Towards Practical Default-On Multi-Core Record/Replay

Ali José Mashtizadeh
Stanford University
mashti@cs.stanford.edu

Tal Garfinkel
Stanford University
talg@cs.stanford.edu

David Terei
Stanford University
dterei@cs.stanford.edu

David Mazières
Stanford University
http://www.scs.stanford.edu/~dm/

Mendel Rosenblum
Stanford University
mendel@cs.stanford.edu

Abstract

We present Castor, a record/replay system for multi-core applications that provides consistently low and predictable overheads. With Castor, developers can leave record and replay on by default, making it practical to record and reproduce production bugs, or employ fault tolerance to recover from hardware failures.

Castor is inspired by several observations: First, an efficient mechanism for logging non-deterministic events is critical for recording demanding workloads with low overhead. Through careful use of hardware we were able to increase log throughput by 10× or more, e.g., we could record a server handling 10× more requests per second for the same record overhead. Second, most applications can be recorded without modifying source code by using the compiler to instrument language level sources of non-determinism, in conjunction with more familiar techniques like shared library interposition. Third, while Castor cannot deterministically replay all data races, this limitation is generally unimportant in practice, contrary to what prior work has assumed.

Castor currently supports applications written in C, C++, and Go on FreeBSD. We have evaluated Castor on parallel and server workloads, including a commercial implementation of memcached in Go, which runs Castor in production.

CCS Concepts • **Software and its engineering → Operating systems**

Keywords Multi-Core Replay, Replay Debugging, Fault-Tolerance

ASPLOS '17 April 8–12, 2017, Xi'an, China.

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037751

1. Introduction

Record/replay is used in a wide range of open source and commercial applications: *replay debugging* [1–3, 36], can record difficult bugs when they occur and reproduce them offline, *decoupled analysis* [2], can record execution online with low overhead then replay it offline with high overhead dynamic analysis tools, and *hardware fault tolerance* [14, 41], can record execution and replay it on a separate machine to enable real-time failover when hardware failures occur.

Unfortunately, the performance overhead of record/replay systems is frequently prohibitive, particularly in multi-core settings. Often this limits their use to test and development environments and makes them impractical for implementing fault tolerance [5].

Our goal with Castor is to provide record/reply with low enough overheads that developers are willing to leave recording on by default [17, 45], making it practical to record and reproduce difficult production bugs, or to enable fault tolerance in high performance network services.

The widespread use of multi-core architectures poses a key challenge to achieving this goal. Record/replay systems work by recording and replaying non-deterministic program inputs. Multi-core architectures introduce a new source of non-determinism in the form of shared memory interactions, most frequently for synchronization between threads running on separate cores.

Open source [36], commercial [1, 2] and some research systems [20] cope with this challenge by eliminating multi-core execution, and its resultant non-determinism, by deterministically scheduling all threads on a single core. Other research systems constrain and record non-determinism by statically or dynamically adding and recording synchronization [19, 26, 30], or use speculative execution [8, 29, 37, 49] to find a deterministic replay. These approaches have significant runtime overhead and scalability issues that limit their usefulness for production workloads.

Another approach is possible that avoids these overheads. When source code is available, explicit synchronization—e.g., locks, language-level atomics, or (in legacy code) `volatile` variables—can be directly instrumented to capture and record inter-thread non-determinism [18, 22, 23, 39]. Castor builds on this approach with several new observations:

Hardware-optimized logging is essential for keeping record/replay overheads low as core counts and workload intensities increase. Non-deterministic events, such as system calls and synchronization, are recorded to and replayed from a log. Maximizing log throughput is critical, since record overheads increase in proportion to the log throughput consumed, modulo cache effects. Simply put, the more resources we spend on logging, e.g., CPU, memory bandwidth, the less the application has to accomplish real work.

Castor maximizes log throughput by eliminating contention. For example, hardware synchronized time stamp counters are used as a contention-free source of ordering for events. This and other optimizations improve log throughput by a factor of 3×–10× or more—e.g., if recording a web server with a given amount of load imposes a 2% overhead without these optimization, with them, we can record 3×–10× more load (requests per second), for the same 2%.

Minimizing log latency is also critical, as it directly impacts application response times. For example, if a thread is descheduled during a performance-critical logging operation, it can introduce delay, as other application threads that depend on the pending operation are prevented from making forward progress. This can result in user-visible performance spikes.

