Sparse Record and Replay with Controlled Scheduling

Christopher Lidbury
Department of Computing
Imperial College London
London, UK
christopher.lidbury10@imperial.ac.uk

Alastair F. Donaldson
Department of Computing
Imperial College London
London, UK
alastair.donaldson@imperial.ac.uk

Abstract

Modern applications include many sources of nondeterminism, e.g. due to concurrency, signals, and system calls that interact with the external environment. Finding and reproducing bugs in the presence of this nondeterminism has been the subject of much prior work in three main areas: (1) controlled concurrency-testing, where a custom scheduler replaces the OS scheduler to find subtle bugs; (2) record and replay, where sources of nondeterminism are captured and logged so that a failing execution can be replayed for debugging purposes; and (3) dynamic analysis for the detection of data races. We present a dynamic analysis tool for C++ applications, tsan11rec, which brings these strands of work together by integrating controlled concurrency testing *and* record and replay into the tsan11 framework for C++11 data race detection. Our novel twist on record and replay is a *sparse* approach, where the sources of nondeterminism to record can be configured per application. We show that our approach is effective at finding subtle concurrency bugs in small applications; is competitive in terms of performance with the state-of-the-art record and replay tool rr on larger applications; succeeds (due to our sparse approach) in replaying the I/O-intensive Zandronum and QuakeSpasm video games, which are out of scope for rr; but (due to limitations of our sparse approach) cannot faithfully replay applications where memory layout nondeterminism significantly affects application behaviour.

CCS Concepts • **Software and its engineering → Software testing and debugging**; *Concurrent programming structures.*

Keywords record and replay, controlled concurrency testing, data race detection, concurrency

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314635

ACM Reference Format:
Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse Record and Replay with Controlled Scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3314221.3314635

1 Introduction

Controlled concurrency testing has proven successful in finding subtle bugs in concurrent programs, by exploring a diverse set of schedules (see e.g. [33, 62, 64, 78, 87]). However, such techniques are known to be limited by the assumption that the thread scheduler is the only source of nondeterminism. For example, in an empirical study of systematic scheduling algorithms, many benchmark programs, such as Apache's httpd, had to be excluded due to their reliance on external factors such as the network [78].

In contrast, *record and replay* tools aim to capture the external factors that affect the behaviour of a system as the system runs, so that an execution can be faithfully replayed (see e.g. [47, 53, 57, 66]). The degree to which replay is faithful varies, but many systems aim to be extremely thorough by monitoring, intercepting and facilitating replay of virtually all sources of nondeterminism. Unlike controlled concurrency testing, these tools typically leave threads to be scheduled by the regular OS scheduler, recording whatever schedule results. This is fine if a bug happens to be triggered, but does not support systematic or controlled-randomized exploration of thread schedules to find subtle bugs. Faithful record and replay is also difficult from an engineering perspective, often requiring surgical changes to the OS or underlying hardware, and demanding high resource usage due to the many details that must be kept track of.

Our aim in this work is to lift controlled concurrency testing so that it can be applied to larger and more realistic settings, by drawing on ideas from record and replay, sacrificing faithfulness in order to keep overhead low. To do this, we take a *sparse* approach: we assume the relevant sources of nondeterminism affecting an application come from (a) the thread scheduler (including the handling of signals), and (b) input from the network, peripherals such as keyboard and mouse, and well-understood system calls that can be configured for particular applications of interest. We present a record and replay mechanism that captures minimal information about these sources of nondeterminism that suffice to enable efficient controlled concurrency testing for a range of applications. The advantage of this is that execution is efficient and recording overhead is low. The price is that the tool cannot handle systems whose behaviour is influenced significantly by other sources of nondeterminism (e.g. memory layout) without programmer intervention.

We have implemented our approach as a tool, tsan11rec, driven by the following objectives: to maximise the extent to which the concurrency semantics of the system under test can be explored, in order to find subtle bugs; to integrate the tool with state-of-the-art data race detection, capturing an important class of concurrency bugs; and to preserve as much parallelism in the system under test as possible for efficient record and replay. While race detection, controlled concurrency testing and record-and-replay are all valuable techniques in isolation, their combination is potentially extremely valuable to allow the finding of races (thanks to race detection) that arise under rare thread schedules (thanks to controlled concurrency testing) such that the thread schedule and environmental factors leading to the race can be replayed for debugging (thanks to record-and-replay).

We present a large experimental evaluation showing that tsan11rec is effective at reliably finding and replaying weak memory-related bugs on a set of benchmarks previously used to evaluate the CDSchecker concurrency testing tool [64], and competitive in terms of performance with the state-of-the-art rr record and replay tool [66] on a number of larger applications, such as Apache httpd, PARSEC benchmarks and Pbzip. In particular, for programs that rely heavily on parallelism our tool often out-performs rr since our approach allows threads to run in parallel as much as possible, rather than being sequentialized. In some cases sparsity turns out to be *key* to enabling record and replay. For example, closed-source proprietary device drivers make it very difficult to intercept and control all nondeterminism arising in the execution of video games. To find and reproduce bugs related e.g. to game/display communication it would be necessary to control this nondeterminism. However, the nondeterminism typically has no impact on core game logic. To find and replay game logic bugs it is thus profitable to sparsely ignore game/display communication when recording, and allow this communication to proceed freely during replay. We demonstrate this by applying tsan11rec to the SDL-based video games Zandronum and QuakeSpasm, both of which are out of scope for rr due to communication between the game and the OpenGL interface, through ioctl, that rr is unable to record and replay. We show that we are able to record and replay a historical Zandronum bug related to an error in client-server communication when the game is played in internet multi-player mode.

After recapping necessary background (§2), we present our contributions via the following paper structure:

```
void T1() {
  nax = 1;
  x.store(1, std::memory_order_release); // A
  y.store(1, std::memory_order_release); // B
}
void T2() {
  if (y.load(std::memory_order_relaxed) == 1 && // C
      x.load(std::memory_order_relaxed) == 0) // D
    x.store(2, std::memory_order_relaxed);
}
void T3() {
  if (x.load(std::memory_order_acquire) > 0) // E
    print(nax);
}
```

Figure 1. A racy C++11 program using atomic operations

Controlled scheduling (§3) We detail our scheduling approach, which enables exploration of the concurrency semantics of the original program by allowing a context switch after every *visible* operation, splitting up high-level operations into multiple irreducible visible operations. By building on top of tsan11 [52], race detection is seamlessly integrated. We preserve the parallelism as much as possible by only sequentializing visible operations; invisible regions of code in different threads can run in parallel.

Sparse record and replay (§4) We describe our sparse approach to record and replay, focusing on the interplay between controlled scheduling and record and replay, and in particular discuss how the granularity at which nondeterminism is captured and recorded can be adapted on a per application basis.

Experimental evaluation (§5) We present a large evaluation applying tsan11rec to concurrency benchmarks, the Apache httpd, PARSEC and Pbzip applications and benchmarks, and the Zandronum and QuakeSpasm games. Our evaluation includes comparisons with rr.

2 Background

We provide relevant background on controlled scheduling, record and replay, and the tsan and tsan11 race detectors.

Controlled scheduling *Controlled scheduling* techniques (sometimes referred to as *stateless model checking*) override the system scheduler in order to perform schedule-space exploration for an application, e.g. systematically or in a controlled randomized fashion [33, 62, 64, 78, 87]. Exploring interesting schedules can reveal subtle bugs that the system scheduler would trigger with low probability, and having control over which schedules are explored is important for replay of bug-inducing schedules.

Scheduling decisions are made at *scheduling points*, which correspond to *visible operations*: a visible operation is an operation performed by a thread that may influence the behaviour of other threads. As an example, consider the

```
void sig_handler() {
  quit.store(1);
}

void Listener() {
  while (!quit.load()) {
    int res = poll(&server_fds, 1, 100);
    if (res == 0) continue;
    CHECK(res > 0 && server_fds.revents == 1 && "poll error");
    char *buf = new char[100];
    recv(server_fd.fd, buf, 100, 0);
    std::unique_lock<std::mutex> lck (mtx);
    requests.push(buf);
  }
}

void Responder() {
  while (!quit.load()) {
    std::unique_lock<std::mutex> lck (mtx);
    if (requests.size() == 0) continue;
    char *buf = requests.front();
    requests.pop();
    lck.unlock();
    Process(buf);
    send(server_fd.fd, buf, 100, 0);
    delete[] buf;
  }
}
```

Figure 2. Generic client for processing and returning requests sent from some server.

program fragment shown in Figure 1. A controlled scheduler will begin by choosing (via some choice strategy) which of the visible operations A, C or E to schedule first. If it chooses A, then at the next scheduling point the choice is between B, C and E, etc. The non-atomic increments of nax and nay are *invisible* operations, and do not constitute scheduling points.

Record and replay The ability to record and replay has many useful applications, notably allowing consist reproduction of bugs in nondeterministic programs. In general, recording and replaying involves identifying relevant sources of nondeterminism, and enforcing the same resolution of this nondeterminism during replay as was observed while recording. The granularity at which nondeterminism is controlled varies between approaches. To see how useful record and replay is, consider the example program shown in Figure 2. The program receives buffers from a server, processes them, and then sends them back. But what happens if the connection fails or we get a “poll error”? The ability to capture an execution that shows an error by connecting to a real server, and then repeatedly replay the execution *without* having to connect to a real server, allows us to reliably explore the cause of the error. This is particularly useful for larger programs that utilise complicated communication protocols, and have time consuming setups or hard to find bugs.

