE for comparison. We make sure that the low-level implementation and optimization settings are both identical for comparison fairness. For Stride [37], we did not implement it for comparison because it contains many optimizations yielding very impressive evaluation results, but not available for us. We tried an alternative way by making LEAP as the normalized line to connect the evaluation results of CARE and Stride, in the sense that LEAP and Stride are from the same authors and they have made careful comparisons between them in [37].

Table 1: Comparison of CARE and LEAP under benchmark programs

Benchmark	CARE				LEAP [18]	
	Overhead (×)	*Log Size (/s)*	*Unordered (#)*	*Resolved (?)*	*Overhead (×)*	*Log Size (/s)*
Avrora	1.52	2.18MB	23K	Y	9.48	24.3MB
Batik	1.49	1.51KB	0	Y	3.77	2.32KB
H2	18.5	24.2MB	0	Y	62.8	27.4MB
Lusearch	3.41	6.53MB	0	Y	9.01	46.0MB
Sunflow	64.9	886MB	0	Y	389	6029MB
Tomcat	4.76	7.80MB	15	Y	11.9	23.5MB
Xalan	7.18	13.6MB	0	Y	12.2	143MB
Tsp	2.79	1.84MB	0	Y	111	570MB
Moldyn	11.9	24.1MB	0	Y	50.5	303MB

5.2 Evaluation Results

5.2.1 Experimental Settings

Our experiments were conducted on a Dell PowerEdge server running Linux Kernel 3.2 with dual 6-core Intel Xeon E5645 processors and 48GB memory. We disabled the Turbo Boost and the Hyper-threading technologies to alleviate unpredictable performance thrashing.

Following existing replay work [18, 37], we chose a wide range of benchmarks from Dacapo suite, including concurrent programs on network simulation (Avrora), vector imaging (Batik), database transactions (H2), text search (Lusearch), graphics rendering (Sunflow), web server (Tomcat) and document processing (Xalan). We also chose two scientific benchmarks used in [37][7]: Tsp solving the travelling salesperson problem and Moldyn simulating molecular dynamics. We used available default settings for all benchmarks in our experiments.

In the following, we first validate CARE's main claims that it can reduce the record-time overhead and log size, at the same time synthesized valid executions at replay. We then study specific details of CARE:

1. How does cache organization impact CARE's runtime overhead and log size?
2. How does variable-grouping heuristic impact CARE's runtime overhead?

5.2.2 Runtime Overhead and Log Size Reduction

Our evaluation results of CARE and LEAP are presented in Table 1. Columns 2 and 6 present the runtime overhead of CARE and LEAP, and Columns 3 and 7 present their log size. All runtime overhead and log size data are normalized based on the running time of non-instrumented executions (i.e., normalized log size per second is the total log size divided by the execution time of a non-instrumented execution). Columns 4 and 5 present evaluation results of our heuristics for synthesizing sequentially consistent executions. The amount of unordered conflicting memory access pairs in the log is shown in Column 4, while results in Column 5 indicate whether CARE successfully resolved these ordering issues for synthesizing sequentially consistent executions (Y denotes "successful resolve").

Regarding the record cost for the Dacapo suite benchmarks, CARE exhibited on average 2.6× and 4.9× reduction on runtime overhead and log size, respectively over LEAP. For scientific benchmark Tsp, CARE was even 40× faster and incurred 300× smaller log. This is because threads in Tsp frequently read from shared memory without any lock protection. LEAP had to serialize all these accesses, while our CARE did not have to because cache hit is almost always the case. Similar situation occurred to Moldyn, and CARE also outperformed LEAP.

