Light: Replay via Tightly Bounded Recording

Peng Liu
Purdue University, USA
peng74@purdue.edu

Xiangyu Zhang
Purdue University, USA
xyzhang@cs.purdue.edu

Omer Tripp
IBM Research, USA
otripp@us.ibm.com

Yunhui Zheng
IBM Research, USA
zhengyu@us.ibm.com

Abstract

Reproducing concurrency bugs is a prominent challenge. Existing techniques either rely on recording very fine grained execution information and hence have high runtime overhead, or strive to log as little information as possible but provide no guarantee in reproducing a bug. We present Light, a technique that features much lower overhead compared to techniques based on fine grained recording, and that guarantees to reproduce concurrent bugs. We leverage and formally prove that recording flow dependences is the necessary and sufficient condition to reproduce a concurrent bug. The flow dependences, together with the thread local orders that can be automatically inferred (and hence not logged), are encoded as scheduling constraints. An SMT solver is used to derive a replay schedule, which is guaranteed to exist even though it may be different from the original schedule. Our experiments show that Light has only 44% logging overhead, almost one order of magnitude lower than the state of the art techniques relying on logging memory accesses. Its space overhead is only 10% of those techniques. Light can also reproduce all the bugs we have collected whereas existing techniques miss some of them.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Design, Performance, Reliability

Keywords Concurrency, Replay, Thread Determinism, Tight Bound, Local Recording

1. Introduction

Concurrent programs are becoming increasingly popular. While concurrency has clear performance advantages, it also introduces nondeterminism due to thread interleavings. Different runs of the same concurrent program may take different execution paths.

A prominent challenge that arises is to reproduce a bug in a concurrent execution. Simply rerunning the program is not guaranteed to (and in fact often doesn't) reproduce the detected problem, which complicates debugging and patching of concurrent software.

Existing Approaches In light of this challenge, a large body of research has been directed toward creating *replay* algorithms [14, 15, 22, 28, 37]. The underlying idea is to insert instrumentation code into the program to record certain runtime events (e.g., memory accesses or branching decisions), thereby enabling re-execution of the original run to reproduce a bug.

Existing replay tools strike different trade-offs between the recording overhead and the reproduction guarantees they provide. Some of the tools lean toward very low recording overhead (3%-15%) but sacrifice the guarantee that the replay run follows the same path and uses the same values as in the original run [22, 28, 37].

Other tools guarantee the reproduction of execution, also known as *replay determinism*. This is done through two main types of approaches.

Record-based algorithms [14, 40] record concrete accesses to shared memory, thereby enabling the recovery of the exact execution schedule. While existing approaches are robust, they result in prohibitive recording overhead (up to 20X) for two main reasons: (1) they record too much information, and (2) they introduce extensive synchronization for the purpose of recording.

Another alternative, known as the *computation-based* approach [15, 39], is to record little runtime information (e.g., only branch outcomes) while delegating the bulk of schedule reconstruction effort to an offline (solver-backed) symbolic analysis. This is an appealing strategy, yielding affordable recording overheads (20%-50%), but unfortunately it suffers from narrow applicability due to solver limitations. The offline symbolic analysis needs to reason about program states in different schedules, which further requires the computation of expression values. State-of-the-art solvers, like Z3 [8] and Yices [38], are limited in handling complex or non-linear arithmetic computations that manifest frequently in real-world software. We have confirmed this limitation over a suite of real-world applications. 63% of the real bugs (evaluated in Section 5) are outside the scope of the advanced computation-based algorithms.

Our Approach The goal we set in this paper is to formulate a replay approach that simultaneously (i) has the determinism guarantee and (ii) offers practical overhead and scalability. In light of the inherent limitations of computation-based replay, we turn to the record-based direction in search of such an approach.

The main challenge with record-based replay is to battle the high overhead. We battle the overhead in the following ways:

- First, we establish a *precise* (i.e., both necessary and sufficient) bound on recording. We formally prove that recording only the flow dependence (rather than anti dependence or output dependence) is sufficient to reproduce the execution. Failure to record the flow dependence compromises the replay determinism guarantee.
- We design a lightweight thread-local recording scheme, which records flow dependences in thread-local buffers without synchronization. Comparatively, previous record-based approaches [14, 40] record data in global sequences, facilitated by synchronization. The data recording is expensive, e.g., it manipulates or even resizes the complex data structure. By making it local, we avoid the high synchronization overhead.

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2738001

- To gain further performance improvement, we apply two optimizations to reduce the intra-thread and inter-thread flow dependences. First, the thread-local sequence of flow dependences that is not interleaved can be reduced from the recording as the thread-local order preserves them. Second, the inter-thread flow dependences between the accesses well protected by locks do not need to be recorded as the recorded lock operation orders ensure them.

The replay system computes, based on the SMT solver, the full schedule of a feasible replay run that preserves the flow dependences. We stress that, different from the computation-based approach [15, 39], our replay system does not need to symbolically reason about the computation of expression values. Hence, our replay system is not affected by the aforementioned solver limitations.

We implement a replay tool called Light and evaluate it on 24 benchmarks including the large ones from the Dacapo benchmark suite. We measure the runtime recording overhead and the space consumption and compare them with existing representative record-based approaches [14, 40]. The evaluation result shows that Light incurs 44% overhead on average, while existing approaches incur around 4X on average. Our space consumption is only 10% of that of existing approaches. We also validate our replay determinism guarantee on top of 8 real world bugs. Light succeeds in replaying all of them, while the computation-based approach [15, 39] fails to reproduce 5 of them and a purely lock-based approach [21] fails to reproduce 3 of them.

2. Overview

In this section, we walk the reader through the main highlights of our approach with reference to the running example in Figure 1.

2.1 Running Example

The code snippet in Figure 1, extracted from the popular Cache4j application, captures the interaction between two methods: `put(...)` resets an existing `CacheObject` instance by setting its `_createTime` field to the current time (in the callee method `resetCacheObject(...)`). `get(...)` utilizes the same field to check the validity of a retrieved `CacheObject` instance (in the callee method `valid(...)`).

Profiling runs of Cache4j using realistic workloads (from actual clients) often exhibit an access pattern to `_createTime` as presented in Figure 2, whereby one thread t_1 executes `put(...)` multiple times (e.g. 10 times); then another thread t_2 invokes `get(...)` several times (e.g. 10 times); then the thread t_1 again calls `put(...)`, and so on.

2.2 Optimality: Tight Recording

A conservative way to reproduce an execution is to record the happens-before access order for every shared memory location. This approach, featured e.g. by the Leap system [14], amounts to maintaining an access sequence per memory location, using data structures such as vectors, as illustrated in Figure 2.

This incurs significant *performance* and *space* overhead. First, for each shared location, *all* accesses are stored into a vector, which essentially encodes different types of dependences. Second, recording of memory accesses is governed by synchronization. As such, the recorded order correctly reflects the access order. Due to the complexity of the recording, which manipulates or even resizes the vector, synchronization overhead is very high. Specifically, Leap incurs 3X overhead.