To prevent this, we make use of a unique property of transactional memory which forces transactions to abort on protection ring crossings. This ensures that performance critical operations will abort if they are descheduled, thus preventing delay. We discuss logging further in §3.

Record/replay can often be enabled transparently, i.e., without any program modification. Castor and similar systems record sources of non-determinism, such as synchronization, by instrumenting source code. Other systems do this manually, e.g., developers replace user level locks with wrapper functions. Manual instrumentation can be tedious and error prone, often does not support all sources of non-determinism, and can introduce additional and unpredictable performance overheads.

To avoids this, Castor uses a custom LLVM compiler pass to automatically record user level non-determinism including atomics, ad hoc synchronization with `volatiles`, compiler intrinsics, and inline assembly; generally this requires no program modifications (see §3.1). It also make use of transactional memory to transparently record inline assembly and efficiently record user level synchronization (see §3.4).

The impact of data races on these systems is often negligible. Shared memory accesses that are not explicitly synchronized, i.e., data races, can introduce non-determinism on replay in Castor and similar systems. In the past, there has been concern that this could undermine the usefulness of replay [29, 30] by interfering with fault tolerance or bug reproducibility. However, this limitation is of negligible importance today for several reasons.

First, modern languages no longer support "benign data races," i.e., races intentionally included for performance. Any benefits they offered in the past are now supported through relaxed-memory-order atomics [11]. Thus, all data races are undefined behavior, i.e., bugs on the same order as dereferencing a NULL pointer [12, 47].

Next, data races appear exceedingly rare relative to other bugs, thus are unlikely to interfere with reproducing other bugs on replay. For example, over a four-year period, there were 165 data races found in the Chromium code base, 131 of which were found by a race detector, compared to 65,861 total bugs, or about 1 in 400. Further, none of these bugs made it into release builds, so at least in this example, data races would likely have no impact on the production use of replay. Other studies of data races suggest that most do not introduce sufficient non-determinism to impact replay [25, 34], and when they do, there are often viable ways of coping. We explore this further in §6.

Castor works with normal debugging, analysis and profiling tools such as `gdb`, `lldb`, `valgrind`, and `pmcstat`. It can also replay modified binaries, enabling the use of ThreadSanitizer and similar tools by recompiling and relinking before replay. Special `asserts` and `printfs` can also be added after the fact. We discuss using Castor, along with its performance on server and parallel workloads in §5.

2. Overview

Castor records program execution by writing the outcome of non-deterministic events to a log so they can be reproduced at replay time, as depicted in Figure 1.

Two types of non-determinism are recorded: *input non-determinism* and *ordering non-determinism*. Input non-determinism occurs when an input changes from one execution to another, such as network data from a `recv` system call, or the random number returned by a `RDRAND` instruction; in these cases, we record the input to the log. Ordering non-determinism occurs when the order operations happen in changes the outcome of an execution—for example, the order that two threads read data from the same socket or acquire a lock. In these cases, we also record a counter that captures the correct ordering (see §3.3).

Castor interposes on non-deterministic events at multiple levels. It relies on a compiler pass to instrument language-level sources of non-determinism such as atomics, compiler intrinsics, and inline assembly (see §3.1). The Castor runtime, which is linked into applications, employs library interposition to instrument OS level events, including system calls and locks (see §2.1).

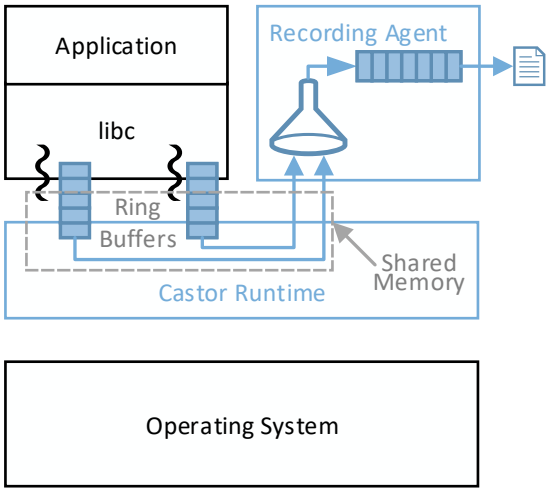


Figure 1: Recording Execution—Using a combination of shared library interposition and compiler instrumentation, Castor records non-deterministic events, logging them to per-thread ring buffers in shared memory. A separate recording agent process drains the events, and sorts them into a partially ordered log it then writes to disk or sends over the network.

