iReplayer: In-situ and Identical Record-and-Replay for Multithreaded Applications

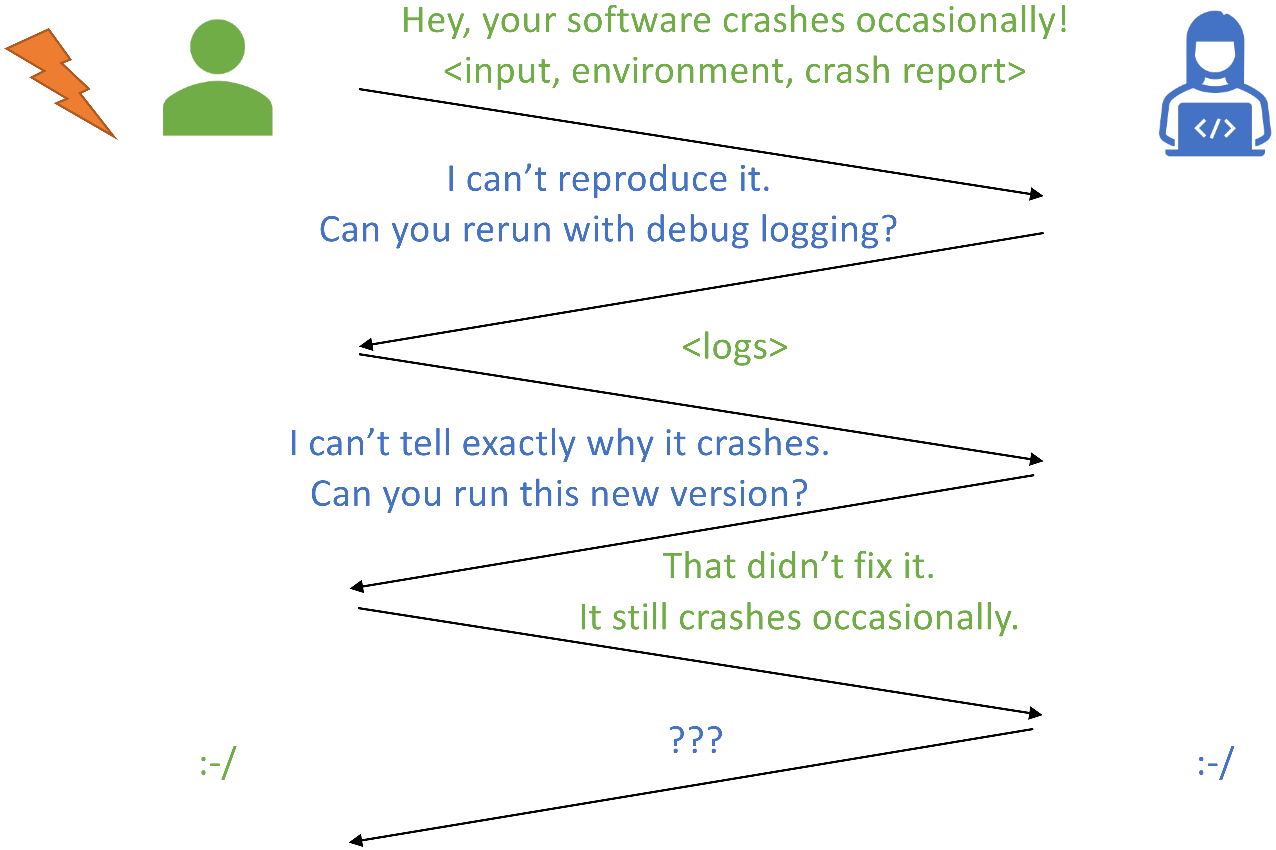
Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian*, Tongping Liu

University of Texas at San Antonio, Huawei US Lab*

PLDI 2018

Discussion Lead: Abhilash Jindal

Why record-and-replay?



Why is reproducing hard?

```
def transfer(src: Account, dst: Account, amount: int):
    if src.bal > amount:
        src.bal -= amount
        dst.bal += amount
```

Initial state: src.bal = dst.bal = 800

T1:
Amount: 500
R src.bal -> 800
W src.bal -> 300
W dst.bal -> 1300

T2:
Amount: 400
R src.bal -> 300

T2:
Amount: 400
R src.bal -> 800
W src.bal -> 400
W dst.bal -> 1200

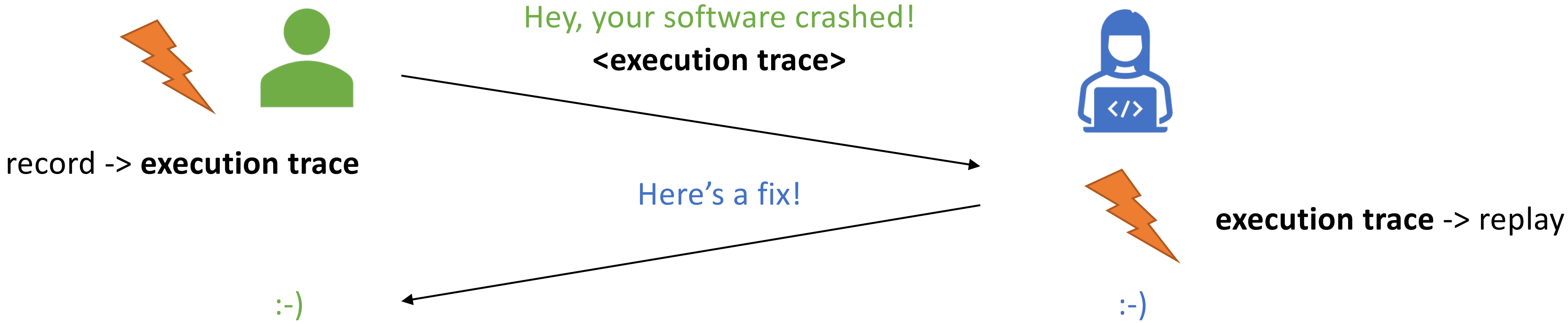
T1:
Amount: 500
R src.bal -> 400

T1:
Amount: 500
R src.bal -> 800
W src.bal -> 300
W dst.bal -> 1300

T2:
Amount: 400
R src.bal -> 800
W src.bal -> 400
W dst.bal -> 1200

Different executions produce different states

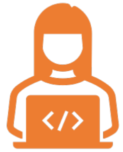
Identical record-and-replay



- Address all sources of non-determinism
- Always on: Low overhead
- Don't leak privacy sensitive information

Why in-situ?