Our first contribution is to state the precise condition for deterministic replay. We formally prove that logging *flow* (i.e., read/write) dependences is a necessary and sufficient condition to recreate a bug due to use of an illegal value. Importantly, neither *output* (i.e., write/write) dependences nor *anti* (i.e., write/read) dependences need be recorded.

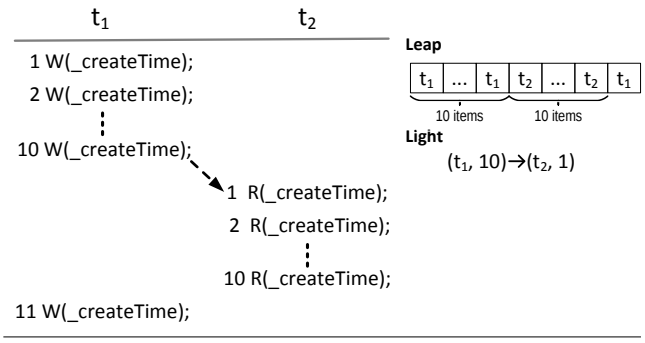


Figure 2. Fragment from an execution trace of Cache4j

Through enforcement of flow dependences, our approach guarantees to yield the same value at each use point, thereby reproducing the buggy behavior. The precise condition is the basis for our light-weight thread-local recording scheme (Section 4.1) as well as certain optimizations (Section 4.3).

2.3 Record and Replay

Having established that only flow dependences need be tracked, we now describe the design of our record and replay algorithm. The primary property we exploit is that it suffices to maintain the last write access to a shared memory location such that when the location is read, the flow dependence is recorded. This is illustrated in Figure 2 with $(t_1, 10) \rightarrow (t_2, 1)$. Compared to the vector of size 21 maintained by Leap, this scheme features substantial improvement in both space and time. Note that we adopt the thread-local index to denote each access, e.g., $(t_1, 10)$, where t_1 refers to the thread and 10 is the value of the thread-local counter.

We record the flow dependence (detected online) in a thread-local buffer of the reading thread (i.e., t_2) without the need for synchronization. Dependence recording is much more expensive than the detection as it needs to manipulate or even resize the container data structure. By making such expensive operations thread-local, we significantly reduce the synchronization overhead. Comparatively, data recording needs to be made synchronous in other approaches [6, 14].

The detection of flow dependence needs synchronization. Specifically, we update the last write to a location ℓ atomically, as well as perform atomic matching between the write and a subsequent read. The good news is that both atomic operations are simple and block other threads for very short time. The simple update ($\mathsf{last}_\ell = n$) is placed in the same atomic section with the shared access from program. For the atomic write/read matching, which happens during a shared read, we have found that the optimistic retry loop is highly effective, yielding few retries in practice.

```
retry :  n1 := last_ℓ
         {R(ℓ)}
         n2 := last_ℓ
         if n1 ≡ n2     record n1 → nr
         else           goto retry
```

In particular, the operation is to record the flow dependence for a shared read to location ℓ, with n_r the index or counter value of the read. After the read, the routine further checks if the last write to ℓ stays the same, to ensure atomicity.

A subtle point is how to schedule, during the replay run, shared memory accesses such that different flow dependences involving the same memory location do not interfere with each other. Otherwise, it may yield undesirable results. For example, assume we have two

```
synchronized  put(objId, obj){
  CacheObject co = _map.get(objId);
  if(co!=null) {
    ...
    resetCacheObject(co);
  }
}
resetCacheObject(co){
  co._createTime = … ;
}
```

```
synchronized Object get(objId){
  CacheObject co = _map.get(objId);
  if(co!=null){
    if(!valid(co)) {...}
  }
}
boolean valid(co){
  return co._createTime>now()...;
}
```

Figure 1. Fragment from the Cache4j application

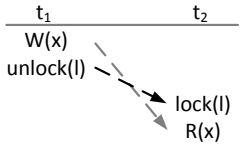


Figure 3. Reduction

flow dependences, $W_1 \rightarrow R_1$ and $W_2 \rightarrow R_2$. Randomly scheduling them may yield an interleaving such as $W_1\ W_2\ R_2\ R_1$, which assigns R_1 the value written by W_2 and breaks the dependence $W_1 \rightarrow R_1$.

Light addresses this challenge by encoding ordering constraints imposed by flow dependences, as well as noninterference constraints, as a constraint system (Section 4.2). The system is then solved offline, yielding a provably correct and feasible scheduling strategy for shared accesses in the replay run.

The write to a shared location is commonly followed by multiple reads from one thread t: $W\ R_1^t\ \ldots\ R_n^t$. In this case, only the first dependence $W \rightarrow R_1^t$ is recorded by Light, whereas the remaining dependences are inferred from it (Section 4.1).

With this overall scheme, Light incurs 1.2X overhead, i.e., only 1/3 of that incurred by Leap.

2.4 Optimizations

Our approach features two optimizations, one for reducing the inter-thread flow dependences and the other for reducing the intra-thread flow dependences.

First, real world concurrent programs often ensure mutual exclusive accesses to shared locations [13, 18]. If a location ℓ is only accessed within synchronized sections protected by the same lock, recording the order of these sections (i.e., the lock operation order) obviates the need to record the order at the access level. Consider Figure 3, we do not need to record the flow dependence over `x` as the lock operation order enforces it. Pleasingly, by modeling the lock operations as shared accesses (Section 4.3), we naturally capture the lock operation order as flow dependences. In our running example, both `put(...)` and `get(...)` are `synchronized`, recording their execution orders obviates the need to record the flow dependence over `_createTime`.

The other optimization is to reduce the recording of thread-local sequence of flow-dependences if the sequence is not interleaved by an access of the same location. Such pattern also commonly exists. Section 4.3 explains the details.

With our optimizations, we were able to lower the overhead from 1.2X to around 30% for the running example.

3. Necessary and Sufficient Recording

In this section, we state necessary and sufficient conditions for correct replay, which are strictly weaker than global determinism.

3.1 Preliminaries: Execution Model

As is standard, we assume an interleaved semantics of concurrency [27, 33, 34], whereby parallelism reduces to nondeterministic thread scheduling. We assume the following standard semantic domains:

$$\begin{array}{rcl} T & \subseteq & \mathcal{T} \\ L & \subseteq & \mathcal{O} \\ V & \subseteq & Val = \mathcal{O} \cup \{\mathsf{null}\} \\ \rho & \in & Env = \mathcal{T} \times \mathsf{VarId} \rightarrow Val \\ H & \in & Heap = \mathcal{O} \times \mathsf{FldId} \rightarrow Val \\ \sigma = (T, L, \rho, H) & \in & \Sigma = \mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{O}) \times Env \times Heap \end{array}$$

T is a set of threads, where each $t \in T$ (with unique label ℓ_t) undergoes reduction throughout execution. The system terminates when every $t \in T$ is reduced to skip. L is a set of objects. ρ and H are the thread-local environment and global heap, respectively. A state is a quadruple consisting of the running threads, the set of live objects, an environment and a heap. We use dot notation to access each state component (e.g. $\sigma.H$).