In general, a program can have many such sources of nondeterminism. Aside from the thread interleaving and value of atomic reads, other sources include interaction with the file system, system calls, certain libc functions (e.g. the conditions under which `malloc` can fail are not deterministic), instructions that query the state of the CPU (such as the x86 RDTSC for reading the processor's time-stamp counter), or in some cases even the value of pointers (e.g. iterating through an ordered container of pointers).

The choice of what nondeterminism to record, and the method of recording and replaying, is a substantial area of research. In this paper we compare our approach with the current state-of-the-art tool, rr [65], which achieves performance overheads compared with native execution of as low as 1.5× for some applications, as well as low storage overheads. It generally enforces a priority-based first come first served strategy for scheduling, with each thread given a time slice before yielding. Execution is sequentialized so that only one thread runs at a time. We present a detailed discussion of other approaches to record and replay in §6.

C++11 dynamic race detection We present our work in the context of C/C++, though our approach to controlled scheduling and record and replay is conceptually more general. The C/C++11 standards define threads, atomics and various constructs for inter-thread communication, together with a description of a memory model that defines how operations are ordered across threads, when synchronisation occurs, and what values may be read from memory [7].

Working with low level atomics and other inter-thread constructs can be very difficult, and their misuse frequently leads to buggy concurrent programs. ThreadSanitizer (tsan) is an efficient dynamic race analysis tool aimed at C++ programs [74]. The tool performs compile-time instrumentation of the source program, in which all (atomic and non-atomic) accesses to potentially shared locations, as well as fence operations, are instrumented with calls into a statically linked run-time library. In general all visible operations, including most libc functions and system calls, are instrumented. This library implements a *vector clock* algorithm for tracking the happens-before relation [28, 48], using shadow memory to keep track of accesses to all locations. Requiring a simple compiler flag to be activated, tsan is very easy to use. It is also fast, with runtime overheads of around 10× to 12× (competitive performance, given the nature of the analysis), although memory overhead can be high.

C++11 features such as atomics and threads are accepted by tsan, but the standard tool ignores most of the semantics of the C++11 memory model. A recent extension, tsan11, has added semantics for a large fragment of the C++11 memory model, allowing it to find races arising due to weak memory behaviours that regular tsan would miss [52]. As an example, consider again the program of Figure 1. For the conditional in thread $T2$ to pass, both the stores A and B in $T1$ must have happened, but an earlier value of x must be read. The end result is that the print in $T3$ will be racy as the the load in $T3$ now reads the value stored in $T2$. This cannot occur under sequential consistency, but can occur under C++11 semantics; the race is detected by tsan11 but not tsan.

A major limitation of tsan and tsan11 is that the executions explored by the tool are at the mercy of the OS scheduler. Bugs that require unusual interleavings to trigger almost never manifest. The tsan11rec tool presented in this work builds on tsan11, combining the strengths of the tsan approach with controlled scheduling and record and replay to allow the detection and reproduction of bugs that would be unlikely to be discovered, and harder still to repeatedly reproduce, using the OS scheduler alone.

3 Scheduling

Recall that one of our main aims in the design of tsan11rec is to combine controlled scheduling with record and replay in a manner that directly incorporates state-of-the-art race detection. We now describe the mechanics of the scheduler; in particular, how the scheduler builds on and interacts with the existing tsan11 instrumentation library, and how the scheduler handles nondeterminism. The mechanics of the scheduler will be important for describing the design of our record and replay facilities in §4.

Rather than using an overarching scheduler thread, details of scheduling decisions are stored in a designated piece of shared state. The threads interact indirectly via this shared state using a protocol, to cooperatively determine when they should be scheduled. The protocol builds on the tsan11 library, and has been designed so that new scheduling strategies can be easily added. We focus on two strategies in this work: *random* and *queue*. With the random strategy, the next thread to schedule is chosen at random at each scheduling point, via a pseudo-random number generator (PRNG) initialized with some fixed initial seed. This provides controlled random scheduling similar to that described in [78]. With the queue strategy, threads are scheduled in a first-come-first-served manner.

3.1 Protocol Details

Recall from §2 that the tsan tool and its tsan11 extension performs compile-time instrumentation of the low-level visible operations, including atomic operations and system calls (syscalls). Each time such an operation is reached, control jumps into a tsan library function. Execution of some of these library functions may be sequentialized, but threads executing outside these library functions—i.e. executing *invisible* operations—are free to run in parallel.

Our tsan11rec tool essentially acts as an additional layer of interception on top of the existing tsan11 instrumentation. On reaching a visible operation, a thread jumps into a tsan11 library function, e.g. `__tsan_atomic_load4` for a 4-byte atomic load, which will additionally call tsan11rec functions that interact with the scheduler logic to determine when the thread can proceed.

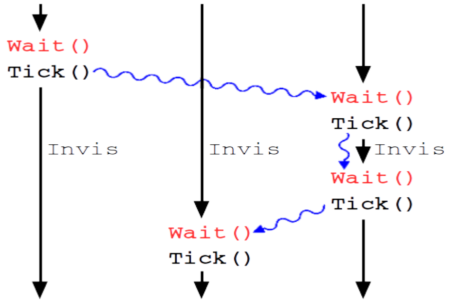

Figure 3. Sequentialized critical sections and parallel invisible operations. Blue wavy arrows represent scheduler-imposed ordering; black arrows represent program order.

Communication with the tsan11rec instrumentation layer uses a protocol based on two functions, `Wait()` and `Tick()`:

`Wait()` – Block this thread until the scheduler activates it.
`Tick()` – Choose a thread to activate.

A thread enters `Wait()` right before it executes a visible operation. Depending on the scheduling strategy used, the state of the scheduler may need to be updated when `Wait` is called: the queue strategy requires the thread to enqueue itself; the random strategy requires no action. If the thread already happens to be the next thread due for scheduling, `Wait()` returns without blocking. A thread enters `Tick()` once it has completed a visible operation. By executing `Tick()`, the thread applies the scheduling strategy (random or queue) to choose the next thread to be scheduled and update the scheduler state to reflect this. In the case of the queue strategy this involves incrementing the current queue position; for the random strategy this involves choosing the next thread id at random. When the thread returns from the `Tick()` function it is free to continue performing invisible operations unhindered until it reaches the next visible operation, where it will enter `Wait()`. This allows for parallelism between threads, as sections of invisible code are unordered. Multiple invisible operations that can execute in parallel are illustrated in Figure 3. Not that, with the random strategy, a thread does not have to reach `Wait()` to be in the list of schedulable threads.

The combination of a visible operation and associated scheduling-related code, wrapped in a `Wait()` and `Tick()` pair, is called a *critical section*.

3.2 Special Cases

The approach described in §3.1 of wrapping the tsan instrumentation for a visible operation in a critical section works directly for most visible operations. We now discuss a number of operations that require extra attention to detail, mainly because their semantics necessitate specific updates to the scheduler state, or because they cannot be represented as a simple critical section.

Calls to functions whose names begin with `intercept_` are already inserted by tsan11 to instrument visible operations. We have modified the implementations of these functions to accommodate our instrumentation.

Thread management Thread creation, deletion and joining are all treated as visible operations. This is because they must update the state of the scheduler, and as such will affect the scheduling of threads going forward. To handle this, we introduce three scheduler functions: `ThreadNew(tid)`, `ThreadJoin(tid)` and `ThreadDelete()`.

The `ThreadNew` and `ThreadJoin` functions are added during instrumentation of thread creation and joining primitives. The `ThreadNew(tid)` function, called by the parent of the newly created thread, enables the new thread within the scheduler. The `ThreadJoin(tid)` function will block itself until thread `tid` has finished, and as such must instead *disable* itself in the scheduler, and also mark itself as waiting on `tid`. On completion, a thread calls `ThreadDelete`, which involves (a) enabling the parent thread if it is waiting for this thread to finish, and (b) disabling itself in the scheduler. All three operations are wrapped in a `Wait()` and `Tick()` pair.

Mutexes The mutex operations *trylock*, *lock* and *unlock* are all visible and require instrumentation. Trylock can simply be wrapped in a `Wait()` and `Tick()` pair as for regular visible operations. Unlock is similar to thread deletion in that it must also re-enable threads that were blocked waiting on the mutex, although in this case we only need to re-enable one of the blocked threads; the thread that is chosen depends on whether the queue or random strategy is being used.

Mutex lock poses an interesting issue in that a thread attempting to acquire a mutex will block if the lock operation fails. To account for this, we modified the instrumented version of mutex lock to be as shown in Figure 4. This changes it to a trylock loop, with a critical section associated with each attempt at locking. Note that this is the native trylock, not the instrumented version. The `MutexLockFail(m)` function is similar to `ThreadJoin`: a thread calling this function disables itself from the scheduler and informs the scheduler that it is waiting on `m`. The thread will then reenter `Wait`, but as it is disabled it will block until it is re-enabled and then scheduled to run. The `MutexUnlock(m)` function is called when a thread releases a mutex, and will re-enable one thread that is disabled due to waiting on `m`.