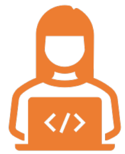
record -> **execution trace**



replay **execution trace** in *same machine process* to debug

- Users don’t want to share privacy sensitive trace
  - *But, in-situ doesn’t help. Users are not technical enough to debug. Programmers must still see the trace for recovery.*
- Can do automatic online recovery
  - *No evidence shown in the paper to perform such recovery automatically*

Why in-situ? (2)



replay **execution trace** in *same machine process* to debug

- Programmer *is* the user
  - Large scale concurrency testing
  - Production environment monitoring

Easier to do *identical* replay in-situ with low-overhead

- Same hardware
  - Identical binary
  - Identical floating-point behavior
- Same OS
  - Identical memory layout
  - Identical file system
- Same process
  - Identical thread ids: Memory allocators use thread ids

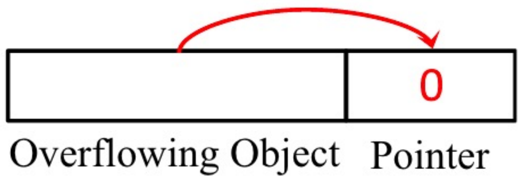
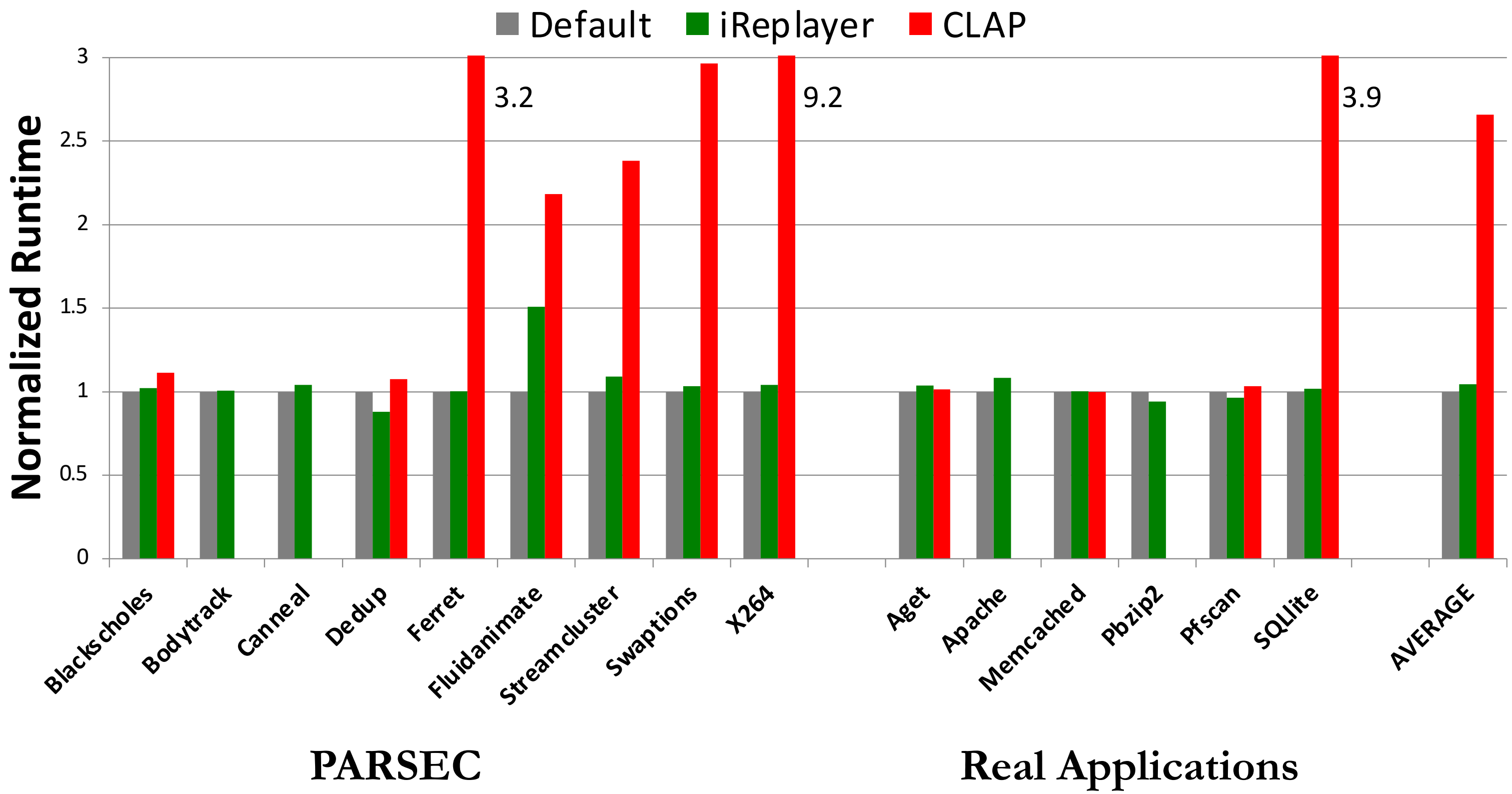


Figure 1. A null reference problem.

Existing RnR Systems

- Offline RnR: replay occurs after recording
  - Instrumentation: iDNA[VEE'06], PinPlay[CGO'10] **10-100X**
  - Offline Assisted Analysis: ODR[sosp'19], CLAP[PLDI'13], Light[PLDI'15], H3[ATC'17], Castor[ASPLOS'17]
    Low overhead, but substantial time of offline analysis
  - Custom Hardware: Strata[ASPLOS'06], DeLorean[ISCA'08], Capo[ASPLOS'09] **Impractical**
  - Hybrid Analysis: Chimera[PLDI'12] **40% for 4 threads and hide failures**
- Online RnR: record and replay execute concurrently
  - Speculation Based: Respec[ASPLOS'10] **55% for 4 threads**
  - Uniparallelism: DoublePlay[ASPLOS'11] **28% for 4 threads**
  - iReplayer (this paper)
    Unmodified OS, hardware, compiler. Low overhead

iReplayer: Recording Overhead



iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

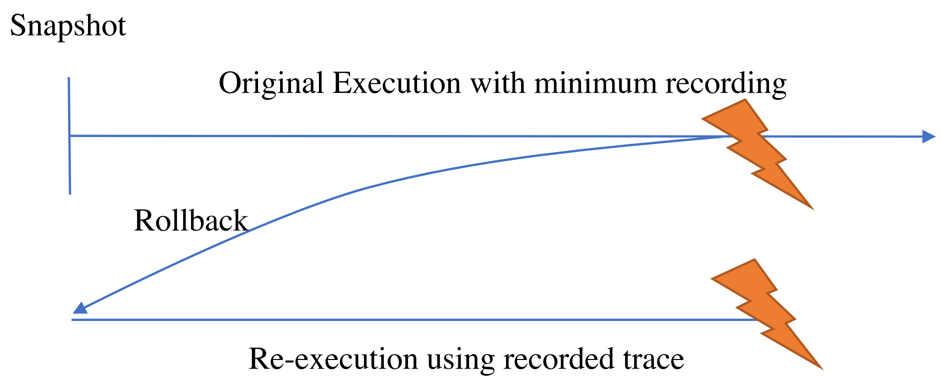
iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

iReplayer demo: Interactive Debugging

Basic Idea


Snapshot
Original Execution with minimum recording
Rollback
Re-execution using recorded trace

Basic Idea

Record phase

Replay phase

Snapshot: memory, etc.

```
sum = 0

l = [1, 2, 3, 4, 5]
for x in l:
        sum += x

assert sum > 0
```

Rollback

Replay

Deterministic program. No logging required.

Addressing sources of non-determinism

- **Single thread**
  - **System calls: Repeatable, recordable, revocable, deferrable, irrevocable**
- Multithreading
  - Thread lifecycle
  - Memory allocators
  - Synchronization
  - Racy memory accesses

Repeatable system calls

Record phase

Replay phase

Snapshot: memory, etc.

```
sum = getpid( )

l = [1, 2, 3, 4, 5]
for x in l:
        sum += x

assert sum > 0
```

In-situ* -> *No need to record.

getpid() will give same output in replay.