We present three reduction rules of interest in our scope, where the first represents non-deterministic scheduling, and the rest two represent field read and field write:

$$[\text{NoDet}]\ \frac{(\{t\}, L, \rho, H) \xrightarrow{\mathsf{s}} (\{t'\}, L', \rho', H'), t \in T}{(T, L, \rho, H) \xrightarrow{t:\mathsf{s}} (T[t \mapsto t'], L', \rho', H')}$$

$$[\text{RdFld}]\ \frac{\sigma \models o \in L, (o, \mathsf{f}) \in \mathsf{dom}\ H, \rho(t, \mathsf{v}) = o}{(\{\mathsf{u} = \mathsf{v.f}\}, L, \rho, H) \longrightarrow (\{\mathsf{skip}\}, L, \rho[(\ell_t, \mathsf{u}) \mapsto h(o, \mathsf{f})], H)}$$

$$[\text{WrFld}]\ \frac{\sigma \models o \in L, (o, \mathsf{f}) \in \mathsf{dom}\ H, (t, \mathsf{v}) \in \mathsf{dom}\ \rho}{(\{\mathsf{u.f} = \mathsf{v}\}, L, \rho, H) \longrightarrow (\{\mathsf{skip}\}, L, \rho, H[(o, \mathsf{f}) \mapsto \rho(\ell_t, \mathsf{v})])}$$

We abstract away array accesses from the formalism for conciseness. We stress, however, that we have full support for array accesses, which we treat analogously to field accesses in our implementation. The operational semantics for the other rules are equally standard [34].

A trace tr is a sequence of transitions, where each transition $\tau \in tr$ represents the execution instance of a statement. The statement may be a field read, field write or a local access (i.e., a read/write of a local variable). Besides, we assume each statement has a simple format (e.g. three-address code). This assumption is made only for ease of presentation, without loss of generality. For example, the compound statement $\mathsf{print\ x.f}$ reduces to $\mathsf{y = x.f;\ print\ y}$.

For each transition τ, we use $\tau.t$ to denote its respective thread t, and $\tau.loc$ to denote the involved memory location if it is a field read/write. We denote the index of τ in tr as $\mathsf{index}\ tr\ \tau$, or simply $\mathsf{index}\ \tau$ if tr is clear from the context.

3.2 Flow Dependences

Definition 3.1 (Flow Dependence)**.** *Let tr be an execution trace, $\tau_r \in tr$ a [RdFld] transition and $\tau_w \in tr$ a [WrFld] transition. We*

say that there is a flow dependence *between τ_w and τ_r, denoted as $\tau_w \rightarrow \tau_r$, iff*

1. **(same field)** *the memory location o.*f *written by τ_w is identical to that read by τ_r: $\tau_r.loc \equiv \tau_w.loc$;*
2. **(adjacency)** index $\tau_w <$ index τ_r*, and any intermediate transition between τ_w and τ_r of type [WrFld] does not involve $\tau_r.loc(\equiv \tau_w.loc)$.*

Though the above definition refers to (global heap) memory locations in general, we are in fact interested only in shared locations (i.e., those that can potentially be accessed by more than one thread). Restricting the replay algorithm only to shared locations is a natural yet significant performance optimization. Detection of shared locations is featured by static analysis libraries like Soot [35] and Chord [26]. In our implementation, we use Soot.

Assumption 1 (Thread Determinism)**.** *If the sequence of [RdFld] transitions in a thread have deterministic values, the thread execution is deterministic. That is, two executions from the same initial state σ_0 that receive the same values in [RdFld] transitions have the same intermediate and final thread local states.*

Note that here we do not assume that the schedule is deterministic. Even with a different schedule, if a thread goes through the same sequence of global read values [1], its execution is deterministic.

To handle system calls that change their output across executions (e.g. time()), we record the value of the call in the original run and replace the call with the recorded value in the replay run.

Definition 3.2 (Buggy Usage)**.** *The bugs of interest are those that arise due to use of local variables that are assigned illegal values.*

Here, the term "illegal value" has different meanings for different bug types. For example, a divide-by-zero exception arises at statement z=x/y if the value of y is 0; a null-pointer exception arises at z=x.f if the value of x is null; and an assertion violation arises at assert (x>0) if the value of x is 0 or less. As these examples illustrate, our characterization of the bugs of interest captures a broad category of bugs. Our approach guarantees reproduction of the same value at each use, thereby guaranteeing reproduction of the bugs of interest (Theorem 1).

Definition 3.3 (Correlated Transitions)**.** *We say that the transition τ in the record run and the transition τ' in the replay run are* correlated *if they share the same thread as well as the same thread-local counter value.*

Theorem 1 (Tight Replay Bound)**.** *Let tr be an execution trace (starting with state σ_0) that exhibits flow dependences $deps$ (denoted as $tr \models deps$), and tr' an execution trace (starting with σ_0) that preserves the same flow dependences ($tr' \models deps$). For transition $\tau \in tr$ that involves statement* s *and reads an illegal value v (per Definition3.2), its correlated transition $\tau' \in tr'$ is guaranteed to exist, and involves the same statement* s *and reads the same value v (thereby replaying the bug).*

Proof Sketch. *We first prove that enforcement of flow dependences is sufficient. We then give an example where failure to satisfy this requirement leads to replay failure, which establishes the necessity direction.*

Sufficiency *A thread t must read a value into its local environment prior to using it. Intuitively, by preserving the flow dependences, a global read in tr' must happen after the same write as in tr, thereby reading the same value. Following thread determinism, reading the same global heap values guarantees the same behavior for each of the threads.*

More precisely, we prove the following properties in an accompanying technical report, available online[2]*:*

I τ and τ' correspond to the same program statement and the execution prefixes prior to both follow the same path.

II If τ accesses local memory, then $\sigma.\rho(\ell_t) = \sigma'.\rho(\ell_t)$, where σ and σ' represent the poststates of τ and τ' respectively, and ℓ_t represents the thread $\tau.t$ (as well as $\tau'.t$).

III If τ accesses the global heap, then $\sigma.H(o, f) = \sigma'.H(o, f)$, where $o.f$ is the field accessed, and σ and σ' represent the poststates of τ and τ', respectively.

The proof is by induction on the length of the trace tr. The sufficiency direction follows straightforwardly from the combination of these three properties.