There is no `Wait` nor `Tick` inside `MutexLockFail(m)` nor `MutexUnlock(m)`. Another thread can acquire the mutex between a thread being re-enabled and it attempting the trylock. This is OK: the thread will simply block itself again.

Condition variables Condition variables allow control over when certain threads will wake up and try to acquire a mutex. When a thread initially acquires a mutex, it may check a condition required for it to proceed, and if it fails, release the mutex and block itself via the condition variable. This thread will only wake up and try to reacquire the mutex when another thread *notifies* it via the condition variable. The conditional is checked via the conditional wait function; waking up one or all of the waiting threads is performed via the signal and broadcast functions, respectively.

```
int intercept_mutex_lock(void *m) {
  int res = EBUSY;
  while (res == EBUSY) {
    Wait();
    res = trylock(m); // native version of trylock
    if (res == EBUSY) {
      MutexLockFail(m);
    }
    Tick();
  }
  return res;
}
```

Figure 4. Instrumented mutex lock.

The conditional wait accepts a timer, determining the length of time after which the thread unblocks itself. This timer represents a *physical* time. This is in contrast to the scheduler's ticker, which represents a *logical* time. This difference between physical and logical means that from the perspective of the scheduler, the conditional's wakeup timer is nondeterministic. Semantically speaking, a thread can wake up from the timer and acquire the conditional's mutex before another thread can, even if the associated time is very long. We choose to handle this by *not* disabling the thread if it calls a conditional wait with a timer. Despite not being disabled when timed, a thread can still *eat* a conditional signal, and so should still mark itself as waiting on the conditional in the scheduler.

For the three conditional functions we provide corresponding `CondWait(m, t)`, `CondSignal(m)` and `CondBroadcast(m)` scheduler functions. `CondSignal(m)` and `CondBroadcast(m)` simply wake up one thread and all threads waiting on `m` respectively. The interaction between the instrumentation and the scheduler for conditional signal and broadcast is simple, with the instrumentation simply calling the scheduler function in a `Wait()` and `Tick()` pair.

Conditional wait is a little more involved, and details are shown in Figure 5. Between the `Wait` and the `Tick`, a thread informs the scheduler that it is performing a conditional wait, via `CondWait`. This informs the scheduler that the thread is either blocked waiting for a signal, or performing a timed conditional wait, so that while not blocked it can nevertheless eat a signal. The thread then releases the mutex, informing the scheduler via the `MutexUnlock` scheduler function described earlier that this has been done. Finally, the thread enters the intercepted version of `mutex_lock`, described above. Because this starts with a `Wait`, in the case of an untimed signal, the thread will block until it is re-enabled by a conditional signal or broadcast. By using distinct critical sections

```
void intercept_cond_wait(void *m, bool timed) {
  Wait();
  CondWait(m, timed);
  mutex_unlock();
  MutexUnlock(m);
  Tick();
  intercept_mutex_lock();
}
```

Figure 5. Instrumented conditional wait.

to separate a thread marking itself as being blocked on a signal, and attempting to reacquire the mutex, we allow the possibility for another thread to be scheduled in between, possibly acquiring the mutex.

Once a thread has reacquired the mutex, it will typically recheck the condition it was originally waiting on. If it is not satisfied, it will call `cond_wait` again. This is where the risk of deadlock comes in, as if it was the only thread signaled and it reenters `intercept_cond_wait`, it will not signal other threads first, and all threads that are blocked by the conditional will remain blocked. We want preserve any potential deadlocks in the underlying program, and so do not limit this from happening; we are also careful not to introduce new deadlocks.

Signals We briefly describe our treatment of signals and signal handlers, noting that these are distinct from the signals associated with conditional wait operations described above. We focus on *asynchronous* signals, which can be received by processes at any time, and thus contribute an additional source of nondeterminism. This is distinct from *synchronous* signals, e.g. SIGSEGV, which are raised by the thread as and when certain operations are performed. Unlike in the case of e.g. a memory load operation, which has a designated program point that can be intercepted to facilitate interaction with the scheduler, a signal can arrive at *any* time. The standard also specifies a `signal` function, that binds a handler function to a specific signal.

Scheduling with signals is handled by simply marking the entrance to the signal handler, and the aforementioned `signal` function, as visible operations. From a scheduling perspective, besides the arrival of a signal, signals are not a problem. Recording and replaying signals is where things become difficult, which we discuss in §4.3.

3.3 Liveness

While the scheduler strives to ensure that all possible program behaviours can be explored in principle, in practice, depending on the strategy, this can lead to massive slowdowns in particular cases. For example, suppose a thread is scheduled and undertakes a vast number of invisible operations, or calls a sleep function for some duration, before finally issuing a visible operation. If all other threads end up blocked waiting to perform visible operations and the scheduler doesn't give them a chance to run, the performance of the program may be drastically impacted. This can become particularly problematic when dealing with programs that rely on responsiveness, such as real-time applications.

To cope with this, we slightly reduce the scheduler's ability to explore any possible schedule by having the scheduler *force* a reschedule in such cases. By forcing a reschedule after *n* milliseconds, the probability of exploring a schedule whereby a thread performs two visible operations consecutively separated by more than *n* milliseconds is greatly reduced. Conveniently for us, tsan has a background thread, which we configure to call our `Reschedule()` function every *n* milliseconds, for some given *n*. Because the `Reschedule()` function relies on physical time, it introduces nondeterminism into the scheduler.

4 Record and Replay

We now discuss the record and replay mechanism we have implemented, with the aim of answering the following questions: What do we mean by "recording" an execution and then replaying said execution later? How do we formalise a recording? What makes replaying an execution *valid*? What should we record and what should we *not* record?

When a program executes, certain visible operations will lead to nondeterminism. Recording an execution therefore means capturing information about these visible operations in a form that can be used to reproduce the execution during replay. We call this captured information the *demo file*, or *demo* for short. We also refer to an execution that is replaying a demo as a *replay*, and that the replay is *synchronised*, unless something has gone wrong with the replay such that execution has diverged from what was recorded, in which case we say it is *desynchronised*.

This leaves us with the question of what it means for a replay to desynchronise. A demo is defined as a series of *constraints* arising from the recorded execution, which the replay is required to satisfy. The tool will attempt to enforce these constraints on the program during replay, and as long as it can, the replay is deemed to be synchronised. If at any point the tool is unable to enforce a constraint on the program, we say the replay has *hard desynchronised*, in which case the tool will abort. In some cases, the replay may abide by the constraints, but appear to diverge from the recorded execution, for example, by producing console output in a different order. We call this *soft desynchronisation*. To illustrate an extreme case of this, the empty demo is trivially synchronised for any replay, but will lead to soft desynchronisation practically everywhere unless the system under test is highly deterministic.

The record and replay mechanism is built into the scheduler we discussed in §3. In cases where a nondeterministic choice needs to be made that is unrelated to scheduling (and so not handled by the queue nor random strategy) a PRNG is used, seeded by two calls to `rdtsc()`.

For the most part, tsan11rec avoids the need for user annotation, however, there are some cases where they are unavoidable; this is shown in §5.4.

4.1 Motivating Example

To help lay out the reasoning and technical explanation given in the rest of this section, we start off with an example program. The program fragment in Figure 2 (already discussed briefly in §2) shows a simple client that repeatedly receives a `char` buffer from a server, applies a transformation to it, then sends it back. We discuss the behaviours of the application that need to be recorded in order to enable faithful replay, vs. those features that need not be recorded.

What to record The obvious case here is the interleaving of threads. Recording this will ensure that the order of operations to the atomic locations `quit` and `mtx`, and the order of the syscalls used throughout will be the same during replay. Other operations are invisible, and thus will not affect other threads or introduce nondeterminism.

System calls that interact with the environment can be seen as inputs to the program, which in this case determines how many requests to handle and the contents of each request. For example, `poll` informs us on whether there is data to be read from the server, and thus needs to be recorded, as do the system calls `recv` and `send`.

The signal handler in this example is used to trigger the end of the program. The arrival of the signal is asynchronous, and comes from outside the program. During replay the tool will need to ensure that the same signal arrives at the same point in logical time.

What to ignore The first, and likely most contentious element, is the layout of memory. This will of course depend on the program in question, but in this example, and most of the programs we encounter, the position of objects in memory will have no effect on the rest of the program. If the request queue was instead an ordered set of `char` pointers, then it would matter, as the pointer values will determine the order in which requests are considered during iteration.

4.2 Interleaving

As explained in §4.1, the ordering of visible operations must be preserved during replay. We describe how the random and queue strategies store ordering-related information.

To recap the strategies described in §3: the random scheduler chooses which thread to allow to run the next visible operation randomly after each visible operation has completed, while the queue scheduler is first come first serve for whichever threads attempts to perform a visible operation.

For the random strategy, the entire thread interleaving is encapsulated in the PRNG. Therefore, no information besides the two seeds used for the PRNG is required.

For the queue strategy, the ordering during record depends on the order in which threads happen to reach `Wait()`, which depends on physical timing. To encode the order so that it can be enforced during replay, a file called QUEUE is used. This file records (a) a map specifying, for each thread id, the first tick at which the thread should be scheduled, and (b) an ordered list of ticks to be consumed by threads each time they leave a critical section—the tick that a thread consumes on leaving a critical section informs the thread as to the next tick at which it is to be scheduled. Run-length encoding is used to efficiently record the case where a thread is scheduled multiple times in succession.