Rollback

Replay

Other examples: *getcwd( )*

Recordable system calls

Record phase

Replay phase

Snapshot: memory, etc.

```
sum = input( )

l = [1, 2, 3, 4, 5]
for x in l:
        sum += x

assert sum > 0
```

Record system call output: 23

Rollback

Replay:

Feed 23

Other examples: *gettimeofday( )*

Revocable system calls

Record phase

Replay phase

Snapshot: memory, file position, etc.

```
sum = f.read( )

l = [1, 2, 3, 4, 5]
for x in l:
        sum += x

assert sum > 0
```

In-situ* -> *No need to record file contents.

Rollback: lseek to same file position

Replay

Other examples: *write( )*

Deferrable system calls

Record phase

Replay phase

Snapshot: memory, file position, etc.

```
sum = f.read( )
f.close( )
l = [1, 2, 3, 4, 5]
for x in l:
        sum += x

assert sum > 0
```

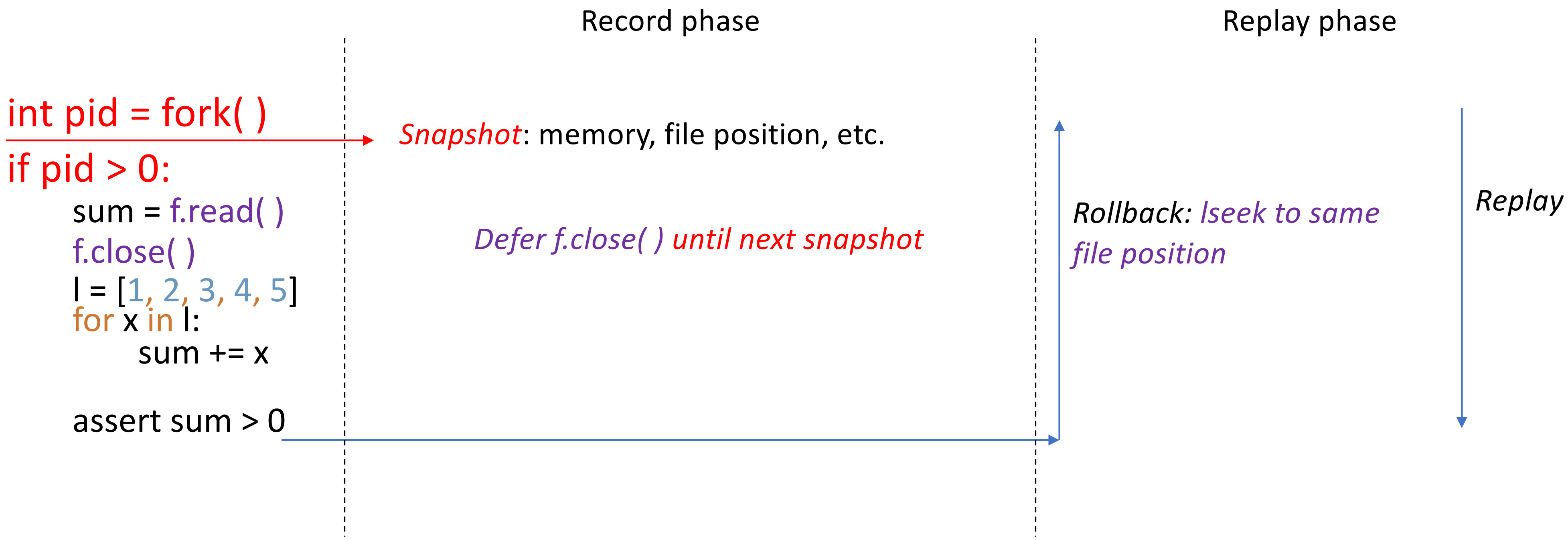
Defer f.close( )

Rollback: lseek to same file position

Replay

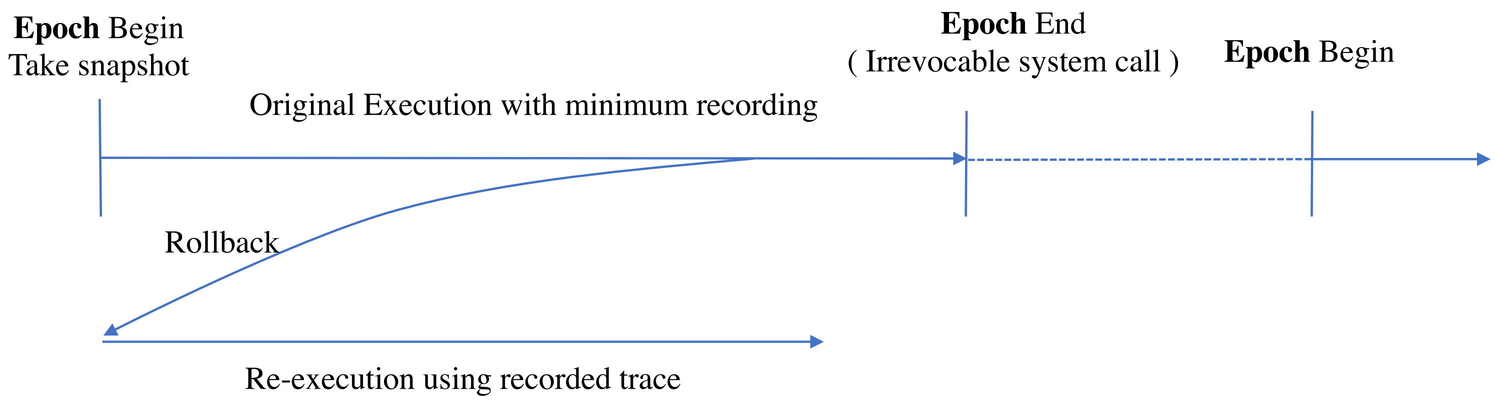
Other examples: *socket.close( ), munmap*

Irrevocable system calls



Don’t know how to rollback irrevocable syscalls. So, we won’t allow rolling back to a state prior to that syscall.

Epoch-based record replay



Cannot rollback to a state prior to the current epoch!