Necessity *For the necessity direction, consider the following simple scenario, where we assume that the precondition on* print *is that its argument is not* null*:*

t_1	t_2
/* x.f *has non-*null *value* */	
x.f = null	y = x.f
	print(y)

If the write statement assigning null *to* x.f *occurs after the read statement, then the bug exhibited by the run above will not be reproduced. This establishes the necessity of the theorem, and our proof is complete.*

Observe that our guarantee is a strict relaxation of global determinism [6]. Global determinism guarantees reproduction of the global state σ, e.g. $\sigma.H(o_1, f) = v_1$ and $\sigma.H(o_2, f) = v_2$. Although we do not make this guarantee, we ensure reproducton of $\sigma'.H(o_1, f) = v_1$ at some state σ' as well as reproduction of $\sigma''.H(o_2, f) = v_2$ at some other state σ''. More importantly, we guarantee to reproduce the same value at each use (and at each global read), which is sufficient for the reproduction of bugs due to usage of an illegal value.

Also note that it is possible to encode synchronization primitives into our approach. As we explain in Section 4.3, synchronization primitives (like lock acquisition/release) are modeled as variable accesses such that the happens-before orders among them are precisely captured by flow dependences. Therefore, our approach neither misses the deadlocks from the original run nor introduces new deadlocks.

4. Record and Replay

We divide our description of the core replay algorithm according to the two phases it consists of: *recording*, which takes place during the original run; and *replay*, where the requirement is to schedule shared memory accesses according to the order constraints computed atop the recorded data to ensure successful replay. We conclude with a description of extensions and optimizations beyond the core algorithm.

4.1 Recording Phase

During the original run, we record the flow dependences, both inter-thread and intra-thread ones. Our optimization in Section 4.3 dramatically reduces the number of both intra-thread and inter-thread dependences to be recorded. The dependences translate into the constraints in the replay run (Section 4.2), where the goal is to form a global order over the shared accesses that is feasible and preserves all flow dependences.

A main challenge is to collect the requisite information (i.e., flow dependences) efficiently, with low space/time overhead, by

[1] Global read refers to the read of the global heap location (or field).

[2] https://sites.google.com/site/light31415926/tr

(i) restricting use of synchronization and (ii) recording only the absolutely necessary information. We explain how both of these challenges are addressed with reference to Algorithm 1.

ALGORITHM 1: The Light Recording Algorithm

```
   Global: Thread/counter map D
   Global: Location/last-write map lw
   Global: Set F of flow dependences
   Global: Thread/location/last-write map prec
1  OnWriteAccess t: o.f = v s.t. o ↦ o:
2     D(t) ⟵ c = D(t) + 1
3     atomic { o.f = v ; lw ⟵ lw[o.f ↦ c] }
4  OnReadAccess t: v = o.f s.t. o ↦ o:
5     D(t) ⟵ c = D(t) + 1
6     atomic { v = o.f ; c_w ⟵ lw(o.f) }
7     if c_w ≠ prec(t)(o.f) then
8        F ⟵ F ∪ {c_w → c}
9        prec(t) ⟵ prec(t)[(o.f) ↦ c_w]
10    end
```

In the algorithm, every shared access (either read or write) within a given thread t increments a thread-local counter $D(t)$. The role of these counters is to correlate between memory accesses in the original and replay runs.

Write access to location $o.\mathsf{f}$ is recorded in an atomic block as the last write to that location (under $lw(o.\mathsf{f})$). The counter value c is used to identify the access. Atomicity is implemented via locking, though we refrain from fine-grained locking at the granularity of the accessed location, as this results in an excess of locks. Instead, we use lock striping with 2^{10} pre-allocated locks and a simple hashing function that decides a lock according to the offset of field f within the class definition.

To handle read accesses, we also need an atomic block to obtain the respective last write c_w. As explained in Section 2, for read accesses atomicity is achieved via speculation, which works well in practice. The last-write variable $lw(o.\mathsf{f})$ is marked volatile, which guarantees sequentially consistent behavior under the new JMM model (JSR 133). According to the new model, accesses to volatile variables cannot be reordered even if the variables are different.

Ignoring lines 7 and 9, which enforce a space optimization, we are thus left to update the set F of flow dependences with the detected flow dependence $c_w \rightarrow c$ (line 8). We highlight that the data recording at line 8 is completely thread local, without any synchronization. In the implementation, each thread maintains a local (unordered) buffer to store the flow dependences of the reads. The local buffers of all threads are finally merged into F. Comparatively, existing approaches [6, 14] record the data into globally ordered sequences with synchronization. This is very expensive in practice, as (1) it requires updating a variable-size data structure (as opposed to a simple last-write update), and (2) it often also requires resizing of the data structure. By making data recording thread local, we greatly reduce the synchronization overhead.

The optimization in lines 7 and 9 is to avoid recording multiple flow dependences between a given write access w and multiple ensuing reads $r_1 \ldots r_n$ by the same thread. This is a common idiom, which Light addresses efficiently by recording, in the $prec$ map, the write of the last read for a given thread and location. When a read access occurs, Light checks (at line 7) if the write of last read is the same as the write of the current read (of the same location for the given thread). If so, we optimize away the recording. Then, during replay, reads that were not recorded are inferred to share the same write access as the read locally preceding them.

4.2 Replay Phase

Correct replay must respect the flow dependences F recorded in the previous phase. This induces two sorts of constraints:

- **(single dependence)** First, for a recorded dependence $c_w \rightarrow c_r$, we enforce the constraint that the respective write and read are scheduled in this order.
- **(multiple dependences)** Second, and more subtly, given a location ℓ and a pair $c_1^w \rightarrow c_1^r, c_2^w \rightarrow c_2^r$ of flow dependences involving it, we must ensure that these dependences do not interfere with each other in the replay run (per Definition 3.1). That is, either c_2^r occurs before c_1^w or c_1^r occurs before c_2^w. Otherwise, given e.g. the order $c_1^w\ c_2^w\ c_1^r\ c_2^r$, we would have that c_1^r obtains the value written by c_2^w and not c_1^w.

We formulate these restrictions on the order of shared accesses as a constraint system. The constraint system reflects all flow dependences, including implicit dependences due to the $prec$ optimization in lines 7 and 9 of Algorithm 1.

Given a location ℓ, we extract from F all its flow dependences F_ℓ. For the access c in a dependence, we define a corresponding order variable $O(c)$. We then state the following constraints:

$$\bigwedge_{c^w \rightarrow c^r \in F_\ell} O(c^w) < O(c^r) \quad \bigwedge_{c_1^w \rightarrow c_1^r, c_2^w \rightarrow c_2^r \in F_\ell} (O(c_2^r) < O(c_1^w) \bigvee O(c_1^r) < O(c_2^w)) \tag{1}$$

We construct such constraints for each shared location and conjoin all of them.

In addition, for the accesses from the same thread (which may access different locations or appear in different dependences), we have to ensure the same intra-thread order. Therefore for two accesses c_1 and c_2 within the same thread and c_1 happens before c_2, we further assert $O(c_1) < O(c_2)$. The constraint system is then discharged to an off-the-shelf constraint solver, which computes as its solution a global order among the accesses, which our scheduler enforces faithfully.