Appropriately compiled applications are run with the `record` and `replay` command line tools, which replace key library functions and set environment variables to tell the logging code whether to record or replay.

The `record` command executes the application in record mode and spawns the *recording agent*. Each application thread writes log entries to its own per-thread ring buffer in shared memory (see §3.2). The recording agent drains the per-thread buffers and sorts the log entries into a single *partially ordered log* of events (see §3.3). Periodically the agent writes the log either to disk or over the network, e.g., for fault tolerance (see §3.9).

The `replay` command executes the program in replay mode and spawns the *replay agent*. The agent reads log entries from the disk or network, and places them into per-thread buffers, where they are consumed by interposition code in the application threads.

2.1 Logged Events

Castor interposes on events at different layers of the stack; which layer is chosen affects performance, usability, and implementation complexity (see §4.2).

Some system calls are recorded directly, like `read` and `clock_gettime`, for other events, higher-level library calls are recorded. For example, Castor records pthread mutex operations directly, rather than recording the multiple `_umtx_op` system calls these operations make internally; this results in smaller, easier to understand logs, and lower overhead.

Castor supports two modes of operation: *Self-contained mode* logs the result of most system calls, including those that interact with the file system such as `open` and `read`; replay then uses the previous results rather than re-executing the system calls. Recording file system calls eliminates external dependencies, i.e., no files other than the log are needed for replay. A self-contained log is easy to use and transport, making this mode ideal for replay debugging.

Passthrough mode, by contrast, re-executes many system calls at replay time to recreate kernel state so replay can "go live," to support fault tolerance. In this mode, the initial file system state is captured using ZFS snapshots (see §4.1).

We record language-level events in the compiler including compiler builtins, inline assembly, and shared memory operations (see §3.1). Recording shared memory non-determinism in C11/C++11 and Go is possible because the memory consistency model explicitly defines constructs such as atomics and locks for inter-thread synchronization [12, 27, 47].

Conflicting memory accesses that do not synchronize with one another, i.e., data races, are undefined behavior that we do not record. We discuss the impact of data races in §6.

2.2 Optimizing for Default-On Use

To support default-on operation, we optimize for log throughput, tail latency, and end-to-end latency. Our goal is to keep performance consistent regardless of the number of threads, locks, or cores in use.

Log throughput is a critical metric, as logging overhead is directly proportional to the amount of log throughput consumed, modulo cache effects. For example, in our benchmarks a 10-core Nginx setup could handle 160K requests per second. If each request generates 10 events, that requires 1.6M events per second of log throughput. Castor can processes 30M events per second if it owns a whole core, 1.6M / 30M is 5%. This translates to 5% application overhead, since that is what Castor steals from the application for logging. Doubling throughput to 60M would cut that overhead in half, either halving application overhead, or allowing Castor to cope with twice as much load for the same overhead. Castor employs a variety of techniques to maximize log throughput by eliminating contention (see §3.2 and §3.3).

Tail latency comes from unpredictable performance spikes that affect the long tail of response times. High performance servers like memcached want to avoid these spikes as they can negatively impact user experience [17]. Recording user-level synchronization carelessly can introduce performance spikes when threads are descheduled during critical logging operations. Castor mitigates this by using transactional memory, to keep tail latency low (see §3.4).

End-to-end latency is how long it takes from the time we record an event until we can replay it—keeping this low and predictable is critical to support fault tolerance, and similar applications that replay a replica in parallel [16, 23]. Castor uses incremental log sorting and other techniques (see §3.4 and §3.9) to keep end-to-end latency low. This ensures that the primary is responsive, and that execution on the replica does not lag to far behind the primary, which can impact downtime on failover.

3. Design

Our discussion begins with looking at how to transparently record language-level non-determinism and optimize logging for high throughput and low latency. Later we discuss how Castor supports replay and fault tolerance.

3.1 Transparently Recording Language Level Events

Past systems required developers to manually modify applications to record user-level synchronization [18, 22, 23, 39]. This can be tedious and error prone, and requires developers to deeply understand the inner workings of replay.