Justification: Root cause of bugs is typically not too far from the actual bug

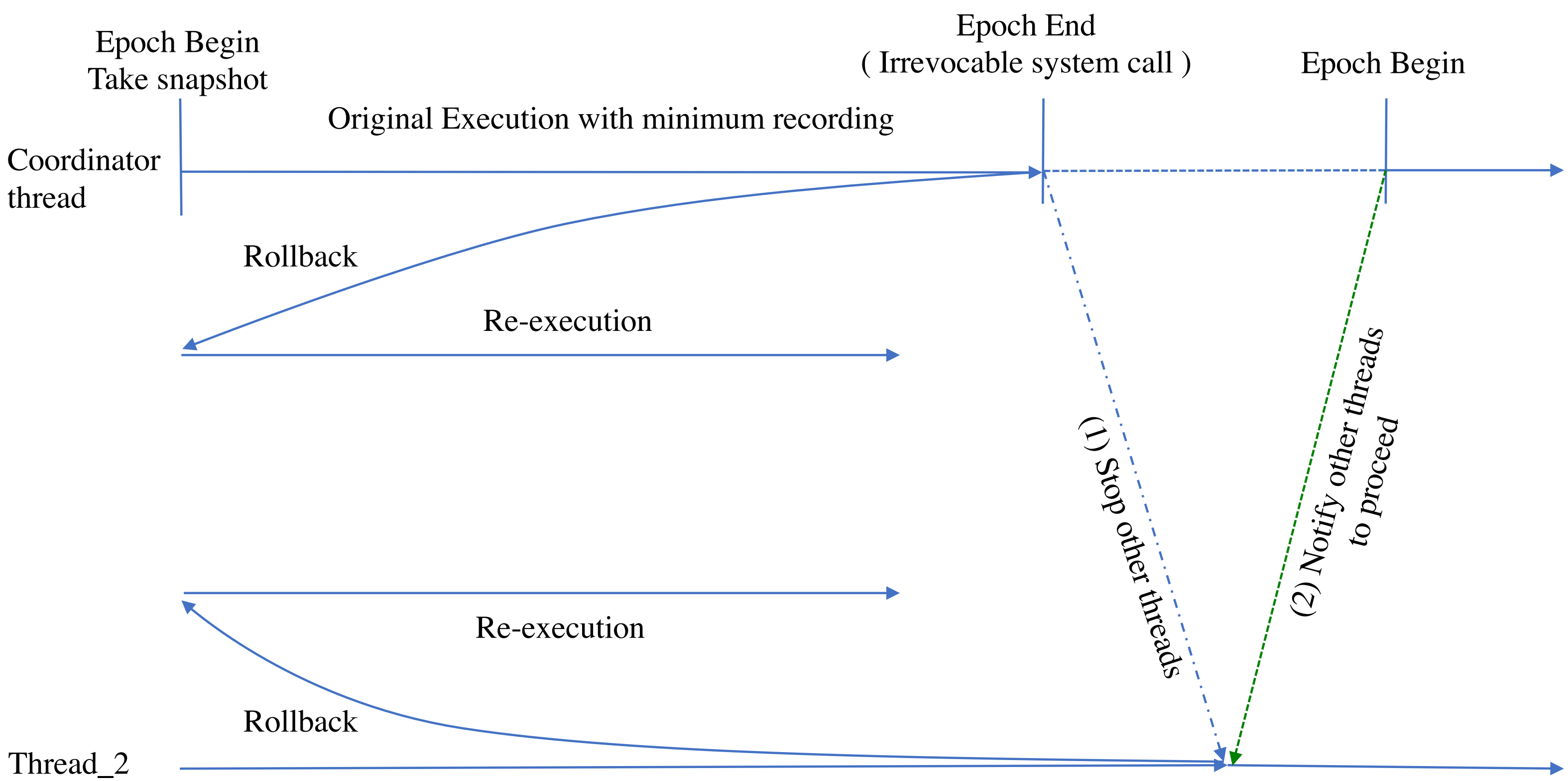
Syscalls: Adapt to In-Situ Setting

Category	Syscall Examples	Handling of the syscall
Repeatable	getpid, getcwd	no handling
Recordable	gettimeofday, mmap, open	record/replay
Revocable	file read/write	rollback side-effect with low overhead
Deferrable	close, munmap, (thread exits)	defer to next epoch
Irrevocable	fork, lseek	stop current epoch

Addressing sources of non-determinism

- Single thread: system calls
  - Repeatable, recordable, revocable, deferrable, irrevocable
- **Multithreading**
  - **Thread lifecycle**
  - **Synchronization**
  - Memory allocators
  - Racy memory accesses

Multithreading



Thread lifecycle

Record phase

Replay phase

T1

pthread_create(&t2..)

T2

Snapshot: memory, etc.

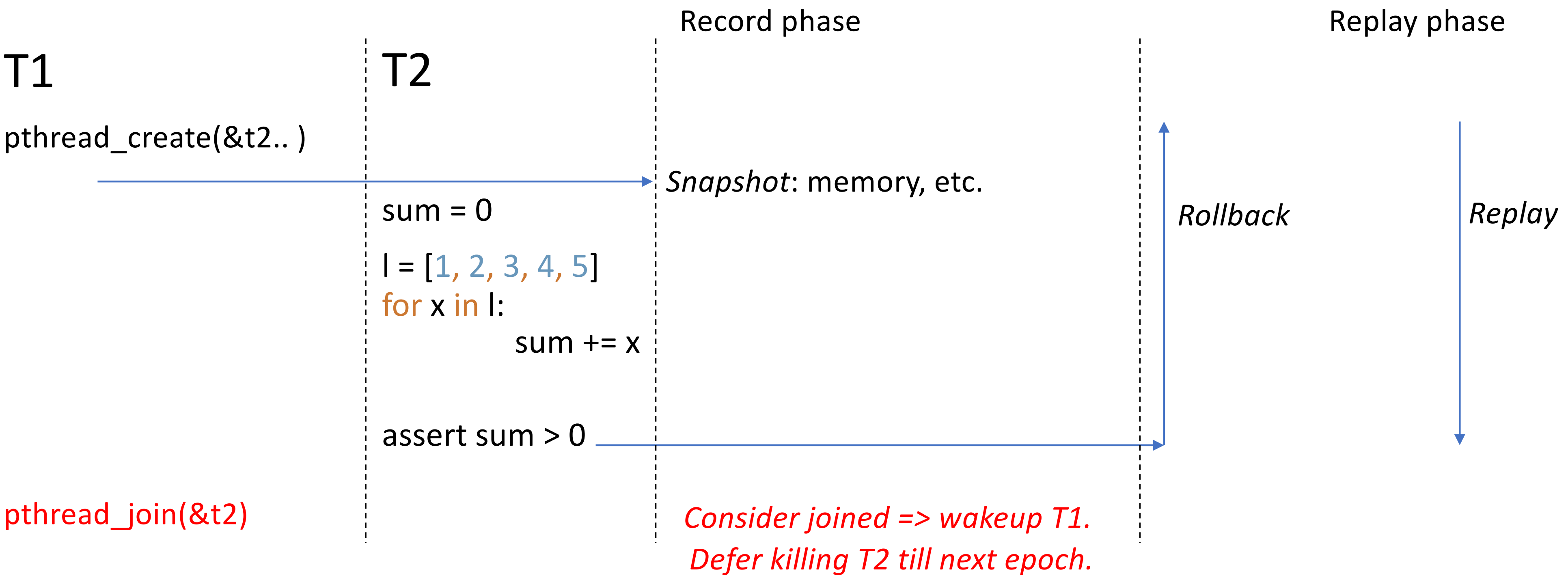
```
sum = 0
l = [1, 2, 3, 4, 5]
for x in l:
        sum += x

assert sum > 0
```