For example, consider the accesses (prefixed by the counters) from the original run. We record the flow dependences $c_4 \rightarrow c_5$, $c_1 \rightarrow c_6$ and $c_3 \rightarrow c_2$.

t_1	t_2
	c_3 : W(y)
	c_4 : W(x)
	c_5 : R(x)
c_1 : W(x)	
c_2 : R(y)	
	c_6 : R(x)

Following Equation 1, to ensure the flow dependences, we first constrain the schedule of the replay run such that, $O(c_4) < O(c_5)$, $O(c_1) < O(c_6)$ and $O(c_3) < O(c_2)$. Furthermore, to ensure the flow dependences on x do not interfere with each other, we require, $O(c_5) < O(c_1) \vee O(c_6) < O(c_4)$. Besides, the schedule should follow the same thread-local execution orders as in the original run, i.e., $O(c_1) < O(c_2)$ and $O(c_3) < O(c_4) < O(c_5) < O(c_6)$. By combining these constraints, we compute the schedule of the replay run, $O(c_3) < O(c_4) < O(c_5) < O(c_1) < O(c_2) < O(c_6)$, which preserves the recorded flow dependences.

The only remaining subtlety is blind writes, which may violate flow dependences if scheduled incorrectly. Light adopts the simple solution of avoiding execution of blind writes. These are identified straightforwardly as the set of writes not participating in any flow dependence. Avoiding the blind writes does not affect the value arising at any use/read point (i.e., the validity of Theorem 1),

although it disables reproduction of the state resulting from the blind write.

We are now ready to state the guarantee for replay feasibility. For now we assume that the program is free of synchronization primitives, which we introduce as a natural extension of memory accesses later in Section 4.3.

Lemma 4.1 (Replay Feasibility)**.** *The constraint system in Equation 1 is satisfiable, and a replay run that satisfies the solution is guaranteed to be feasible.*

Proof Sketch. *To ensure feasibility, our replay approach must meet two conditions: (i) the order of field read/write events in each thread remains unchanged; and (ii) flow dependences are preserved (i.e., read values are the same, as they correspond to the same write events) [5, 7, 31]. The first condition is satisfied, as we do not modify the intra-thread execution order. The second condition is satisfied by preventing flow dependences from interleaving each other.*

4.3 Extensions and Optimizations

To further reduce the cost, we have developed a set of extensions and optimizations atop the core algorithm.

The main extension is to model lock operations as shared accesses. Following lock semantics, the acquisition atomically reads and updates the owner-thread field and lock-count field of the lock. Release also updates these fields. Therefore, we treat lock acquisition as a read followed by a write of a ghost field, which abstracts the fields of the lock object. The read and the write run inside the lock region, which ensures atomicity across them. Similarly, we treat lock release as a write. Note that the existing lock region already ensures the atomicity required by Algorithm 1, thereby obviating the need for additional synchronization.

The flow dependences recorded following this extension precisely encode the happens-before orders among lock acquisition/release operations. Thus, our approach neither misses existing deadlocks in the original run nor introduces new deadlocks into the replay run.

Threading primitives, such as start and join, are similarly handled. A thread start is modeled as a shared write and the first transition executed by the thread is treated as the corresponding read. Wait and notify are more complex. We uses a method similar to [16, 17], which models wait as two operations: wait_before (that releases the lock) and wait_after (that reacquires the lock). We then record the order between notify and its corresponding wait. Other more advanced primitives, such as notifyAll, are handled similarly to [16].

Beyond modeling synchronization primitives, we also take advantage of such primitives to reduce the amount of recording at the concrete field level. The idea, as discussed in Section 2, is to subsume the flow dependences over concrete fields by the dependences recorded for the guarding locks following the above extension. Other work has also applied this optimization [10, 21, 30].

Lemma 4.2 (Coarse-level Recording)**.** *Assume that the accesses to a location ℓ are always guarded by some synchronization object, such that there are no data races involving ℓ. Then preserving the execution order of the synchronized regions suffices for correct replay of the accesses to ℓ.*

Proof Sketch. *Preservation of the order between the synchronized regions effectively determines the order of accesses to ℓ, and thus preserves the flow dependences.*

In the implementation, we used a conservative static analysis [26] to determine if a location is consistently guarded by some lock. When the analysis fails to reach a definitive answer, we simply disable the optimization w.r.t. accesses to the given location.

In addition, we observe that there are often a large number of consecutive accesses from the same thread. We show that recording all flow dependences of the same thread is redundant if it's not interleaved.

Lemma 4.3 (Sequence without Interleaving)**.** *Given a sequence of consecutive accesses to a location ℓ from a same thread in the record run, it's safe to skip flow dependences in this sequence if there are no interleaving accesses (from a different thread) to ℓ.*

Proof Sketch. *In both record and replay, the flow dependences are only determined by the intra-thread execution order in the same thread when there are no interleaving accesses to ℓ. Since we preserve intra-thread execution order, the unrecorded flow dependences are guaranteed naturally during replay.*

Therefore, instead of logging flow dependences inside a non-interleaved sequence w.r.t ℓ, we record its starting and ending accesses, and enforce no interleavings to them by composing constraints similar to Equation 1.

5. Evaluation

We now describe experiments that we have conducted to validate our approach, as implemented in Light. We govern our discussion of the experiments by three research hypotheses, which we discuss in turn.

5.1 Setup and Methodology

Prototype Implementation For our experiments, we have implemented a prototype of the Light technique. Our implementation consists of three main components, dubbed the transformer, recorder and replayer. The transformer weaves instrumentation code into the target application's class files to record memory accesses. The recorder utilizes these hooks to obtain requisite runtime information and dump the recorded data to disk. Finally, the replayer loads the recording data, computes a reproduction schedule offline, and then launches a new execution according to the computed schedule.

To detect shared accesses, we applied available analyses from the Soot [35] framework in conjunction with the static race detector implemented in Chord [26]. Schedule computation is based on the Z3 SMT solver [8]. Importantly, unlike Clap and other computation-based approaches, our use of the solver does not involve arithmetic computations. We simply derive an order between shared accesses, rather than modeling the value computations inside the statements. Our modeling is efficiently solved via the Integer Difference Logic (IDL) theory provided by z3.

Execution Environment We have performed our experiments atop an x86_64 Thinkpad W530 workstation with eight 2.30GHz Intel Core i7-3610QM processors, 16GB of RAM and 6M caches. The workstation runs version 12.04 of the Ubuntu Linux distribution, and has the Sun 64-Bit 1.7 Java virtual machine (JVM) installed. For the record/replay runs, we used a concurrency level of 8 threads, which is the maximal level supported natively by the workstation.