In contrast, Castor uses the compiler to transparently record language level non-determinism. This requires no user input except in a few rare cases discussed below.

We implemented this by leveraging Clang/LLVM, the standard compiler on FreeBSD and OS X. Clang/LLVM is extensible, allowing us to dynamically load our compiler pass without modifying the compiler.

Our pass records C11/C++11 atomics, compiler intrinsics (e.g., `__sync_*` builtins), and builtins for non-deterministic instructions, e.g., `RDRAND`, `RDTSC`. We also record/replay operations on `volatile` variables. While ad hoc synchronization is officially unsupported in C/C++, our conservative interpretation of the `volatile` qualifier allows us to precisely replay legacy code with "benign races."

Inline assembly is replayed as a black box by wrapping it in a transaction. Unfortunately, a few instructions such as `CPUID` and `PAUSE` are unsupported inside of transactions, forcing us to issue a warning and fall back to locking until the code is fixed. Fortunately, this is not common.

Transactions are recorded in a similar manner, using nested transactions on x86. On replay, the runtime ensures transactions either execute or are aborted deterministically. The runtime retries spurious aborts that occur for many reasons, including interrupts.

3.2 Efficient Log Structure

Log structure significantly affects log throughput. Many previous record/replay systems log all events to a single FIFO queue, which scales poorly. As core counts increase, contention between threads grows and log throughput plummets, as shown by the curve labeled *Lock* in Figure 2b. Overhead is primarily due to write/write conflicts on the queue's `tail` pointer. This structure can also induce head of line blocking.

Using per-thread FIFO queues with an atomically incremented global counter increases log throughput by 3×, as shown by the curves labeled *Atomic* in Figures 2a and 2b.

Castor can still experience contention on adding and removing entries from the FIFO queues because the reader and writer are on separate cores. Cache coherence traffic dominates log append times as load increases, mostly due to contention on the global counter and also from the cache-line bouncing between the logging and recording threads. We reduce this using familiar techniques, e.g., eliminating false sharing by placing the `head` and `tail` pointers of our queue, and log entries, in separate cache lines.

3.3 Scalable Ordering with Hardware Time Stamps

Many operations must replay in-order to ensure consistency. Examples include any operation that modifies shared state, such as an `open()` system call which modifies the descriptor table, or an atomic compare and exchange. If we are not careful, the mechanism used to provide ordering can become a significant source of overhead.

One common approach is to impose a total order on events using a global counter (shown as curve Atomic in Figures 2a and 2b). Unfortunately, this becomes a central point of contention between threads.

To avoid this, we leverage the synchronized global time stamp counter (TSC) present in modern processors to provide a partial order, approximating Lamport time stamps [27].[1]

Pairing every logged operation with a time stamp of when it completed (see Listing 1) eliminates contention for the counter between cores. Unlike the global counter approach, there is a small chance two cores will read the same time stamp; hence Castor must read the time stamp atomically with the operation in question.

Using the TSC approach, log throughput nearly doubles compared to the global counter approach (Atomic) on our smaller machine, as shown in Figure 2a. On our larger machine, the log throughput for TSC can exceed Atomic by 10× or more, as shown in Figure 2b. Here, the cost of write-write conflicts becomes more pronounced due to higher cache latencies.

Our TSC approach also eliminates contention between agents. Thus, when the record agent becomes a bottleneck, we can add another one. By running two agents, one on each socket, our TSC approach can log 50 million messages/second with 22 cores, as shown in Figure 2b as TSC2X.