There is clearly a trade-off between these strategies. Where the random strategy stores no data, the queue strategy may need to store data on every visible operation. The queue strategy will be much faster however, as it is unlikely to be blocked in `Wait()` unless another thread is already critical.

4.3 Signals

We discussed signals briefly in §3.2, but deferred discussion to this section as most of the difficulties are in attempting to replay them. Synchronous signals are ignored (e.g. SIGSEGV, SIGPIPE) as these should reoccur at the same point in the execution without the help of our tool. To clarify, it is *entering the signal handler* that is the visible operation. When inside the signal handler, a thread cannot interact with the rest of the process except though atomic operations, which are themselves visible. From this, we can say that it does not matter at which point between a `Tick()` and following `Wait()` pair that the signal handler is entered.

In tsan11rec, any asynchronous signal that arrived during recording becomes a synchronous signal upon replay. When a thread receives a signal, it records the value of the tick seen during the most recent `Tick()`, along with the signal value in a file called SIGNAL. For example, consider the case in Figure 2 where the `Responder` thread, T2, has just performed the atomic load on tick 5, but has not yet attempted to acquire the lock. It receives the signal and performs a `Wait()` and `Tick()` so that it can enter the signal handler. The SIGNAL file will therefore have the line "`2 5 15`", indicating that thread T2 receives signal 15 at tick 5. During replay, when the `Responder` thread calls `Tick()` during tick 5, it will raise signal 15 itself at the end of `Tick()`. It does not matter at which precise point between `Tick()` and the following `Wait()` that the signal arrived at during recording; it will float to the end of `Tick()` as shown in Figure 6.

4.4 System Calls

As discussed in §3.2, system calls are a significant source of nondeterminism in an application. To ensure that relevant properties of an application are preserved during replay, we need to record the results of relevant system calls. This is a fundamental challenge that any record-and-replay system must face, and state-of-the-art tools such as rr [66] aim to

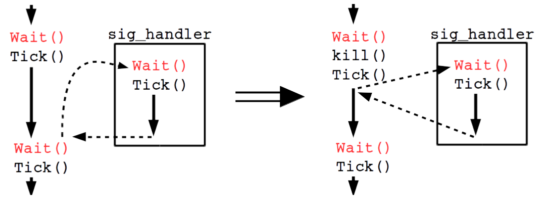

Figure 6. Signals are replayed immediately after the preceding tick.

be as comprehensive as possible in the system calls they support, so that they can be applied directly to a wide range of applications. In contrast, the idea behind our sparse approach to identify a minimal subset of system calls such that recording these system calls suffices to enable faithful replay of particular applications of interest. At a high level, we approached this by incrementally adding support for system calls based on a trial-and-error process of first using `strace` to understand the full set of system calls issued by an application, and then repeatedly attempting to record and replay the application with tsan11rec, incrementally adding support for additional system calls through analysis of the sources of replay failures. We emphasise that this process did require quite some manual effort, and would need to be iterated further to handle applications with significantly different system call requirements compared to our case studies.

The term syscall is a bit of a misnomer, as both tsan and the instrumentation detailed in this paper will instead intercept the glibc wrappers around the syscalls, instead of the syscall directly. These glibc functions are much easier to use on behalf of the programmer, as they will take care of system specific details, pushing the arguments onto the stack and interpreting the results returned by the kernel. We still use syscall throughout, as it is in the underlying syscall where the nondeterministic behaviours occur.

Each syscall takes a variable number of user-allocated buffers and fills them with the appropriate data, before setting `errno` and returning some value. As these are sources of nondeterminism, the return value, `errno` and any appropriate buffers will be compressed and stored in a demo file called `SYSCALL`. During replay the actual data returned will be overwritten by the data in `SYSCALL`. Only the interaction with the SYSCALL file is part of the critical section, which reduces contention in the scheduler.

As an example, consider the `Listener` thread in Figure 2 performing the `poll` and `recv` syscalls in succession. The return value, error number and two elements in the `server_fds` structure must be stored for `poll`; the return value, error number and contents of the buffer must be stored for `recv`. These will be treated as character buffers and have a simple run length encoding applied.

One of the difficulties that arises from adding a syscall is the *knock-on* effect it can have with respect to other syscalls. Consider, for example, the syscalls that interact with the filesystem, `create`, `open`, `read`, etc. If you intercept `open`, on the assumption that you may not have a valid file descriptor during replay where you did during record, you will then have to intercept every syscall that works with that file descriptor. What starts out as a single interception becomes potentially hundreds of intercepted syscalls. There is a delicate balance to be struck between those syscalls that need to be recorded to make important execution features deterministic during replay, vs. those syscalls that are better left un-recorded because (a) determinism of replay does not depend on them being recorded, and (b) recording them leads to a snowball effect where many other syscalls must also be recorded to avoid desynchronisation.

We have added syscall support to tsan11rec in a demand-driven manner by using `strace` to identify important calls for key applications of interest. The current set of syscalls supported includes `read`, `write`, `recvmsg`, `recv`, `sendmsg`, `accept`, `accept4`, `clock_gettime`, `ioctl`, `select` and `bind`. These have allowed us to get a significant number of applications up and running, including the applications studied in our evaluation, modulo a few workarounds (detailed in §5); these applications issue many additional system calls that we have found it unnecessary to record—unnecessary in the sense that simply re-issuing the system call during replay has no observable effect on the application's behaviour. Sometimes whether a call must be recorded depends on the file descriptors that the call receives. For instance, for all of our case studies it never proves necessary to record `read` and `write` calls whose file descriptors correspond to files in the file system, but it is necessary to record these calls if the associated file descriptors are associated with pipes used for inter-process communication. Rather than this set of syscalls being a starting point towards full syscall coverage, our view is that efficient record and replay that preserves parallelism can benefit from selective syscall recording, based on application-specific knowledge, and we envision the tool supporting a core set of essential syscalls, and being configurable with support for further syscalls to suit particular record and replay scenarios. For example, to handle a program such as `htop` would require instrumentation of the interaction with the `/proc` filesystem, but doing this in the general case would be wasteful, and maybe even harmful if future calls depended on this interaction.

4.5 Asynchronous Events

Asynchronous events are specific events that do not fit in with any of the other categories discussed. An important characteristic is that they are *not* wrapped in a `Wait()` and `Tick()`, either because it was infeasible to do so during recording, or because it would create a lot of unnecessary overhead. These events still need to be replayed to ensure the replay remains synchronised. Currently there are two types of events: `Reschedule` and `Signal_wakeup`.

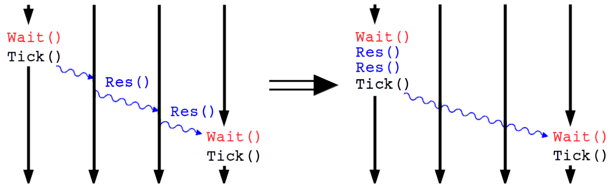

Figure 7. Right shows how reschedules are floated above the Tick().

The reschedule event was discussed in §3.3. It is necessary to include this to ensure that the PRNG will be called the same number of times in each critical section. To see why the signal wakeup event is necessary, consider again a signal being received in the context of the example of Figure 2. This time, suppose the Responder thread receives the signal while it is disabled trying to acquire the lock. Assume that the thread disabled itself on tick 10, the signal arrives and the thread re-enables itself during tick 12, and then enters the signal handler on tick 14. It is *not* OK for the thread to simply not disable itself on tick 10 during replay, as the pool of enabled threads for the scheduler to choose from during ticks 10 and 11 is different between recording and replaying, which will affect the choice the scheduler will make.

As with signals, all asynchronous events are replayed synchronously, with all events that occur between a Tick() and the following Wait() floating up to the previous Tick(). This is shown in Figure 7. These events are stored in the ASYNC file.

5 Evaluation

To evaluate the controlled scheduling abilities of tsan11rec, we compare the strategies on the CDSchecker benchmark suite (§5.1). We then use larger applications to compare tsan11rec with rr [66]. Of the few record-and-replay tools that are publicly available, we chose to compare with rr because (a) it represents the state-of-the-art, (b) it is similar to tsan11rec in terms of the kinds of applications it aims to support. We consider programs that bring challenges related to networking, signals, I/O and real-time constraints: Apache's httpd web server (§5.2), the PARSEC benchmarks and pbzip (§5.3), and two first-person shooter games built on the SDL library (§5.4). The SDL case studies showcase applications that tsan11rec can handle that are out of scope for rr, due to communication between the game and the OpenGL interface. In contrast, we also discuss practical limitations of tsan11rec that rr does not face related to the SQLite database application and Firefox's SpiderMonkey (§5.5).

Throughout, we describe how we evolved the sparse recording facilities of tsan11rec, and many practical challenges we faced along the way; our experience is that such challenges, which seem fundamental to record and replay, are typically described only briefly if at all in the literature. We hope our exposition will be valuable to other researchers.

Common experimental setup All experiments were run under Ubuntu 14.04 LTS on an Intel i7-4770 8x3.40GHz platform with 16GB RAM. The tsan11 and tsan11rec tools were built on top of clang revision 286384. The version of rr used is 5.1.0. As a key goal of our work is to apply race detection to record and replay with controlled concurrency testing, most of the testing is done with race detection enabled, even when using rr. We still show times for rr without race detection for reference. We use *native*, *rr*, *tsan11*, *tsan11+rr* and *tsan11rec* to refer to a program running without instrumentation, under rr, with tsan11 instrumentation, under rr with tsan11 instrumentation, and under tsan11rec, respectively.