Benchmarks and Methodology For our measurements, we utilize a diversified suite of 24 benchmarks of varying sizes and from different domains. These include scientific applications: 3 Java Grande Forum (JGF) benchmarks [32] and 8 benchmarks from the Java port of STAMP [25]; web server-side and crawling applications: 7 benchmarks from recent studies on concurrency [16, 24]; as well as 6 concurrent benchmarks from the Dacapo suite [2].

For all benchmarks and measurements, we utilized publicly available workloads rather than creating our own. These are either shipped together with the benchmark or described in previous scientific publications. For 5 out of the 24 benchmarks, the workload

exposes one or more bugs, for a total of 8 bugs, and so we used this subset of the benchmarks to validate the correctness of our replay implementation. We include more complete information on the workloads in the accompanying technical report (TR) [3].

For statistical significance, the data we present below is based on aggregate statistics across 20 independent runs per every application/workload configuration. For reproducibility, we have made our implementation, benchmarks and test drivers publicly available.[4]

5.2 Recording Overhead

Our first hypothesis states that the Light algorithm is significantly more efficient than existing (shared-access) record-based approaches, and that it features tolerable recording overhead in absolute terms. We validate this hypothesis by comparing Light against two recent (shared-access) record-based approaches, Leap [14] and Stride [40], in both time and space overhead. To avoid out-of-memory crashes in long-running benchmarks, we have configured all three tools to dump recorded data into a buffer, and flush the buffer to the disk once the available VM memory drops beyond a prespecified threshold.

The results are visualized in Figures 4 (time) and 5 (space). We present the data in normalized form, as there is high variance across the benchmarks and workloads. We have made the absolute measurement numbers available in the TR. To ensure the reliability of space measurements, we directly count the number of long integers recorded by each of the techniques (where `int`s recorded by Stride are each counted as one half of a long integer for fair comparison). We refer to the unit as Long-integer.

As the figures make clear, Light is indeed significantly more efficient than both Leap and Stride.

For performance overhead, Light either features negligible overhead compared to Leap and Stride (13 cases) or is in the range of roughly $30\% - 60\%$ of the overhead they impose. This is also seen clearly from the aggregate statistics of the overhead, as follows:

	Leap	Stride	Light
average	4.11	4.66	0.44
median	2.58	2.92	0.42
minimum	0.17	0.19	0.15
maximum	17.85	23.89	0.73

The statistics above also establish the second clause in the hypothesis, which states that Light has tolerable overhead. Indeed, on average Light imposes a slowdown of $< 50\%$ compared to $> 4X$ slowdown due to Leap and Stride. Even more significantly, in the worst case, the slowdown we observed for Light was 73%.

Switching to space, we encounter an even more significant trend. While Leap and Stride are largely tied in space consumption, Light makes drastically lower space requirements. Aside from 3 benchmarks where Light requires roughly $30\% - 50\%$ of the space needed by Leap/Stride, in all other 21 cases the space consumed by Light is negligible. Indeed, recording only non-transitive flow dependences is a significant discount compared to all data dependences, and beyond that Light features the space optimizations described in Section 4.3.

Another view of the space-consumption comparison is given by the aggregate statistics (in K Long-integer units), as follows:

	Leap	Stride	Light
average	136,053	135,570	9429
median	36,425	34,566	1,461
minimum	19	30	1
maximum	1,394,378	1,394,378	69,559

[3] https://sites.google.com/site/light31415926/tr

[4] https://sites.google.com/site/light31415926/

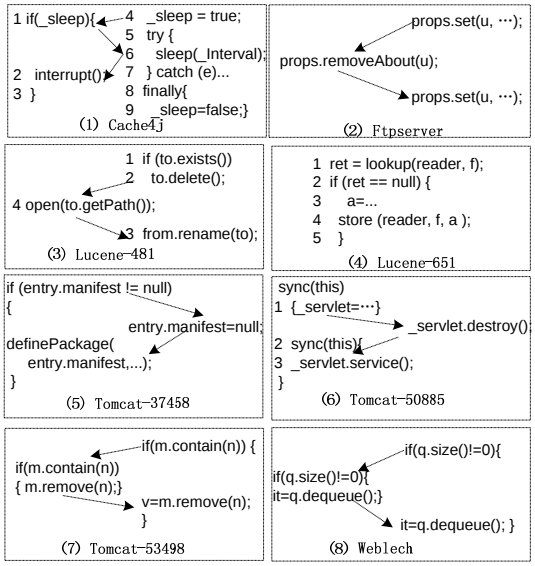


Figure 6. Real world bugs

The gap is indeed dramatic in favor of Light.

Naturally, there is high correlation between the time and space statistics, as Figures 4 and 5 highlight. In both cases, Light is not only significantly better than Leap and Stride, but is also tolerable in absolute terms. We thus conclude that hypothesis H1 is confirmed.

5.3 H2: Effectiveness of Record-based Replay

As noted in the introduction, there are alternatives to record-based replay, which so far have been desirable due to the high cost of record-based techniques. We have demonstrated above that low-overhead record-based replay is feasible. Our second hypothesis is that it is preferable to other approaches, enabling robust reproduction of bugs.

To validate this claim, we compared Light with Clap [15] and Chimera [21], two state-of-the-art approaches that are not based on recording of shared accesses. We did not compare with Leap [14] and Stride [40] as all the (shared-access) record-based approaches have the same guarantees. Clap is computation based, deriving the order among shared accesses without recording them. Chimera is a patch-based approach. It patches the program, transforming it into race-free code, and then records the orders among lock operations. Because the source code of Clap and Chimera is not available, we have created our own implementation of these techniques, which we have made publicly available for scrutiny and validation.

For the comparison, we used the 8 bugs mentioned above, all of which are recorded in the Apache bug database. The bugs are summarized in Figure 6. Following the schedules highlighted in the figure, these bugs cause exceptional situations (e.g., FileNotFoundException in (3) and NullPointerException in (6)). Although the scenarios shown in Figure 6 are concise, the replayer needs to follow a non-trivial series of steps to achieve reproduction. Also, as with real-world code in general, most of the bugs involve use of nontrivial constructs (e.g. `wait`/`notify`, exception handling, etc).