Rollback: make T2 wait on a condition variable

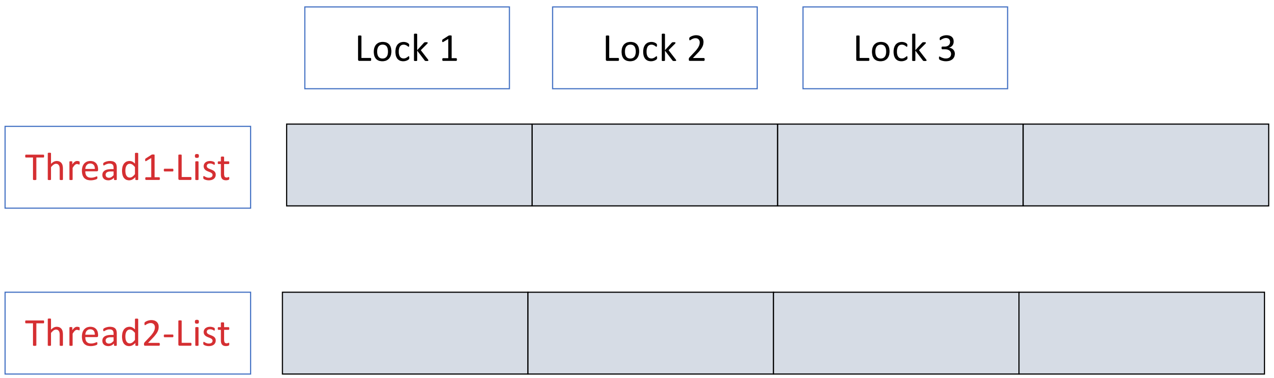
Replay: don't create a new thread. Signal T2's condition variable

Thread lifecycle (2)



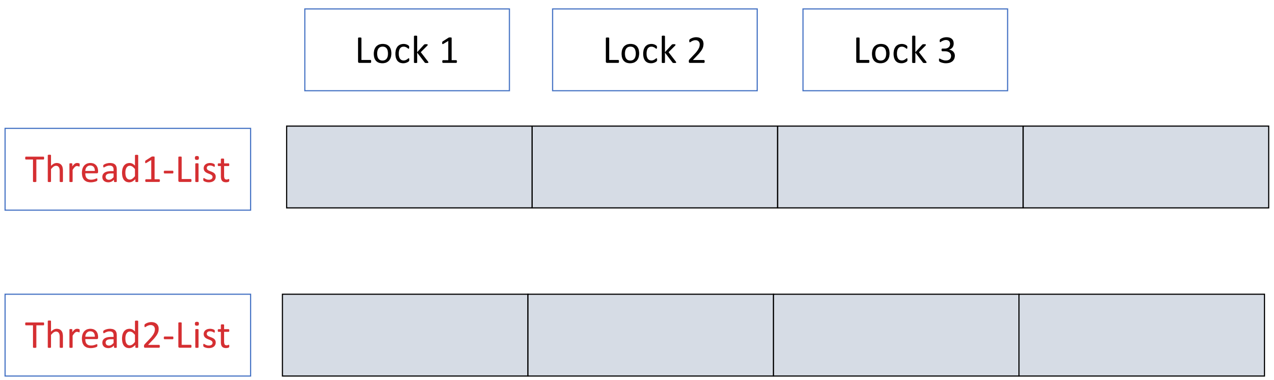
Recording Synchronizations

```
Thread1:                  Thread2 :
                          Lock(&lock2);
Lock(&lock1);             Unlock(&lock2);
Lock(&lock2);
Unlock(&lock2);
Unlock(&lock1);
Lock(&lock3);
Unlock(&lock3);           Lock(&lock1);
                          Unlock(&lock1);
```



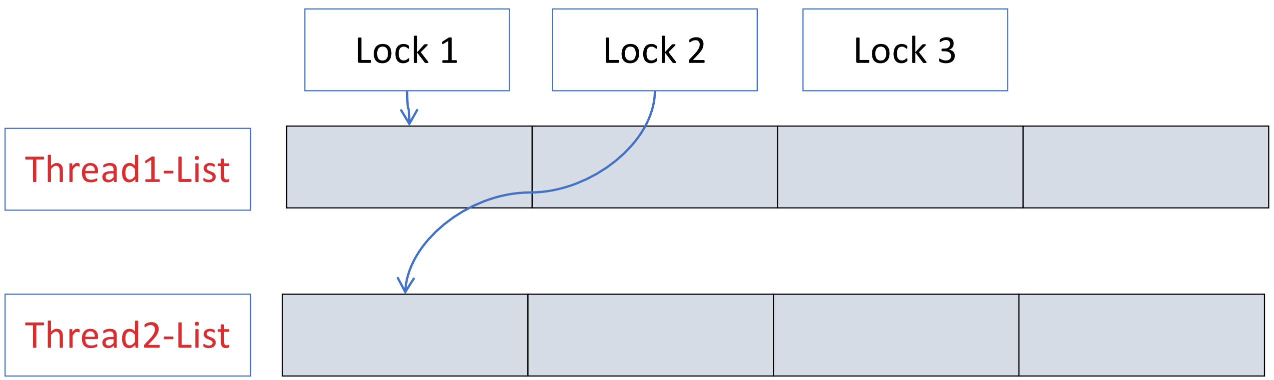
Recording Synchronizations

```
Thread1:                  Thread2 :
                          Lock(&lock2);
Lock(&lock1);             Unlock(&lock2);
Lock(&lock2);
Unlock(&lock2);
Unlock(&lock1);
Lock(&lock3);
Unlock(&lock3);           Lock(&lock1);
                          Unlock(&lock1);
```



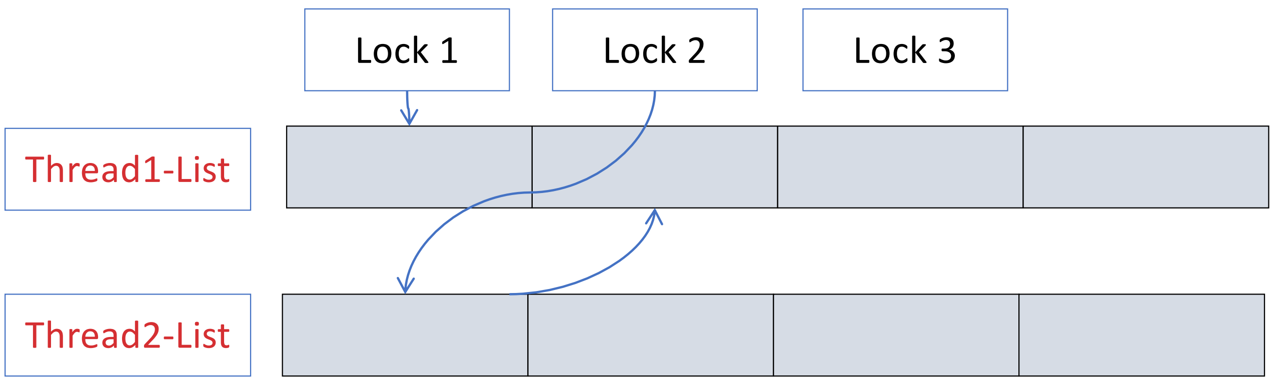
Recording Synchronizations

```
Thread1:                Thread2 :
                        Lock(&lock2);
Lock(&lock1);           Unlock(&lock2);
Lock(&lock2);
Unlock(&lock2);
Unlock(&lock1);
Lock(&lock3);
Unlock(&lock3);         Lock(&lock1);
                        Unlock(&lock1);
```



Recording Synchronizations

```
Thread1:                Thread2 :
                        Lock(&lock2);
Lock(&lock1);           Unlock(&lock2);
Lock(&lock2);
Unlock(&lock2);
Unlock(&lock1);
Lock(&lock3);
Unlock(&lock3);         Lock(&lock1);
                        Unlock(&lock1);
```