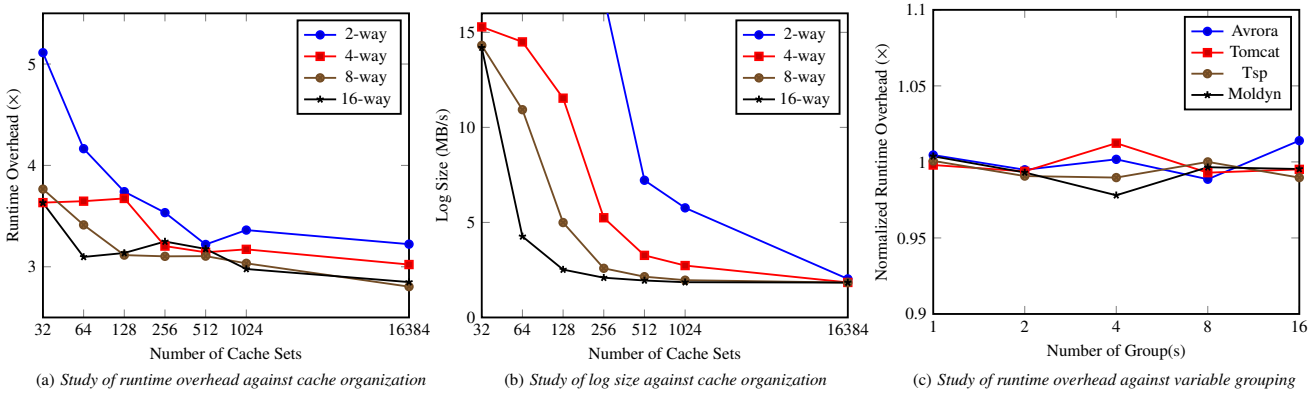
[7] Derby, SpecJBB and ICE evaluated in [37] were not included in our evaluation because they are neither available to us nor contain available default settings.

Table 2: Comparison of CARE and Stride with normalized values

Benchmark	CARE		Stride [37]	
	Overhead	*Log Size*	*Overhead*	*Log Size*
Avrora	16.0%	8.97%	54.0%	36.4%
Batik	39.5%	65.1%	50.0%	34.9%
H2	29.5%	88.3%	29.8%	23.9%
Lusearch	37.9%	14.2%	34.7%	30.0%
Sunflow	16.7%	14.7%	38.5%	9.17%
Tomcat	40.0%	33.2%	64.3%	34.6%
Xalan	59.2%	9.52%	19.0%	23.1%
Tsp	2.51%	0.32%	9.36%	7.18%
Moldyn	23.8%	7.94%	1.32%	0.71%

We also observed that CARE's heuristics worked well in synthesizing sequentially consistent executions because the evaluated benchmarks are almost correctly synchronized, and there were not many actions left unordered.

As mentioned earlier, we indirectly compare our CARE with Stride [37] based on LEAP [18] by aligning two LEAP implementations' data. We had to do so as two LEAP implementations performed quite differently for different benchmarks for unknown reasons. For benchmark Avrora, Lusearch and Xalan, our LEAP implementation incurred much smaller runtime overhead than reported in [37]. On the contrary, for Tomcat, H2 and Sunflow, LEAP's log size reported in [37] was much smaller than ours. We conjecture that this is due to the optimizations used in [37], which can benefit some programs but may harm the others. After LEAP data alignment (LEAP data are all normalized to 1), we obtain results in Table 2. According to absolute comparisons (in Table 1 and in [37]) and relative comparisons (in Table 2), we can safely conclude that the current CARE and Stride implementations have comparable record costs. For some benchmarks, CARE outperformed Stride but for some other benchmarks, Stride beat our CARE. Nevertheless, we note that Stride has done extensive engineering optimizations like escape analysis (introduced by one whole page in [37]) to additionally reduce runtime overhead and log size. Therefore, we believe that our CARE still has the potential to be further improved by additional engineering efforts.



(a) *Study of runtime overhead against cache organization* (b) *Study of log size against cache organization* (c) *Study of runtime overhead against variable grouping*

Figure 3: Impacts of Cache Organization and Variable Grouping

5.2.3 *Impact of Cache Organization*

To study the impact of cache organization, we studied software-implemented k-way set associative cache with n sets (from 2 sets to 16 sets) using the widely-used LRU strategy for replacement. We conducted the study on the Tsp benchmark because it contains frequent racing memory reads, realizing that cache hit can largely affect the performance of such benchmarks. Figures 3(a) and 3(b) show the study results. We observe that increasing cache size can greatly reduce the runtime overhead and log size when the cache size is small (e.g., less than 512 sets). When the cache is larger than a moderate size (e.g., 1024 sets), the improvement becomes less significant and such a moderate size can be easily deployed on a commodity computer. Studies of other benchmarks also give similar results, and we therefore omit their details. From these results, we conclude that using a moderate-sized cache is already sufficient for reducing the record cost in CARE.

5.2.4 *Impact of Variable Grouping*

As mentioned earlier, our variable-grouping heuristic succeeded in synthesizing sequentially consistent executions, but one may wonder its accompanied cost. According to [37], a Global approach that serializes all write actions can outperform LEAP and even has comparable performance with Stride in many evaluated benchmarks. This result gives a clue in concluding that our heuristic could be efficient in practice, because even the most aggressive variable grouping strategy that groups all variables into a single bundle still allows write actions to different variables to be executed in parallel, which is much more efficient than Global.