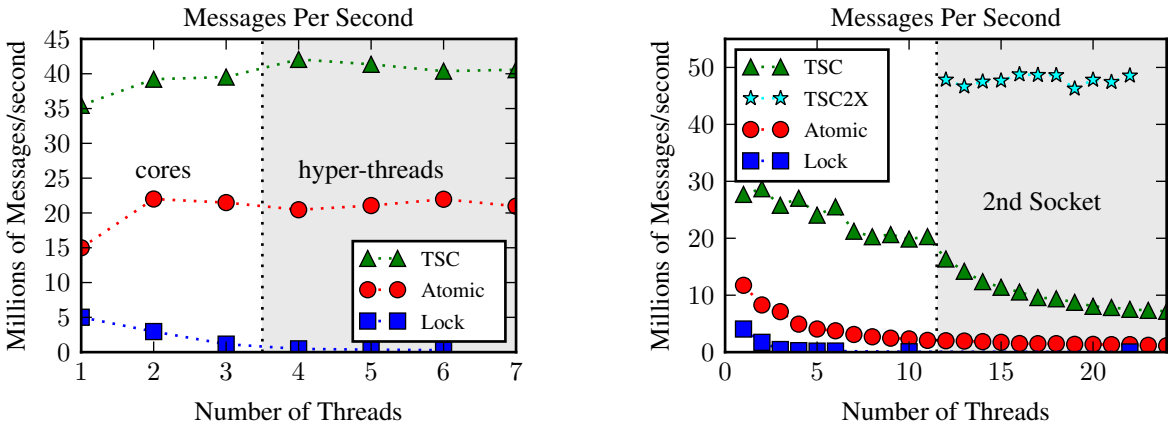
3.4 Transactional Memory for Bounding Latency

Taking an OS context switch in the critical section of logging operations can induce high overheads, either because the operation that was interrupted now takes much longer, or because all the other threads waiting on the lock associated with that critical section are forced to wait.

Transactional memory allows us to avoid this situation since transactions cannot span a protection ring crossing. Running critical sections in a transaction ensures they will abort if preempted. We exploit this in two different scenarios.

Predictably Low Overhead Atomics Developers assume atomics and similar operations will be fast; violating this assumption can introduce strange new performance problems that impact tail latency. Recording atomics without violating this assumption can be challenging.

[1] Synchronized time stamp counters have been available on Intel processors since the Nehalem microarchitecture (2008). Older CPUs and other architectures have OS synchronized per-CPU counters that can be synchronized to within a few cycles of one another. Having the delta between counters bounded low enough allows us to wait inside each event enqueue to ensure we know the relative ordering of events.



(a) 4-Core/8-thread Skylake i7-6700.

(b) Dual Socket 12-Core each Xeon E5-2658 v3.

Figure 2: Log throughput with different designs—*Lock*, is a single shared FIFO. *Atomic* is per-thread FIFOs with a global atomic counter, removing contention between threads. *TSC* replaces the counter with synchronized time stamps, for less contention between cores. *TSC2X* has a per-socket agent that doubles the log throughput. The only scalability problems for the TSC approach comes from read/write conflicts between the application thread and agent, while the counter approach has write/write conflicts for atomic increments.

To see why, remember that recording atomics requires a source of ordering, and that reading the order and doing the operation must be atomic for consistency as discussed in §3.3.

Next, consider what happens if we use per-object locking to provide atomicity for reading the order and doing an atomic increment. If the OS deschedules the thread performing the increment, all other contending threads must wait. Thus, an operation that the developer expected to take a hundred cycles or less can now take millions of cycles.

We have seen this manifest in workloads as visible performance spikes when using per-object locks. When microbenchmarking this, we saw periodic spikes in latency as high as 500 million cycles or approximately 0.22 seconds, on our Skylake machine.

To prevent these spikes, we use transactional memory to atomically read the time stamp counter and perform the operation. Thus, if the OS deschedules the thread, the transaction will abort on the protection ring crossing.

Of course, using transactions also helps by eliminating contention for per-object locks. Unavoidably, we fall back to per-object locks for processors that do not support transactional memory.[2]

Incremental Log Sorting for Low End-to-End Latency To support real-time replay for fault tolerance, we need to minimize the delay between when we record a log entry, and when we replay it. When using fault tolerance (see §3.9), this delay affects the response time on the primary, and how far our replica lags behind the primary, which affects downtime when a failover occurs.

Before we can replay the log, we need to sort it. To minimize delay, we sort log entries incrementally using a sliding window. Once entries are in sorted order they are sent to the replica, and the window advances.

To bound the number of entries in the window to a predictable function of the logging rate, we need to bound the time between when events are appended to the log and when their time stamp was computed.

Unfortunately, if the OS deschedules a thread before it can append an event to the log—i.e., after the operation has been performed and the time stamp has been read—the append can be delayed indefinitely. To prevent this, we put the entire log append within a transaction, again, forcing a complete abort and retry if the thread is interrupted.