5.1 CDSchecker Litmus Tests

Overview The small programs (roughly 100 LOC each) used in prior work to evaluate CDSchecker [64] are useful to assess whether tsan11rec's controlled scheduling improves on tsan11's ability to find races (including races related to weak memory). As these programs are closed, the scheduler and memory model are the sources of nondeterminism.

Experimental setup The experiments are run in the following four modes: *tsan11*, where tsan11 (which does not use controlled concurrency testing) finds races; *tsan11 + rr*, where tsan11 finds races with rr recording; and *tsan11rec rnd* and *tsan11rec queue*, where tsan11rec finds races using the random and queue strategies, respectively. We measure the runtime of each tool on each benchmark, averaged over 1000 runs, reporting standard deviation and remarking on the coefficient of variation (CV)—the ratio of the standard deviation and mean.

Results and discussion Table 1 summarises the results. The *Time* columns show mean execution times, with standard deviation. Because these are short-running tests, whose behaviour depends intimately on the manner in which threads interleave, the variance across runs is fairly high, with the CV usually exceeding 1, with the exception of the longer-running results for rr, for which the CV is always less than 1 and usually less than 0.5. *Rate* columns show the percentage of all executions that exposed a data race.

Comparing the tsan11rec rnd results with the tsan11 and tsan11rec queue results, we see that randomized controlled scheduling means tsan11rec finds more races across all benchmarks except chase-lev-deque and dekker-fences. This is because tsan11 runs at the mercy of the OS scheduler, which tends to explore similar schedules on repeated runs, and in these small programs typically causes the main thread to run to completion before other threads are scheduled. The price for this is higher runtime, e.g. mcs-lock and ms-queue suffer slow-downs of around 2× compared with tsan11; we

	tsan11 + rr		**tsan11**		**tsan11rec rnd**		**tsan11rec queue**	
Test	**Time**	**Rate**	**Time**	**Rate**	**Time**	**Rate**	**Time**	**Rate**
barrier	590 (14.45)	0.0%	8 (9.14)	0.0%	4 (5.23)	37.5%	6 (4.99)	0.0%
chase-lev-deque	579 (166.69)	0.5%	0 (1.94)	5.9%	1 (3.07)	0.2%	2 (3.78)	0.0%
dekker-fences	1776 (1208.54)	49.9%	2 (4.02)	50.3%	4 (4.83)	38.7%	3 (4.46)	52.8%
linuxrwlocks	579 (126.45)	0.3%	2 (4.18)	0.1%	4 (4.93)	62.4%	3 (4.50)	0.0%
mcs-lock	574 (13.71)	0.0%	3 (4.45)	0.0%	5 (5.23)	77.0%	3 (4.47)	0.1%
mpmc-queue	574 (13.71)	0.0%	3 (4.49)	0.0%	5 (5.00)	60.5%	3 (4.46)	0.0%
ms-queue	3093 (100.19)	100.0%	91 (64.13)	100.0%	93 (80.68)	100.0%	52 (62.60)	100.0%

Table 1. Comparison over CDSchecker benchmarks between tsan + rr, tsan11 and tsan11rec with controlled random and queue scheduling. Each benchmark was executed 1000 times in each mode. The "Time" columns shows the mean execution time (ms) and standard deviation (in parentheses). The "Rate" column shows the percentage of runs that exhibited data races.

attribute this to the total ordering of visible operations imposed by tsan11rec. The rr results show huge increases due to a constant overhead applied to all programs. But as rr is designed for larger applications, this overhead will usually become insignificant in more realistic examples.

We examined a trace from chase-lev-deque to understand why tsan11rec rnd detects fewer races than tsan11. We found that from the creation of thread 2 to the point of the race, thread 1 must perform 29 operations before thread 2 performs just 4 operations in order for the race to manifest. The probability of this happening under uniform random scheduling is very low. We were able to coerce a race report out of the program by moving the creation of thread 2 to later in the program. This shows that different scheduling strategies will affect how effective we are at finding data races, and that *probabilistic concurrency testing* (PCT) can be effective at prying out concurrency bugs [12].

5.2 httpd

Overview Apache's httpd [3] is a widely-used modular http server that makes heavy use of concurrency to handle many simultaneous connections. For record and replay it is of further interest due to its dependence on external network input. We were able to handle httpd by capturing the system calls described in §4.4, with one workaround: the `accept` system call, which listens for incoming connections, relies on `epoll_wait` to listen for events. This returns user-allocated pointers, file descriptors, and other data in a union with no easy way of knowing the active member, something which tsan11rec cannot currently handle. We worked around this by using httpd's option to switch to a simpler but slightly less efficient syscall, `poll`, which instead simply listens to file descriptors; the results presented here employ this workaround. A strength of rr is that it can handle httpd without this workaround, due to its non-sparse record and replay mechanism.

Experimental setup We tested httpd version 2.4.28 in single-process-multiple-thread mode using `ab`, an Apache-provided program for server stress testing. We sent 10,000 queries across 10 concurrent threads to an httpd server for each of the setups shown in Table 2, averaging results over 10 runs. We report on standard deviation and again remark on the CV. In the table, *rnd* and *queue* refer to configurations of tsan11rec, and the presence or absence *+ rec* indicates whether recording was enabled.

Results and discussion The results are shown in Table 2. The columns under *Race reports* show regular results with race reporting enabled; the data under *No reports* shows results where race-checking-capable tools do perform race checking behind the scenes, but do not actually emit race reports. We make this distinction because tsan11 detects so many races that the overhead of generating reports noticeably affects performance; results when fewer races are detected are more representative of the performance one would expect using a future version of httpd in which many races are fixed. The *Throughput* columns indicate the mean number of queries the server responds to per second. The *Rate* column is the mean number of race reports generated (only relevant for tsan11-based configurations). For each, standard deviation is shown in parentheses. Variance, as measured by CV, is low: below 0.8 in all cases and usually less than 0.5. The *Overhead* columns indicate how much slower performance is compared with native execution.

Without reporting, tsan11 already incurs a 3× overhead compared to native. Comparing results for *native* with *rnd* and *rnd+rec*, we find that adding controlled random scheduling increase this overhead massively, to between 79–89× depending on whether recording is enabled. This is in the same ball park as the overhead associated with rr: 61× without race checking and 160× with tsan11 instrumentation (but still with the actual reporting of races disabled). In contrast, when our queue strategy is used, the overhead compared with *native* drops to 9× and 21× with recording disabled vs. enabled. We attribute the gap between rr/rnd and queue to httpd's heavy reliance on parallelism and frequent use of shared mutexes. This parallelism is removed by rr because the tool sequentializes the execution of threads, while our random scheduler only allows the thread that it has chosen

	Race reports			No reports	
Setup	**Throughput**	**Overhead**	**Rate**	**Throughput**	**Overhead**
Native	N/A	N/A	N/A	28895 (4622.56)	1×
rr	N/A	N/A	N/A	475 (6.08)	61×
tsan11	3687 (294.28)	8×	113 (19.85)	9824 (1432.01)	3×
tsan11 + rr	86 (63.20)	336×	34 (16.80)	181 (46.37)	160×
rnd	141 (8.92)	205×	162 (38.78)	367 (33.26)	79×
queue	818 (310.33)	35×	381 (73.62)	3261 (843.67)	9×
rnd + rec	142 (11.85)	203×	155 (31.03)	326 (38.94)	89×
queue + rec	513 (85.34)	56×	360 (64.28)	1387 (249.61)	21×

Table 2. Comparison of throughput and race rate between native, tsan11, rr and tsan11rec for Apache's httpd. Results are averaged over 10 runs. "Throughput" shows mean throughput in queries per second, "Rate" is the number of races detected per run (where relevant). Standard deviations are shown (in parentheses). Overhead is computed relative to native throughput.

to be scheduled next to execute a visible operation, even if many other threads are ready to execute visible operations. In contrast, the queue strategy allows threads to perform visible operations largely on demand. Turning to the results with race reporting enabled, we see that the queue strategy has the highest race detection rate, improving on uncontrolled tsan11. All other race detecting configurations lower the race detection rate; we believe this is because rr and rnd reduce the number of queries being responded to concurrently.

Comparing demo file sizes when recording is enabled, the tsan11rec demo files are around 48MB for both strategies, dropping to 4.8MB when only 1000 queries are issued, suggesting that demo file size increases linearly with the number of requests at a rate of around 4.8KB per request. This could be reduced further with a more aggressive compression strategy, but would likely increase the time overhead. The demo file for rr is significantly smaller: 6.6MB for 10,000 queries, which goes down to 3.9MB with 1000 queries, implying a rate of around 0.3KB per request plus a constant 3.6MB.

5.3 PARSEC and pbzip

Overview We next turn to the PARSEC benchmark suite [11] and pbzip application [68], both widely used for evaluating concurrency analysis tools. For PARSEC, we consider the benchmarks used to evaluate iReplayer [53], however, three of these would not work on our system (dedup and swaptions do not compile, and canneal crashes).