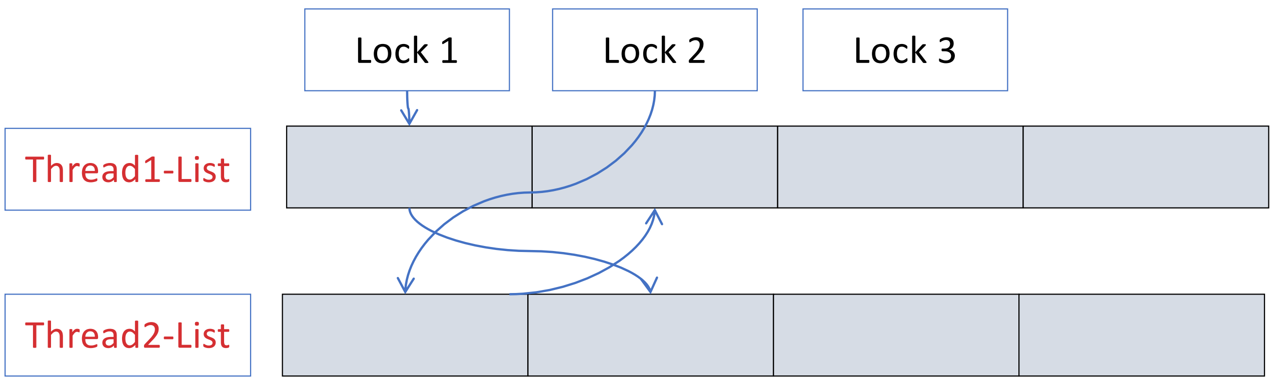
Recording Synchronizations

```
Thread1:

Lock(&lock1);
Lock(&lock2);
Unlock(&lock2);
Unlock(&lock1);
Lock(&lock3);
Unlock(&lock3);
```

```
Thread2 :
Lock(&lock2);
Unlock(&lock2);

Lock(&lock1);
Unlock(&lock1);
```

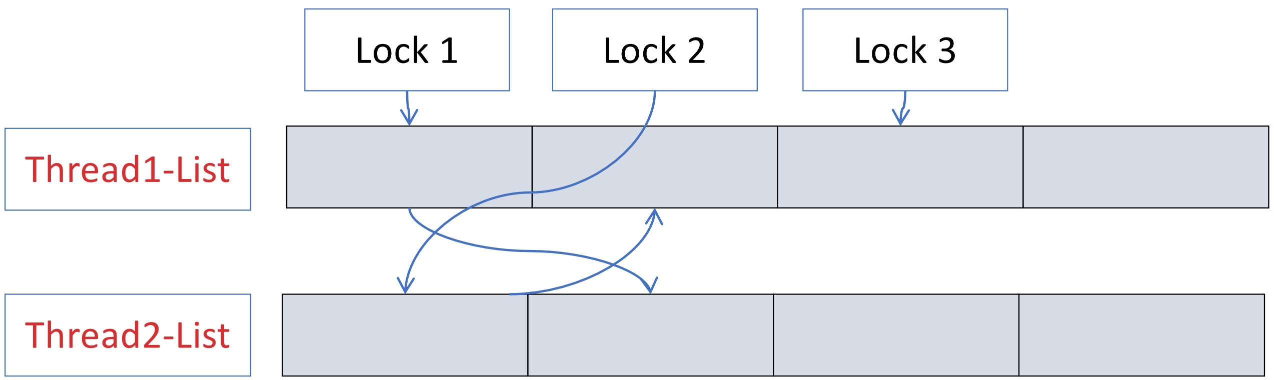


Recording Synchronizations

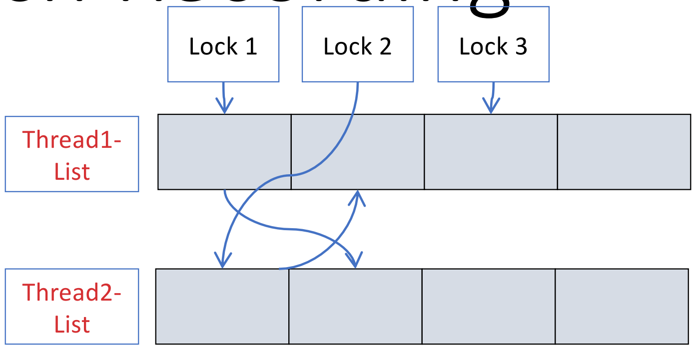
```
Thread1:

Lock(&lock1);
Lock(&lock2);
Unlock(&lock2);
Unlock(&lock1);
Lock(&lock3);
Unlock(&lock3);
```

```
Thread2 :
Lock(&lock2);
Unlock(&lock2);
Lock(&lock1);
Unlock(&lock1);
```



Benefits of Such Recording



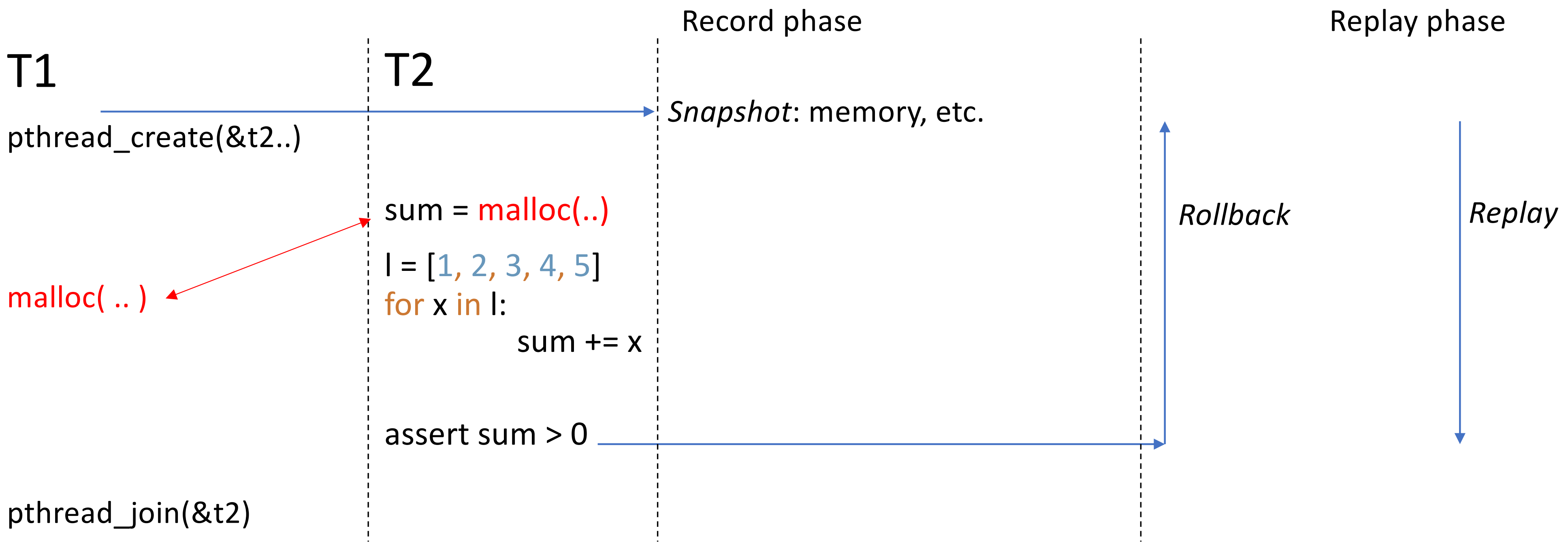
- Local-order recording guarantees identical reproduction
- Pre-allocated list avoids allocation overhead
- No additional locks required for recording
- Events are connected via per-thread or per-sync-variable lists, which can be used directly for reproduction

If an event is at the header of both per-thread list
and its per-variable-list, the thread can proceed. Otherwise, wait.