We study the impact of variable grouping in Figure 3(c) and observe from it that the cost of variable grouping is almost negligible for our four benchmarks, as compared to the setting of no variable grouping. This is mainly because CARE generates log only at cache miss. In a commodity processor, executing millions of atomic operations only requires milliseconds of time, and its out-of-order execution mechanism does not even suspend a thread if it just waits for an atomic operation to complete. Therefore, we can safely use the most aggressive variable grouping strategy, i.e, grouping all sequence numbers into a single atomic counter, to achieve a higher probability of reproducing sequentially consistent executions. In fact, we use such a strategy to produce data in Table 1.

6. RELATED WORK

Deterministic replay tools can either be implemented purely at an application level, within a virtual machine monitor, or even with assistance of customized hardware [4, 5, 9, 16]. CARE is an application-level and software-only work, but adopts experiences from approaches of other categories.

In a uni-processor system, deterministically replaying an execution is readily solved because logging thread preemptions and external inputs essentially eliminates non-determinism for replay [10, 12]. However, in a multi-processor setting, racing memory accesses make deterministic replay much more challenging: diverse execution paths emerge from interleaving data races. According to the fundamental approach of taming non-deterministic racy memory accesses, existing pieces of work can be roughly categorized into search-based ones and order-based ones.

Search-based work [3, 19, 29, 33, 36] takes a different fundamental approach than CARE does, which logs only selective information at runtime and synthesizes a valid execution later at replay. Logging strategies for search-based work can be very optimistic such that runtime overhead and log size can be largely reduced. Aggressive log reduction, on the other hand, makes finding a valid execution computationally intractable. Search-based work typically only gives best-effort replay guarantee, or the success of replay depends on powerful state space exploration engine such as an SMT solver.

In contrast, order-based work[8] [9, 13, 18, 35, 37] carefully keeps the actual happens-before order of executed actions, obtaining a partial order over these actions. Failure reproduction is generally achieved by executing a program in guidance of the topologically sorted order of the logged actions of this program. To distinguish the precedence of variable accesses, order-based work has to instrument a program with extra synchronizations serializing racing memory accesses. CARE also belongs to this category.

Despite the fact that all order-based work shares a similar idea of enforcing CREW semantics and recording memory access orders, there are still subtle trade-offs for reducing runtime overhead, log size and replay cost. To the best of our knowledge, CARE is the first order-based deterministic replay technique striking the balance that both runtime overhead and log size are reduced, replay is done at linear time, and replay fidelity is only slightly weakened.

In order-based deterministic replay, logs for successive thread-local memory accesses can be combined into a single entry, thus reducing the log size. Such thread locality can be detected by a value prediction cache, as used in iDNA [8] and Stride [37]. CARE further observed the cache's ability of finding inexact causal depen-

[8]Stride [37] considers itself search-based, but we classify it as order-based because it records inexact read-write dependences and perform replay by topologically sorting the recorded actions.

dences between memory accesses. Compared with LEAP [18], its core algorithm can be considered as a special case of CARE that triggers cache miss at every memory read. The technique of refetching read value at cache miss in CARE is similar with the one presented in ReEmu [9]. ReEmu uses atomic counters and memory fences to guard shared memory reads, which incur implicit synchronizations. In contrast, CARE's cache requires no extra inter-processor synchronization at cache hit. In reducing synchronizations, one piece of ingenious work is Stride which eliminates all read synchronizations by logging inexact version numbers, and ensures strong replay guarantee. Stride tackles the problem of runtime overhead, but not log size. Inference of exact read-write dependences requires logging value fingerprint for every read, and this substantially increases the log size and suffers from hash collisions. CARE logs no such value and hence does not have such issues. Also note that extra synchronization of CARE is within a small constant factor than that in Stride in practice, because CARE needs synchronized read only when cache miss is encountered.