Moving to the results, Light was able to reproduce all the bugs (consistently with its formal guarantee). Replay time for each of the bugs (in seconds) is given in Table 1. As the table discloses, constraint solving time is correlated with space consumption (i.e., the number of recorded shared accesses). Overall, scheduler computa-

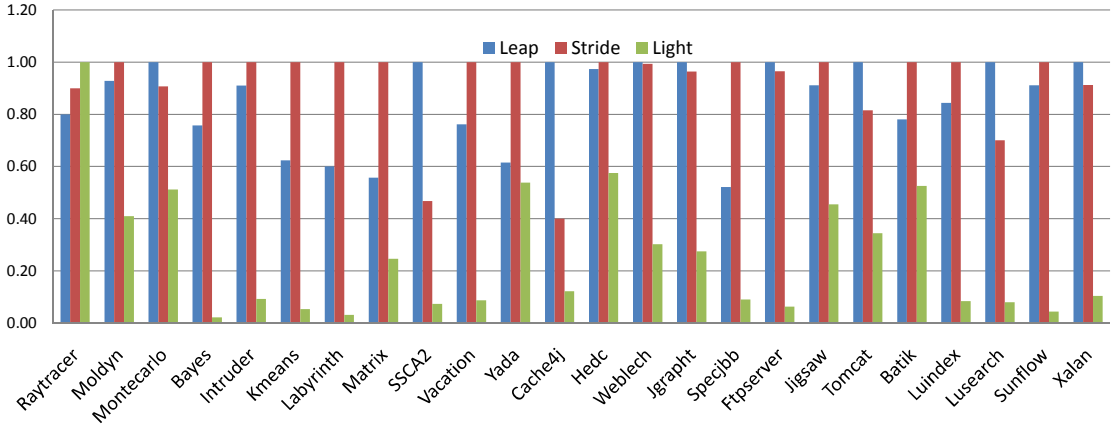


Figure 4. Normalized comparison between Light, Leap and Stride in terms of time overhead

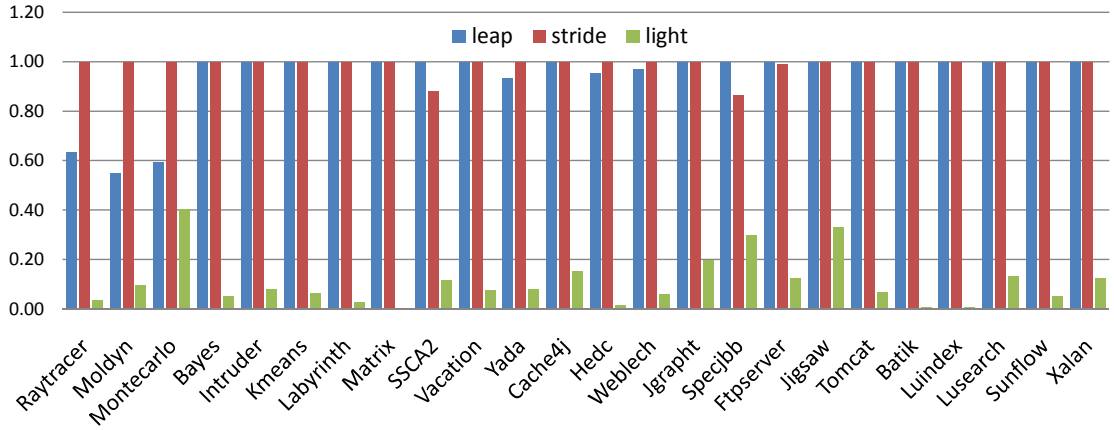


Figure 5. Normalized comparison between Light, Leap and Stride in terms of space consumption

tion requires 62 seconds on average, and the replay run an additional average of 34 seconds.

Table 1. Replay Measurement

	Space (K)	Solve(s)	Replay(s)
Cache4j	297	39	8
Ftpserver	13	10	42
Lucene-481	1088	112	62
Lucene-651	2596	301	87
Tomcat-37458	15	5	23
Tomcat-50885	590	30	44
Tomcat-53498	28	4	9
Weblech	2	2	3

As for Clap, it fails to reproduce the Ftpserver, Lucene-481, Lucene-651, Tomcat-53498 and Weblech bugs. In our analysis, this is because Clap is designed to reproduce bugs in C programs, especially in scientific computations with primitive values. Real-world Java programs, on the other hand, often use data types that do not have native solver support, such as `HashMap` in the bugs above. It is not clear to us if and how Clap can be extended to support these cases.

Chimera fails to reproduce the Cache4j, Tomcat-37458 and Tomcat-50885 bugs. These are missed due to the design of Chimera, whereby execution is monitored for pairs of racing statements whose enclosing methods rarely run in parallel. Chimera then enforces lock-based mutual exclusion over the methods, which is expected to have minor impact on overhead. This heuristic conflicts with reproduction of the bugs above, as they occur in methods that rarely run in parallel.[5] Chimera serializes the methods, thereby hiding the bugs.

To summarize, out of the three replay approaches we examined, the only approach that is able to reproduce all 8 bugs, and with relative efficiency, is the record-based approach Light. To be sure, the misses suffered by the other approaches are due to inherent, rather than engineering, limitations: solver expressiveness and side effects due to locks. We conclude, therefore, that hypothesis H2 also holds.

5.4 H3: Significance of Optimizations

The last hypothesis, H3, concerns the importance of the two optimizations we present in Section 4 atop the core Light algorithm. H3 states that the gain thanks to each of these optimizations (and their conjunction) is significant. For simplicity, we use O_1 to refer to the optimization for the sequence without interleavings (Lemma 4.3),

[5] See e.g.: `https://issues.apache.org/bugzilla/show_bug.cgi?id=50885`

and O_2 to refer to the optimization based on synchronization objects (Lemma 4.2). To measure the effects of the optimizations, we have created three versions of Light: V_{basic}, which applies neither O_1 nor O_2; V_{O_1}, which applies O_1 only; and V_{both}, which applies both O_1 and O_2.

Figure 7a (time) presents the relative contributions of the two optimizations w.r.t. the overhead reduction: 100% refers to the normalized overhead of V_{basic}. The top bar denotes the overhead reduction thanks to O1, i.e. the difference between the overhead of V_{basic} and V_{O_1}. The middle bar denotes the overhead reduction thanks to O2, i.e. the difference between the overhead of V_{O_1} and V_{both}. Finally, the bottom bar denotes the remaining overhead of V_{both} (w.r.t. the original program). Similarly, Figure 7b (space) shows the relative contributions w.r.t. the space reduction, where 100% refers to the normalized space consumption of V_{basic}.

The results, as visualized in the figures, support the hypothesis. First, for time (Figure 7a), in 20/24 benchmarks the O_1 optimization reduces the overhead by $\geq 20\%$, and in 8/24 it reduces overhead by $\geq 50\%$. O_2, though less effective, is still visibly useful with a contribution of $\geq 20\%$ reduction in 9 cases and $\geq 50\%$ in 4 cases.

The trend with space optimization (Figure 7b) is similar, if not even more significant. O_1 is highly dominant with an improvement of $\geq 50\%$ in 16/24 cases. O_2 reduces the space overhead by $\geq 20\%$ in 6 of the cases.

In conclusion, the experimental data confirms both that O_1 and O_2 are significant, and that their combination is effective. We thus conclude that H3 is confirmed.

6. Related Work

There has been substantial work on record and replay of concurrent programs [3, 6, 14, 15, 19–22, 28, 36, 37, 39, 40]. We have discussed some of them in Section 1. Hence in this section, we focus on other work.