Experimental setup Each PARSEC (version 3.0) benchmark was run with the 'simlarge' test size shipped with the benchmarks, using 4 threads. Pbzip (version 2-1.1.13) was used to compress a 400MB file with 4 threads. We ran each benchmark 10 times per tool configuration and report average runtimes. We report on standard deviation and again remark on the CV. A small number of races were discovered for some benchmarks, and the race detecting tools largely agreed on the number of races; we do not detail these further.

Results and discussion Table 3 shows the average time taken to run each benchmark with each tool configuration, with standard deviation. Variance, as measured by CV, is reasonably low (CV is always below 1). For the tsan11rec results, *+ rec* indicates whether recording was enabled. Table 4 is computed from the data of Table 3, and reports the overhead associated with running using each tool configuration compared with native execution.

With the exception of bodytrack and fluidanimate, the overhead tsan11rec brings over that of tsan11 is small, and for all benchmarks whether recording is enabled or not makes little difference. However, the overhead associated with tsan11 + rr (i.e., running tsan11-instrumented code under rr) is significant compared with the tsan11 overhead alone, despite the fact that running under rr without race detection is generally efficient. Interestingly, rr *without* race detection performs less well on blackscholes compared with the tsan11rec configurations. Digging into this, we found that the benchmark distributes work between threads at the start of execution and then lets threads run with little interaction. This high parallelism/low communication execution plays to the strengths of tsan11rec, where invisible operations are left to run in parallel, but is bad for rr, which forces sequentialization across all operations.

5.4 SDL-based Games

Overview Simple DirectMedia Layer (SDL) is a library that consolidates input, graphics and various other forms of I/O under a single interface [75], and is typically used for games. On Ubuntu 16.04, SDL communicates with X11 for I/O, pulseaudio for sound and OpenGL for display. We investigated record and replay for two SDL-based games: Zandronum [73], a multi-player Doom port (≈400kLOC), and QuakeSpasm [71] (≈88kLOC), a port of Quake. While these games support custom record and replay by logging high level commands, by working at the threading and system call level tsan11rec can facilitate record and replay of bugs that rely on low-level interactions to manifest. We discuss below successful record and replay of a bug in Zandronum that arises due to communication of game data between the game client and server, which is not present in the game's native replay.

					tsan11rec			
Program	**native**	**tsan11**	**rr**	**tsan11 + rr**	**rnd**	**queue**	**rnd + rec**	**queue + rec**
pbzip	9.2 (0.31)	11.7 (0.49)	66.4 (3.11)	77.2 (1.87)	18.1 (0.30)	12.3 (0.66)	18.2 (0.30)	12.9 (1.29)
blackscholes	0.4 (0.03)	0.8 (0.07)	1.0 (0.01)	2.0 (0.00)	0.7 (0.07)	0.7 (0.06)	0.7 (0.07)	0.7 (0.07)
fluidanimate	0.8 (0.04)	16.0 (1.27)	2.1 (0.01)	38.5 (0.49)	46.4 (3.39)	39.0 (2.67)	50.4 (2.09)	39.8 (1.96)
streamcluster	1.7 (0.19)	38.8 (4.41)	113.3 (3.13)	181.1 (1.79)	103.0 (0.59)	48.5 (2.69)	102.8 (0.18)	41.6 (2.61)
bodytrack	0.5 (0.02)	7.2 (0.36)	3.8 (0.94)	32.7 (0.94)	50.0 (0.40)	7.3 (0.33)	50.0 (0.78)	7.8 (0.23)
ferret	1.2 (0.07)	14.0 (0.86)	8.7 (0.44)	81.5 (2.20)	16.4 (0.66)	14.6 (0.60)	16.7 (0.71)	14.5 (0.37)

Table 3. Execution times (s) for tool configurations across the pbzip and PARSEC benchmarks, averaged across 10 runs. Standard deviation is shown (in parentheses).

					tsan11rec			
Program	**native**	**tsan11**	**rr**	**tsan11 + rr**	**rnd**	**queue**	**rnd + rec**	**queue + rec**
pbzip	1.0×	1.3×	7.2×	8.4×	2.0×	1.3×	2.0×	1.4×
blackscholes	1.0×	2.0×	2.7×	5.3×	1.9×	1.9×	1.9×	1.8×
fluidanimate	1.0×	20.3×	2.7×	48.9×	59.0×	49.7×	64.2×	50.6×
streamcluster	1.0×	22.4×	65.4×	104.5×	59.5×	28.0×	59.3×	24.0×
bodytrack	1.0×	13.5×	7.2×	61.4×	93.8×	13.8×	93.9×	14.7×
ferret	1.0×	11.9×	7.4×	69.5×	14.0×	12.5×	14.3×	12.4×

Table 4. Overhead compared with native execution for tool configurations across the pbzip and PARSEC benchmarks, computed from the results of Table 3.

Our initial attempts to replay these SDL-based games failed due to communication between the application and the closed and proprietary NVIDIA OpenGL module on our experimental platform via `ioctl` syscalls. We worked around this by ignoring `ioctl` during recording, and letting it run natively during replay. This works because communication with the display driver has no effect on the game logic. Display interaction led to further problems with initialization of the input module. We resorted to adapting the scheduler to let the application run uninstrumented until SDL module initialization had completed, adding a custom scheduler hook to allow the application and scheduler to synchronize related to this. These problems are not specific to our approach or tool—indeed rr cannot handle these SDL-based games for similar reasons—but are rather a fundamental limitation of recording and replaying applications that make heavy use of I/O. To handle such applications, one either needs to fully mock out I/O components, requiring a tremendous engineering effort, or carefully determine those components that should not be instrumented and specify this via annotations.

With these workarounds we were able to handle both games such that gameplay is displayed on screen during replay; gameplay would not be visible if the I/O subsystem had been mocked out, and visibility might be useful in debugging problems that manifest as visual artifacts.

Experimental setup Measuring game performance in a way that allows us to compare the overhead of our scheduling strategies is non-trivial. The only metric we have is the frame-rate (fps)—the number of frames drawn to the screen per second. QuakeSpasm and Zandronum are capped at 60 fps, and will try to maintain this frame-rate, dipping if they cannot keep up. If the frame-rate is reduced too much, the games become unplayable. As a best-effort evaluation mechanism, we report on whether the games are playable under various tool combinations. Additionally, we found that it was possible to remove the frame cap for Quakespasm, so we report ball-park figures for the overheads of various tool configurations when playing this game un-capped. (We could not find a way to reliably remove the frame cap for Zandronum.) As mentioned previously, we do not compare with rr, as it cannot record or replay the games. We used Zandronum revision 10013:dd3c3b57023f updated to use SDL2, QuakeSpasm version 0.93.0, and SDL version 2.0.5.

Results and discussion With the random tsan11rec scheduler, Zandronum was unplayable even with recording disabled: the frame-rate dropped to below 1 fps. This is due to the random scheduler starving the main thread by frequently scheduling other less critical threads (e.g. the audio thread). In contrast, the queue scheduler could maintain the full 60 fps with recording enabled; for 100 seconds of play the demo size grew to just under 8MB, of which 6.5MB was for syscalls.

To test tsan11rec's ability to replay network communication, we found a previously fixed Zandronum bug [88] that relies on an error in this communication to manifest. This bug involves incorrect game state information being sent from the server to the client during a map change. We replicated the bug with a server and two clients, one of which was recording. After about 12 minutes the bug appeared and resulted in a demo file of 43MB. We then replayed the demo and the bug appeared as expected. This demonstrates that our tool can be used to accurately capture and facilitate replay of bugs in large networked applications.

For QuakeSpasm, we found that it was possible to play the game without dropping below 60 fps using tsan11 and all tsan11rec configurations. To further investigate the overhead of each tool configuration on this case study, we removed the fps cap, then played the game 5 times per tool configuration, for 90 seconds per play, enabling a mode where the game's fps is periodically appended to a file. We made a best effort to play the game in a similar manner on each run, but inevitably there will still be high variation in game activity between plays. Indicative results are shown in Table 5. The rnd and queue configurations refer to tsan11rec with the random and queue strategies, respectively, and with recording disabled, while the "+ rec" tool configurations are similar but with recording enabled. The "Overhead" column shows the overhead observed compared with native execution. The take-away from these results is that the instrumentation overhead for both tsan11 and tsan11rec is surprisingly modest (generally less than 2×), and that the additional overhead associated with enabling recording in tsan11rec is low.

5.5 Limitations: SQLite and SpiderMonkey

A downside of our sparse approach to record and replay is that different applications may have incompatible requirements regarding what should be recorded and what must *not* be recorded. For example, recording memory layout and attempting to enforce the same layout on replay would not only slow down the SDL games (see §5.4) to the point of being unplayable, but would also cause problems related to communication with the display driver. Yet, the behaviour of some programs will depend on the memory layout, such as iterating over an ordered C++ container that holds pointers.

In particular, we experimented with applying tsan11rec to the SQLite database management library [76] and to SpiderMonkey, Firefox's JavaScript management engine [60]. While tsan11rec was applicable for controlled scheduling of these applications, we found that replay would rapidly desynchronise due to memory layout nondeterminism causing conditionals that rely on the values of pointers to evaluate differently during replay. Tools such as rr can handle these programs reliably by enforcing the same memory layout. This is a trade-off: the non-sparse approach of rr can lead to higher overheads, as demonstrated in §5.2 and §5.3. An alternative to adapting the record-and-replay tool so that it always enforces memory layout determinism would be to adapt the application of interest so that default memory allocation is replaced with a deterministic memory allocator.