To see why this matters in the context of fault tolerance, notice that before our primary can externalize a pending output, e.g., send a packet, we must sort all entries prior to the output and send them to the replica, allowing it to take over if the primary fails. If an append is delayed, we must stall the pending output and grow our window size, inducing user visible latency proportional to the size of the window.

To see how this could impact latency, we measured how much delay is induced if our thread is descheduled, using either the atomic counter+lock or TSC+TSX approach. In the common case, this was on the order of thousands of cycles, however, worst case behavior was 300 million cycles or 134 ms of delay for the counter approach—greater than

[2] Transactional memory was introduced in 2014 by Intel, and is available in all Intel processors from laptop to server class chips as of 2016. It has been available in the POWER8 and SPARC T5 architectures prior Intel's release.

the RTT across the United States. Meanwhile, the worst case behavior when using transactional memory was 500 thousand cycles or 223 μs of delay.

Finally, while our TSX approach works for all user level operations, we cannot use it for recording OS level operations since they cross protection boundaries. Future kernel support could allow us to mitigate this.

3.5 Many Threads with Low Overhead

As the number of running threads increases, the overhead of polling logs for pending entries to drain by the record agent can become significant.

This overhead was particularly pronounced in our Go implementation (see §4.2). In Go, the use of hundreds or even thousand of users level threads, called *goroutines* is common. As our logs are per-goroutine this quickly became a problem. At even 10 goroutines the overhead of polling logs became noticeable. At 100 goroutines throughput could drop from 9M to just 4.2M events. At 1000 goroutines throughput dropped to 0.8M events, less than one-tenth of peak throughput.

To address this, we added hooks to the Go scheduler so the agent can track which goroutines are running and avoid scanning logs with no pending entries. When recording, the agent drains a goroutine's log when it is descheduled. On replay, the current goroutine will yield if its waiting too long for the next event to occur since another goroutine may need to consume earlier log entries to make forward progress.

Using this approach, Castor can handle up to 10K goroutines with a nearly constant 9M events per second throughput.

3.6 Record Buffer Sizing

Castor provides record modes optimized for different use cases:

Fault tolerance: Requires low latency. We use short ring buffers (4 KiB per thread), and dedicate a core to the recording agent, which constantly polls. This can also improve throughput as all data can fit in the L1/L2 cache. Log sorting is done online.

Debugging and analysis: We use 4 MiB/thread buffers, and allow the recording agent to sleep when there are few events. Thus, it uses only a fraction of a CPU core. Log sorting is done offline.

Crash recording: We use 128 MiB/thread for recording crashes in long running applications similar to commercial tools [1, 2]. In this case there is no agent and no draining of logs. When logs fills up, Castor dumps a checkpoint to disk, flushes the log to disk asynchronously, and allocates a fresh log for the next interval. This supports recording a finite interval prior to a crash. Older logs and checkpoints can be discarded to bound storage consumption.

Buffer sizing can impact Castors log throughput and ability to handle workload spikes. Larger per-thread buffers handle workload spikes better while increasing cache pressure. Smaller buffers tend to perform better as all data fits in the cache reducing Castor's cache footprint, but increases sensitivity to workload spikes and descheduling/interrupts of the agent process. Adaptively tuning this based on workload could be useful.

3.7 Replay Modes

As discussed in §3.3 certain events must replay in-order to ensure consistency. Castor supports two approaches, each with its own trade-offs. In both modes, the replay agent reads log entries from either the disk or network and places them into per-thread queues.

Totally ordered replay replays events in the same total order they were recorded in. To start, Castor takes log entries, which are already sorted (see §3.4) by *event id*, and replaces the event id with a monotonic count. Next, Castor sets up a global monotonic counter to control replay order, an event can only replay when the global counter matches its event id.

On replay, when an application thread reaches an instrumented event, such as a lock acquire, it waits for the agent to enqueue a log entry for that event. Once it has the entry, it waits for the event id of the entry to match the global counter. When this happens, the thread replays the lock acquire, increments the global counter, and execution continues.

This approach has a few benefits: it is simple, performs reasonably at low core counts, and requires almost no processing by the agent. However, while threads run in parallel, they are still forced to synchronize on a global counter. This limits parallelism, since events without dependencies could be running in parallel.