Conventional deterministic record and replay usually introduces a significant overhead. For example, Dejavu [6] maintains a global order for all synchronizations and shared variable accesses and thus introduce high runtime overhead. JaRec [11] and RecPlay [30] made trade-offs between the efficiency and replay determinism. They maintain partial orders on lock acquisition/release events. However, they cannot guarantee the determinism since the shared memory accesses are not tracked.

To overcome the challenges, more approaches are proposed. Rocket [3], a deterministic record and replay approach built on top of a dynamic analysis infrastructure [4], logs all inter-thread dependences using synchronizations. Besides, Rocket requires non-trivial changes in JVM. DoublePlay [36] applies a strategy called uniparallelism to achieve deterministic replay by time slicing multi-threaded executions. Doubleplay has important advantages, but runs the risk of serializing execution in the worst case. Also, the efficient checkpointing required by DoublePlay is hard to implement in Java [23].

ODR [1] guarantees output determinism, allowing the replay run to follow a different execution path. Similarly to Clap [15], ODR uses the solver to symbolically analyze expression values, which has limitations in handling complex arithmetic computations in practice. Although our approach is also backed by a solver, it does not need to symbolically reason about expression values. Therefore, it does not suffer from these limitations.

Lee et al. [20] propose to record the values of global reads and rely on offline symbolic analysis to construct the dependences between the global reads and writes. However, Gibbons et al. [12] prove that computing a feasible schedule using the value trace is NP-complete. In practice, their approach lacks exact dependence information, and may not scale to industry-scale systems as it induces a very large search space. Besides, the approach of Lee et al. [20] records values at all global reads, which may result in an excess of data. Our approach leverages a set of optimizations to greatly reduce the data to be recorded.

Replay also draws insight from the industry. Intel has released PinPlay [29], a user-friendly product built atop the Pin instrumentation tool. Similarly to Rocket [3], PinPlay logs all inter-thread dependences to achieve multi-process replay.

Similarly to our approach, Eidetic Systems [9], which apply to the operating system kernel, also track flow dependences (over file-system accesses) for replay. However, the Eidetic Systems replay algorithm is heuristic and ensures correctness only in the specific context for which it was developed. Our work has the formal guarantee of replay determinism and proposes a systematic algorithm for computing the feasible schedules. Besides, from the performance view, our work formally proves the tight bound for recording and describes certain advanced optimizations.

Similarly to our approach, FastTrack [10] associates each access with a counter, though the counter is used for a completely different purpose. The FastTrack counter is in fact a vector clock, which enables fast online querying of the happens-before relation in race detection by trading space against query time. Replay systems do not need the online querying capability (and thus the vector clock). Besides, FastTrack has another form of counter called epoch, which resembles our counter. The epoch is applied to optimize FastTrack if certain conditions are met, which differs from our use of the counter.

7. Conclusion

We present Light, a lightweight technique that can reproduce concurrent bugs with guarantees. It leverages (and also formally proves) that logging flow dependences alone is sufficient and necessary to reproduce concurrent bugs. We also develop a set of advanced optimizations. Our evaluation shows that Light has runtime and space overheads that are almost one order of magnitude lower than existing record-based techniques. It can reproduce all the bugs we have studied, whereas techniques following other approaches miss some of them due to incompleteness.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. We also thank Charles Zhang for initiating the discussion. This research is supported, in part, by the National Science Foundation (NSF) under grants 0845870, 1320444 and 1320326. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *SOSP*, 2009.

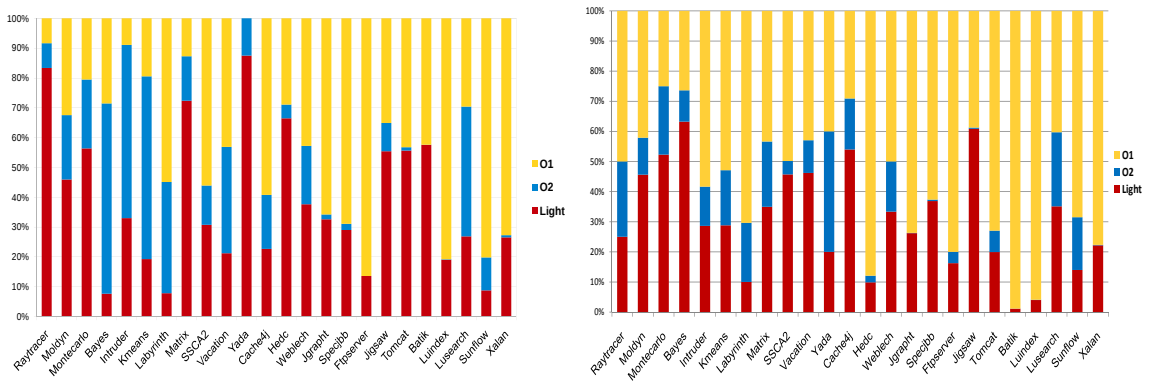
[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[3] M. D. Bond and M. Kulkarni. Tracking conflicting accesses efficiently for software record and replay. Ohio State CSE Technical Report, OSU-CISRC-2/12-TR01, 2012.

[4] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *OOPSLA*, 2013.

[5] F. Chen and G. Roşu. Parametric and sliced causality. In *CAV*, 2007.

[6] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT*, 1998.



(a) (b)

Figure 7. (1) Breakdown of Time Overhead (2) Breakdown of Space Overhead

[7] T. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. In *Runtime Verification*. 2013.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[9] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *OSDI*, 2014.

[10] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[11] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: A portable record/replay environment for multi-threaded java applications. *Software Practice and Experience*, 34(6), May 2004.

[12] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4), Aug. 1997.

[13] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008.

[14] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent java programs. In *FSE*, 2010.

[15] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *PLDI*, 2013.

[16] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

[17] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.

[18] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, 2010.

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

[20] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *MICRO*, 2009.

[21] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *PLDI*, 2012.

[22] K. H. Lee, D. Kim, and X. Zhang. Infrastructure-free logging and replay of concurrent execution on multiple cores. In *PPoPP*, 2014.

[23] P. Liu and C. Zhang. Pert: The application-aware tailoring of java object persistence. *TSE*, 38(4), July 2012.

[24] P. Liu, O. Tripp, and C. Zhang. Grail: Context-aware fixing of concurrency bugs. FSE, 2014.

[25] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization*, 2008.

[26] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, 2007.

[27] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5), May 1976.

[28] 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 *SOSP*, 2009.

[29] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, 2010.

[30] M. Ronsse and K. De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2), May 1999.

[31] K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, 2005.

[32] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC, 2001.

[33] O. Tripp. *Incorporating Data Abstractions into Concurrency Control*. PhD thesis, Tel-Aviv University, 2014.

[34] O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: Effective discovery of dataflow impediments to parallelization. In *OOPSLA*, 2011.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, 1999.

[36] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *ASPLOS XVI*, 2011.

[37] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS XV*, 2010.

[38] Yices. The yices smt solver. `http://yices.csl.sri.com/`.

[39] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.

[40] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *ICSE*, 2012.