6 Related Work

Controlled scheduling A large amount of work has gone into the use of scheduling strategies as a form of state space exploration (e.g. [29, 30, 61, 62, 78, 87]) and on techniques aimed at reducing the size of the state space, such as dynamic partial-order reduction [31, 89]). A particularly notable controlled scheduling tool, in terms of successful practical application, is Microsoft's CHESS [62], which aims to systematically explore all interleavings of a test scenario. Similar to our approach, each visible instruction has an associated custom *wrapper* that intercepts the real instruction, calling into the CHESS scheduler.

Schedule bounding techniques, notably *preemption-* and *delay-bounding* [26, 61], have been shown to be successful in prioritising the order in which thread schedules are explored during controlled concurrency testing. They prioritise exploring schedules that exhibit small numbers of preemptions between threads, in line with empirical evidence that bugs rarely require large numbers of preemptions in order to manifest [56]. Combining such techniques with the tsan11rec algorithm is an appealing idea in principle, but is hindered by the assumption that the program under test takes a fixed input and that the scheduler is the only source of nondeterminism. This assumption allows running the program again and again trying different schedules. In the context of tsan11rec, which can be used to record and replay applications where the environment presents other forms of nondeterminism, the manner in which the program interacts with its environment is captured with respect to a particular thread schedule, and other thread schedules might involve completely different environmental interactions. We believe a more promising approach would be to bring ideas from the *probabilistic concurrency testing* (PCT) algorithm [12] to the tsan11rec setting, to introduce a degree of skewing to our random strategy so that it explores more diverse schedules.

Record and replay Record and replay has been a significant area of research, with many tools being created to facilitate it [2, 9, 19, 24, 32, 36, 39, 40, 43–45, 47, 50, 53, 54, 57, 57, 65–67, 72, 79, 81]. The general premise behind them is similar: identify *order nondeterminism* and *input nondeterminism*, and create techniques to capture them while recording and control them during replay.

Various tools extend the OS in some way or require specific hardware [2, 6, 9, 19, 20, 24, 47, 50, 79, 81]. This has the benefit of giving the tool access to much more of the system, such as memory pages and process information. For example, Scribe [47] will directly modify the system scheduler, instead of coercing it, and achieves slowdowns as low as 1.05×. However, this severely hits the usability of the tool, as it requires the user to deploy a modified OS.

Other tools reside entirely in user space [10, 15, 32, 34, 43–45, 49, 51, 53, 54, 57, 66, 67, 72, 80, 84], and trade performance for usability. This is the category that tsan11rec falls into. Ease of use is particularly important in persuading users to adopt the tool, rr [66] in particular allows the user to record a program by simply passing the binary to rr as a parameter,

	Min	25th	Median	75th	Max	Mean	Overhead
Native	291	369	400	428	502	400.4	1.0×
tsan11	125	193	225	275	431	233.8	1.7×
rnd	128	205	238	281	421	247.1	1.6×
queue	84	188	233	273	435	233.3	1.7×
rnd + rec	113	178	212	253	366	216.8	1.8×
queue + rec	86	161	185	227	348	193.0	2.1×

Table 5. Indicative frames per second (fps) result captured by playing Quakespasm for 90 seconds five times per tool configuration, and capturing the fps reports recorded by the game. Frame-rate results are reported for the minimum, maximum, median and mean case, as well as the 25th and 75th quartiles.

and as such has become the definitive tool for record and replay. We have performed an extensive comparison with rr in §5, and note that while rr outclasses tsan11rec in some applications, and can handle applications that are out of scope for tsan11rec (see §5.5), rr shows significantly higher overhead compared with tsan11rec for a number of applications that rely on a high degree of parallelism for performance. Further, our sparse approach, with suitable workarounds, enables record and replay for graphical applications (the SDL-based games of §5.4) that rr cannot currently handle.

Because it builds on tsan11, which itself uses compiler instrumentation and a modified libcxx, tsan11rec shares similarities with tools that depend on language implementation or library-level support [1, 13, 13, 18, 36, 58]. Notable examples here include R2 [36] and IntelliTrace [58].

Whole system replay aims to record *all* system nondeterminism [14, 19, 22, 24, 25, 27, 53, 77]. Among these, the recent iReplayer tool [53] performs record and replay *in-situ*, avoiding many of the problems (e.g. memory layout issues) that otherwise come from running the record and replay executions under different processes.

Some tools focus on the order-nondeterminism, allowing them to retain their parallelism and thus reducing the overhead of multi-threaded application [25, 42, 59, 63, 70, 86]. Castor [57] will provide each thread with its own buffer for storing information, and serialize them at a later time. tsan11rec also fits into this category, as it will both preserver parallelism of invisible operations and apply a scheduling strategy to resolve this nondeterminism.

An alternative to recording a program's nondeterminism is to *remove* it, making some or all aspects of the program deterministic [5, 8, 16, 17, 21, 55]. For example, Dthreads [55] ensures that memory accesses are deterministic on each execution. Such approaches can have a significant *probe* problem by removing the behaviour necessary for certain bugs to manifest, in return for avoiding the performance overhead associated with handling order-nondeterminism.

Multi-version execution Multi-version (or multi-variant) execution (MVE) is a method for concurrently running multiple processes that are expected to behave in a semantically similar manner [41, 46, 69, 82, 83]. MVE can be used to detect security vulnerabilities in applications: if a variant diverges, this could indicate that an attacker has modified the process in some way [46, 82, 83]. It can also be used for running different analyses on identical processes, that would not work when run together on the same process, such as the clang sanitizers [69]. Most MVE systems hinge on a special monitor thread that controls the generation an maintenance of a number of variants. Keeping the variants in sync with respect to nondeterministic behaviours presents many of the same problems that are associated with record and replay.

7 Conclusion

We have presented tsan11rec, which brings together controlled scheduling, record and replay and dynamic data race detection for the dynamic analysis of C/C++11 applications. Our experimental evaluation demonstrates that the tool is in many cases competitive in terms of performance with rr, a state-of-the-art record and replay tool, in some cases out-performing rr due to tsan11rec's ability to preserve parallelism in applications under test to a high degree. We have also shown that our tool is capable of recording and replaying SDL-based video games, by exploiting our *sparse* approach to avoid recording aspects of game/display communication that are fundamentally hard to control. The flip side of our sparse approach is that by limiting what is recorded, our tool desynchronises on uncontrolled forms of nondeterminism, such as that related to memory layout; by capturing this form of nondeterminism tools such as rr do not suffer from this problem. Two exciting avenues for future work include investigating a spectrum of recording granularities to bridge the gap between our sparse approach and stricter approaches in a configurable manner, and to investigate bug detection using a richer range of scheduling strategies, including schedule bounding [26, 61, 62] and probabilistic concurrency testing [12].

Acknowledgments

This work was supported by a PhD studentship funded by GCHQ, and EPSRC projects EP/N026314/1 and EP/R006865/1.

References

[1] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. 2001. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, April 23-27, 2001*. IEEE Computer Society, 23. https://doi.org/10.1109/IPDPS.2001.924957

[2] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 193–206. https://doi.org/10.1145/1629575.1629594

[3] Apache Software Foundation. 2018. Apache httpd. https://httpd.apache.org/dev/devnotes.html

[4] Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). 2010. *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association. http://www.usenix.org/event/osdi10/tech/full_papers/osdi10_proceedings.pdf

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-Enforced Deterministic Parallelism, See [4], 193–206. http://www.usenix.org/events/osdi10/tech/full_papers/Aviram.pdf

[6] David F. Bacon and Seth Copen Goldstein. 1991. Hardware-assisted Replay of Multiprocessor Programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91)*. ACM, New York, NY, USA, 194–206. https://doi.org/10.1145/122759.122777

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. 55–66.

[8] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: a compiler and runtime system for deterministic multithreaded execution, See [38], 53–64. https://doi.org/10.1145/1736020.1736029

[9] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS, See [4], 177–191. http://www.usenix.org/events/osdi10/tech/full_papers/Bergan.pdf

[10] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 154–163. https://doi.org/10.1145/1134760.1220164

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, Andreas Moshovos, David Tarditi, and Kunle Olukotun (Eds.). ACM, 72–81. https://doi.org/10.1145/1454115.1454128

[12] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs, See [38], 167–178. https://doi.org/10.1145/1736020.1736040

[13] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13, St. Andrews, United Kingdom, October 8-11, 2013*, Shahram Izadi, Aaron J. Quigley, Ivan Poupyrev, and Takeo Igarashi (Eds.). ACM, 473–484. https://doi.org/10.1145/2501988.2502050

[14] Anton Burtsev, David Johnson, Mike Hibler, Eric Eide, and John Regehr. 2016. Abstractions for Practical Virtual Machine Replay. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Atlanta, GA, USA, April 2-3, 2016*, Vishakha Gupta-Cledat, Donald E. Porter, and Vivek Sarkar (Eds.). ACM, 93–106. https://doi.org/10.1145/2892242.2892257

[15] M. E. Chastain. 1999. MEC. (January 1999). https://lwn.net/1999/0121/a/mec.html

[16] Heming Cui, Jirí Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 388–405. https://doi.org/10.1145/2517349.2522735