All these order-based deterministic replay techniques have attractive replay guarantee, but with less satisfactory record-time overhead. To reduce this overhead, static determinism analysis, transitive reduction and domain-specific optimizations are proposed:

1. Examining non-racing memory accesses unnecessarily slows down the instrumented execution. Runtime overhead can be greatly reduced if these accesses on these variables are not instrumented, and this is typically achieved by a static program analysis [18, 37]. CARE is orthogonal to these deterministic analyses such as escape analysis and lock-set analysis. In extreme end of determinism analysis, Chimera [25] statically converts racy programs to equivalent race-free ones by adding extra synchronizations. Compared with Chimera, CARE tries to make racing reads as parallel as possible. For programs with intensive intended racing reads of shared variables, CARE would have less runtime overhead.
2. The key idea of order-based deterministic replay is logging causal dependences between read and write actions. However, logging all of them is of course not a minimal setting. The problem of transitive reduction is addressed in [20, 26, 28, 34]. Actually, the cache in CARE also does an approximate transitive reduction with very low runtime overhead, removing some redundant logs. We can adopt existing transitive reduction techniques for even smaller logs.
3. For specific application domains, performance of deterministic replay can be further tuned. For example, replaying Android applications with GUI interactions is studied in [15]. Debugging non-determinism caused by external inputs in libraries is discussed in [6]. They are out of our scope and thus not discussed further.

To replay concurrent executions, one can also adopt deterministic execution [11] that ensures a determinsitic execution environment. However, for general applications running on generic operating systems, eliminating all non-determinism is essentially too costly in the absence of customized hardware's assistance [7, 17].

Finally, deterministic replay technology is the cornerstone of numerous testing and debugging tools for concurrent programs. Examples include cyclic debugging [23], data race detection [30, 31], deadlock prediction [22], to name but a few. We believe that improvements in deterministic replay can considerably benefit these kinds of work.

7. CONCLUSION

In this paper we present CARE, a cache guided deterministic replay technique. By utilizing cache miss as the indicator of recording necessary memory access orders, CARE reduces a majority of additional synchronizations for read actions, runtime overhead and log size compared with conventional techniques, while still offering reasonable replay guarantee for debugging. Our experimental results show that CARE's generated log can be several orders of magnitude smaller than state-of-the-art order-based replay techniques.

To the best of our knowledge, our work is the first one trying to trade slightly weakened replay guarantee for much reduced record cost in deterministic replay. We also note that this trade-off, though might potentially create invalid traces at replay, works fine in practice with the aids of our two replay heuristics. Hence it would be interesting to see whether it is possible to further extend this central trade-off idea in CARE to more practical systems (e.g., systems with relaxed memory models).

8. ACKNOWLEDGMENTS

This work was supported in part by National High-tech R&D 863 Program (2013AA01A213) and National Natural Science Foundation (61100038, 91318301, 61321491, 61361120097) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

9. REFERENCES

[1] ASM toolkit for bytecode manipulation. `http://asm.ow2.org/`.

[2] JVM tool interface. `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[3] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP, pp. 193–206, 2009.

[4] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD, pp. 194–206, 1991.

[5] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS, pp. 359–368, 2011.

[6] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pp. 362–371, 2013.

[7] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS, pp. 53–64, 2010.

[8] 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. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE, pp. 154–163, 2006.

[9] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP, pp. 207–218, 2013.

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

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS, pp. 85–96, 2009.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI, pp. 211–224, 2002.

[13] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE, pp. 121–130, 2008.

[14] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, Aug. 1997.

[15] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pp. 72–81, 2013.

[16] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive application-level record-replay for replay parallelism. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS, pp. 193–206, 2013.

[17] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA, pp. 333–334, 2011.

[18] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE, pp. 207–216, 2010.

[19] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pp. 141–152, 2013.

[20] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE, pp. 57–66, 2010.

[21] Y. Jiang, C. Xu, and X. Ma. DPAC: An infrastructure for dynamic program analysis of concurrency Java programs. In *Proceedings of the 2013 Middleware Doctoral Symposium*, 2013.

[22] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pp. 110–120, 2009.

[23] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC, 2005.

[24] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, C-36(4):471–482, 1987.

[25] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, pp. 463–474, 2012.

[26] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward generating reducible replay logs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pp. 246–257, 2011.

[27] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pp. 378–391, 2005.

[28] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, PADD, pp. 1–11, 1993.

[29] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP, pp. 177–192, 2009.

[30] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pp. 11–21, 2008.

[31] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pp. 387–400, 2012.

[32] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP, pp. 131–144, 2007.

[33] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS, pp. 155–166, 2010.

[34] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS, pp. 49–60, 2006.

[35] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object centric deterministic replay for java. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ATEC, pp. 30–43, 2011.

[36] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys, pp. 321–334, 2010.

[37] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE, pp. 892–902, 2012.