Addressing sources of non-determinism

- Single thread: system calls
  - Repeatable, recordable, revocable, deferrable, irrevocable
- **Multithreading**
  - Thread lifecycle
  - Synchronization
  - **Memory allocators**
  - **Racy memory accesses**

Memory allocation



Identical Allocations/Deallocations

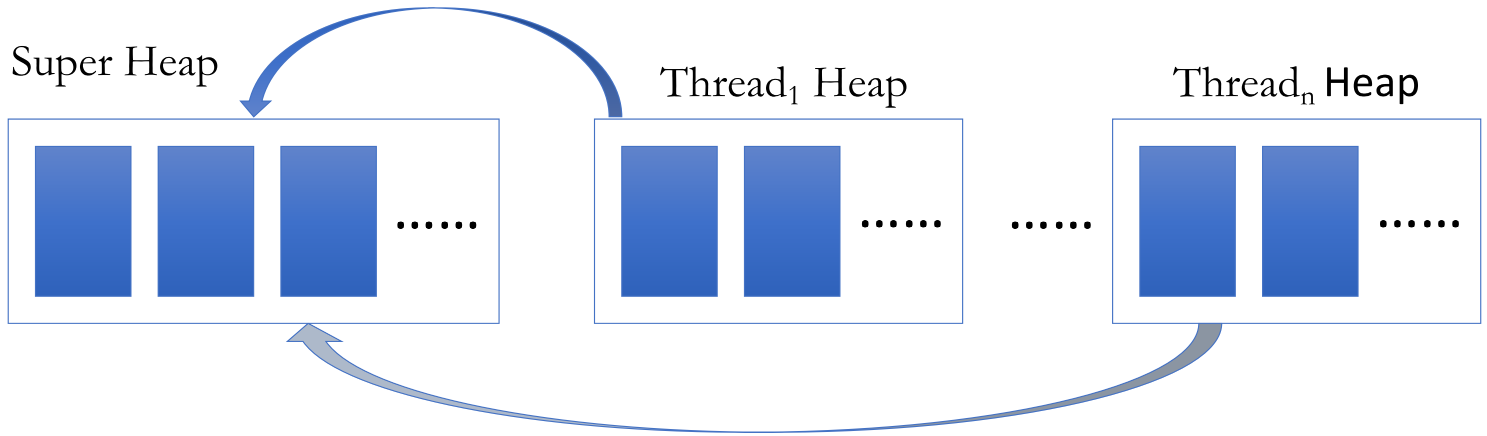
✧Observation:

- Memory allocations/deallocations inside each thread is determined by the program order

✧**Basic approach:**

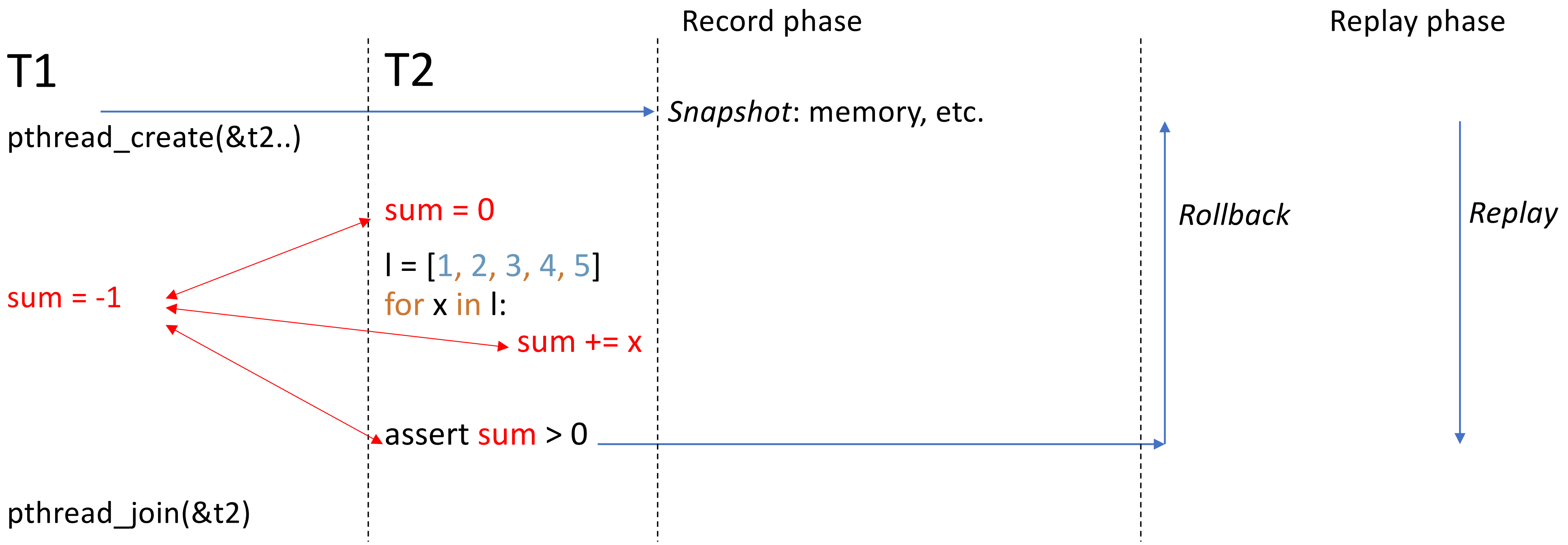
- Every thread has its own heap
- Controlling each thread's interaction with other threads

Identical Allocations/Deallocations



- Allocation: deterministically fetch blocks via a global lock
  Deallocation: freed objects are returned to the current thread's heap