[17] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient deterministic multithreading through schedule relaxation, See [85], 337–351. https://doi.org/10.1145/2043556.2043588

[18] P. Deva. 2018. Chronon. (2018). http://chrononsystems.com/blog/design-and-architecture-ofthe-chronon-record-0

[19] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 525–540. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/devecsery

[20] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, Mary Lou Soffa and Mary Jane Irwin (Eds.). ACM, 85–96. https://doi.org/10.1145/1508244.1508255

[21] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: a relaxed consistency deterministic computer, See [37], 67–78. https://doi.org/10.1145/1950365.1950376

[22] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5)*. ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/2843859.2843867

[23] Richard Draves and Robbert van Renesse (Eds.). 2008. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association. https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A73

[24] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, David E. Culler and Peter Druschel (Eds.). USENIX Association. http://www.usenix.org/events/osdi02/tech/dunlap.html

[25] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. 2008. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, David Gregg, Vikram S. Adve, and Brian N. Bershad (Eds.). ACM, 121–130. https://doi.org/10.1145/1346256.1346273

[26] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. https://doi.org/10.1145/1926385.1926432

[27] Jakob Engblom, Daniel Aarno, and Bengt Werner. 2010. *Full-System Simulation from Embedded to High-Performance Systems*. Springer US, Boston, MA, 25–45. https://doi.org/10.1007/978-1-4419-6175-4_3

[28] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 121–133. https://doi.org/10.1145/1542476.

1542490

[29] Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 244–254. https://doi.org/10.1145/1806596.1806625

[30] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, Sorin Lerner and Atanas Rountev (Eds.). ACM, 1–8. https://doi.org/10.1145/1806672.1806674

[31] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 110–121. https://doi.org/10.1145/1040305.1040315

[32] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications (Awarded Best Paper!). In *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, Atul Adya and Erich M. Nahum (Eds.). USENIX, 289–300. http://www.usenix.org/events/usenix06/tech/geels.html

[33] Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design* 26, 2 (2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

[34] C. Gottbrath. 2008. Reverse debugging with the TotalView debugger. (May 2008).

[35] David Grove and Steve Blackburn (Eds.). 2015. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* ACM. http://dl.acm.org/citation.cfm?id=2737924

[36] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay, See [23], 193–208. http://www.usenix.org/events/osdi08/tech/full_papers/guo/guo.pdf

[37] Rajiv Gupta and Todd C. Mowry (Eds.). 2011. *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011.* ACM.

[38] James C. Hoe and Vikram S. Adve (Eds.). 2010. *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010.* ACM.

[39] Nima Honarmand and Josep Torrellas. 2014. RelaxReplay: record and replay for relaxed-consistency multiprocessors. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, 223–238. https://doi.org/10.1145/2541940.2541979

[40] Nima Honarmand and Josep Torrellas. 2014. Replay debugging: Leveraging record and replay for program debugging. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014.* IEEE Computer Society, 455–456. https://doi.org/10.1109/ISCA.2014.6853229

[41] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 339–353. https://doi.org/10.1145/2694344.2694390

[42] Derek Hower and Mark D. Hill. 2008. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China.* 265–276. https://doi.org/10.1109/ISCA.2008.26

[43] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 207–216. https://doi.org/10.1145/1882291.1882323

[44] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 141–152. https://doi.org/10.1145/2462156.2462167

[45] Shiyou Huang, Bowen Cai, and Jeff Huang. 2017. Towards Production-Run Heisenbugs Reproduction on Commercial Hardware. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.* 403–415. https://www.usenix.org/conference/atc17/technical-sessions/presentation/huang

[46] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016.* IEEE Computer Society, 431–442. https://doi.org/10.1109/DSN.2016.46

[47] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010*, Vishal Misra, Paul Barford, and Mark S. Squillante (Eds.). ACM, 155–166. https://doi.org/10.1145/1811039.1811057

[48] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[49] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. Chimera: hybrid program analysis for determinism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 463–474. https://doi.org/10.1145/2254064.2254119

[50] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: efficient online multiprocessor replayvia speculation and external determinism, See [38], 77–90. https://doi.org/10.1145/1736020.1736031

[51] Kyu Hyung Lee, Dohyeong Kim, and Xiangyu Zhang. 2014. Infrastructure-Free Logging and Replay of Concurrent Execution on Multiple Cores. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science)*, Richard E. Jones (Ed.), Vol. 8586. Springer, 232–256. https://doi.org/10.1007/978-3-662-44202-9_10

[52] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 443–457. https://doi.org/10.1145/3009837

[53] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: in-situ and identical record-and-replay for multithreaded applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018,*

Philadelphia, PA, USA, June 18-22, 2018, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 344–358. https://doi.org/10.1145/3192366.3192380

[54] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. 2015. Light: replay via tightly bounded recording, See [35], 55–64. https://doi.org/10.1145/2737924.2738001

[55] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading, See [85], 327–336. https://doi.org/10.1145/2043556.2043587

[56] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, Susan J. Eggers and James R. Larus (Eds.). ACM, 329–339. https://doi.org/10.1145/1346281.1346323

[57] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazières, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 693–708. https://doi.org/10.1145/3037697.3037751

[58] Microsoft. 2018. Understanding IntelliTrace part I: What the @#$% is IntelliTrace? (2018). https://blogs.msdn.microsoft.com/zainnab/2013/02/12/understanding-intellitrace-part-i-what-the-is-intellitrace

[59] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Effciently. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China.* 289–300. https://doi.org/10.1109/ISCA.2008.36

[60] Mozilla. 2018. SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[61] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 446–455. https://doi.org/10.1145/1250734.1250785

[62] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs, See [23], 267–280. http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

[63] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*. IEEE Computer Society, 284–295. https://doi.org/10.1109/ISCA.2005.16

[64] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013.* 131–150.

[65] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2016. Lightweight User-Space Record And Replay. *CoRR* abs/1610.02144 (2016). arXiv:1610.02144 http://arxiv.org/abs/1610.02144

[66] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.* 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[67] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli (Eds.). ACM, 2–11. https://doi.org/10.1145/1772954.1772958

[68] pbzip2 development team. 2018. pbzip2. https://launchpad.net/pbzip2

[69] Luís Pina, Anastasios Andronidis, and Cristian Cadar. 2018. *FreeDA*: deploying incompatible stock dynamic analyses in production via multi-version execution. In *Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018*, David R. Kaeli and Miquel Pericàs (Eds.). ACM, 1–10. https://doi.org/10.1145/3203217.3203237

[70] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Emile Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. 2013. QuickRec: prototyping an intel architecture extension for record and replay of multithreaded programs. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, Avi Mendelson (Ed.). ACM, 643–654. https://doi.org/10.1145/2485922.2485977

[71] QuakeSpasm. 2018. QuakeSpasm: An engine for iD software's Quake. http://quakespasm.sourceforge.net/

[72] Yasushi Saito. 2005. Jockey: a user-space library for record-replay debugging. In *Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, Monterey, California, USA, September 19-21, 2005*, Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius (Eds.). ACM, 69–76. https://doi.org/10.1145/1085130.1085139

[73] Torr Samaho. 2018. Zandronum. https://zandronum.com/

[74] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *WBIA*. 62–71.

[75] Simple DirectMedia Layer. 2018. SDL 2.0 library. https://www.libsdl.org/download-2.0.php

[76] SQLite. 2018. SQLite 3.24.0. https://sqlite.org/releaselog/3_24_0.html

[77] Deepa Srinivasan and Xuxian Jiang. 2012. Time-Traveling Forensic Analysis of VM-Based High-Interaction Honeypots. In *Security and Privacy in Communication Networks*, Muttukrishnan Rajarajan, Fred Piper, Haining Wang, and George Kesidis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–226.

[78] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 15–28. https://doi.org/10.1145/2555243.2555260

[79] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: diagnosing production run failures at the user's site. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 131–144. https://doi.org/10.1145/1294261.1294275

[80] Undo. 2018. Reversible debugging tools for C/C++ on Linux & Android. (2018). http://undo-software.com

[81] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: parallelizing sequential logging and replay, See [37], 15–26. https://doi.org/10.1145/1950365.1950370

[82] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming Parallelism in a Multi-Variant Execution Environment. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 270–285. https://doi.org/10.1145/3064176.3064178

[83] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure

and Efficient Application Monitoring and Replication. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 167–179. https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert

[84] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. DrDebug: Deterministic Replay based Cyclic Debugging with Dynamic Slicing. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, David R. Kaeli and Tipp Moseley (Eds.). ACM, 98. https://doi.org/10.1145/2544137.2544152

[85] Ted Wobber and Peter Druschel (Eds.). 2011. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011.* ACM. https://doi.org/10.1145/2043556

[86] Min Xu, Rastislav Bodík, and Mark D. Hill. 2003. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*, Allan Gottlieb and Kai Li (Eds.). IEEE Computer Society, 122–133. https://doi.org/10.1109/ISCA.2003.1206994

[87] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: a coverage-driven testing tool for multithreaded programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 485–502. https://doi.org/10.1145/2384616.2384651

[88] Zandronum. 2015. Zandronum bug tracker. https://zandronum.com/tracker/view.php?id=2380 Bug 0002380.

[89] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models, See [35], 250–259. https://doi.org/10.1145/2737924.2737956