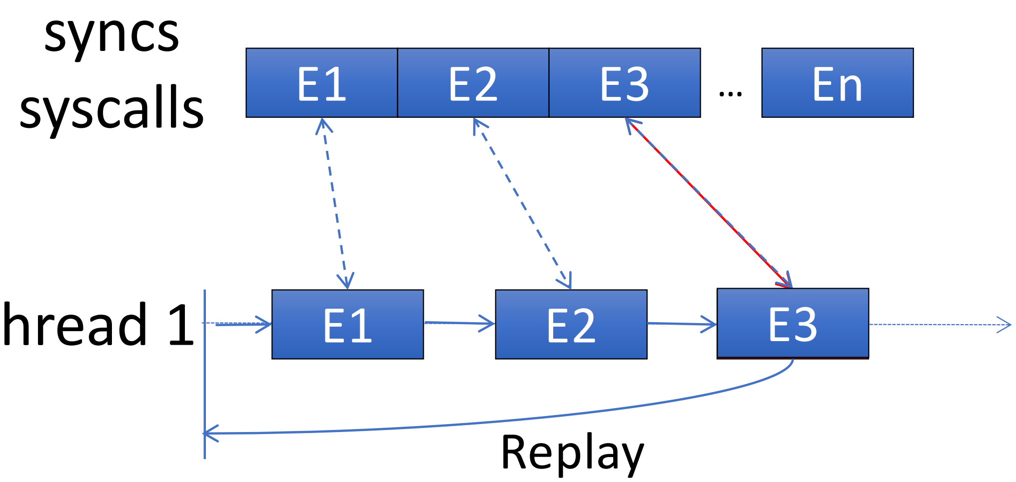
Racy accesses



Observations

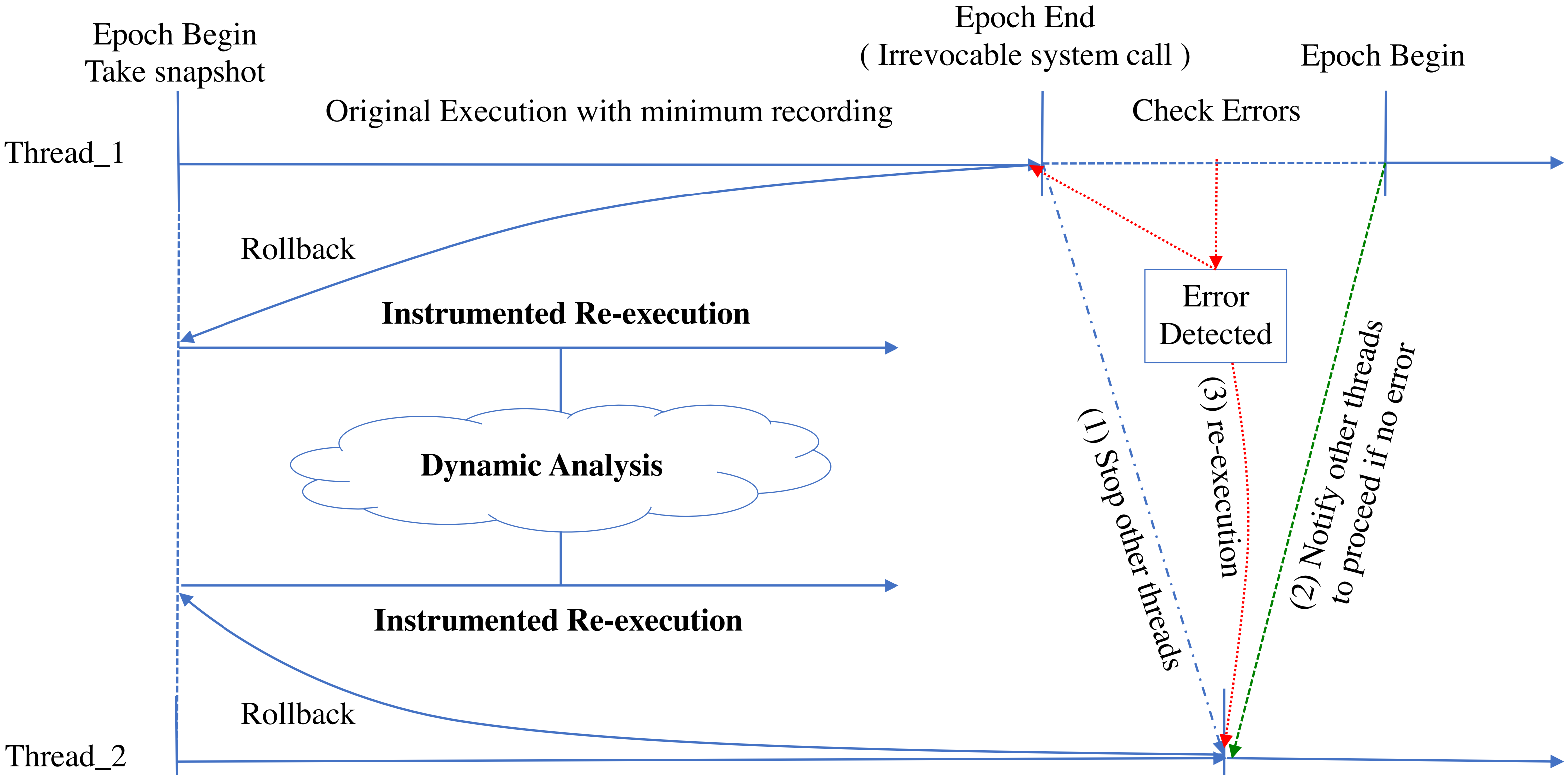
- Ordering every load/store will create significant overhead
- Unnecessary logging overhead if I need not rollback the epoch
- Most code is non-racy

Handling Races in Replays



If replay diverges, possibly caused by races,
iReplayer re-executes until an identical schedule is found!

Overview of iReplayer



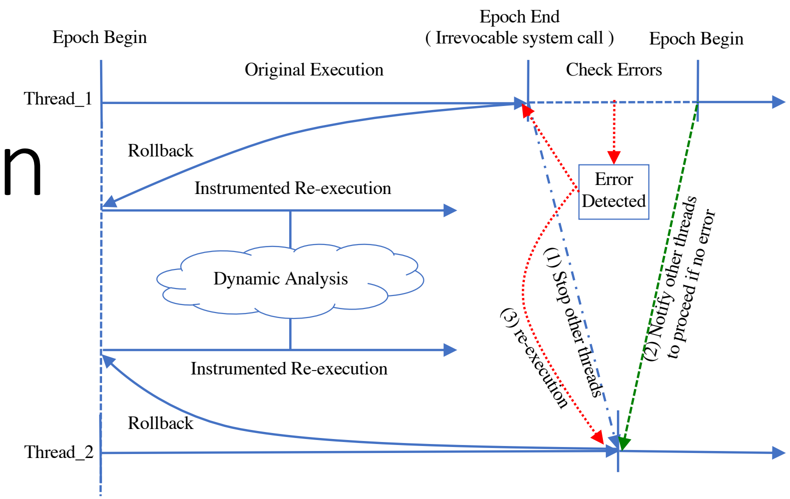
How to achieve in-situ and identical RnR efficiently?

Other Evaluation

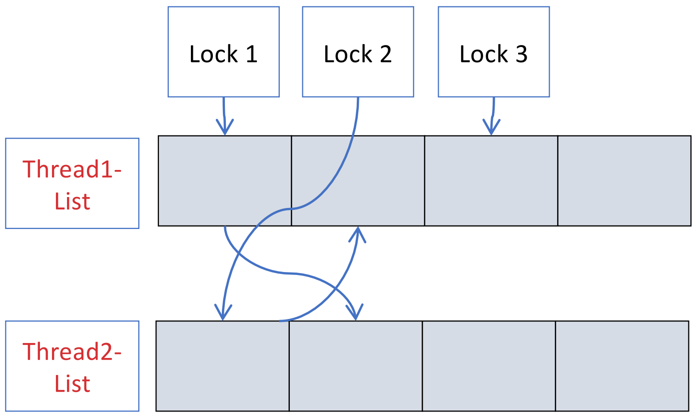
- Identical Re-execution
  - All applications were identically reproduced

- Reproducing the race of Crasher
  - 99.8718%: in one execution
  - 0.1088%: in two executions
  - 0.0121%: in three executions
  - 0.0073%: >= four executions

Conclusion



Category	Syscall Examples
Repeatable	getpid, getcwd
Recordable	gettimeofday, mmap, open
Revocable	file read/write
Deferrable	close, munmap, thread exits
Irrevocable	fork



Handling Synchronizations