Partially ordered replay removes global synchronization. Instead, the replay agent dynamically infers which events have dependencies and constrains them from executing concurrently.

To do this, the agent tracks dependencies in the log using a bitmap where each bit corresponds to an event's object ID, e.g., the address of a lock. During replay, before the agent enqueues a log entry for an operation on an object, it checks to see if a bit has already been set for the object; if it has, the agent waits until the bit is cleared; if it has not, it sets the bit. Once an application thread has processed an operation on an object, it clears the bit for that object.

Partially ordered replay has two advantages. First, it eliminates the write/write contention on a global counter. Second, it relaxes the execution schedule of operations to maximize parallelism. This can improve replay performance and also reduces sensitivity to scheduler and interrupt delivery differences between record and replay.

3.8 Divergence Detection

Divergence occurs when replayed execution differs sufficiently from recorded execution that continued replay is either infeasible or inadvisable. Divergence may be due to implementation bugs in record/replay, data races, hardware nondeterminism, or replaying on a different microarchitecture with behavioral differences.

On replay, Castor compares the log to program behavior to detected three types of divergence: event divergence, deadlock divergence, and argument divergence.

Event divergence is a disagreement between a thread and its own log about type of event is next. For example, if a thread makes a `gettimeofday` system call, and Castor goes to the log to fetch the log entry for this event and instead finds an entry for a `read` system call, something has gone wrong. We saw these frequently while implementing Castor when we failed to record a source of non-determinism that changed control flow on replay.

Deadlock divergence is a disagreement regarding the ordering of the log between threads, i.e., when synchronization that is not being recorded/replayed (races or uninstrumented code) deadlocks with the log-enforced ordering. For example, suppose thread *A* is blocked waiting for thread *B*, thread *B* tries to acquire a lock, and the log shows that thread *A* should get the lock next. Something went terribly wrong as replay will deadlock. We encountered these during development when we failed to properly instrument synchronization.

Argument divergence is a disagreement regarding the values associated with an event and the log, e.g., the arguments to a system call. For example, for the `write` system call we log the number of bytes written and a hash of the value being written. For efficiency, we currently use xxHash [6], though hardware support for SHA1/256 will be available in future processors [50]. This check is critical for fault tolerance as the program output on record and replay should be indistinguishable.

All of these checks proved very helpful while developing Castor. The first two checks caught most bugs, including two intentional races in the Go runtime.

3.9 Fault Tolerance

Fault tolerance supports continuous availability in the face of hardware failures, and involves two servers; the *primary* which is being recorded, and the *replica*, which streams the log in real-time from the primary and replays it to create a hot standby.

Failover occurs when the hardware running the primary fails and the replica takes over. Failover in Castor is transparent, i.e., the primary and replica are indistinguishable from the perspective of an external observer, since divergence detection (see §3.8) ensures that output from the primary and backup are identical up to the point of failover.

This is a potentially subtle point, the replica and primary can differ in their internal state prior to failover, but this irrelevant. Like two runs of an NFA that share a common prefix, before failover they are identical from an external observer's perspective. After failover nothing changes, from the observer's perspective it is as if only the replica ever existed.

When the replica detects a divergence, it immediately pauses and resynchronizes. To resynchronize, the replica notifies the primary, which sends a checkpoint. The Castor runtime replays all log entries prior to the checkpoint to ensure kernel state is consistent, then loads the checkpoint and resumes execution.

Before Castor allows the primary to externalize an output event, i.e., send a packet, it ensures that the replica has consumed all log entries prior to that event. This guarantees that the replica received the log entries and that a divergence did not occur. Thus, if the primary dies, the replica has everything it needs to take over transparently.

To implement this, when a thread on the primary attempts to send a packet, Castor delays the packet and tags it with a time stamp. As the replica consumes the log, it asynchronously acknowledges the latest time stamp it has consumed. When the primary receives an acknowledgment, it releases all packets with a lower time stamp.

4. Implementation

The Castor runtime is under 6 KLOC. The record, replay, and fault tolerance agents comprise just over 1 KLOC. The LLVM compiler pass is just over 1 KLOC. We changed 5 lines in `libc`. Go support requires 2 KLOC, 500 lines of which are shared with the C version.

4.1 FreeBSD

Capturing OS-level non-determinism in libc and libthr (pthread) is done through library interposition. FreeBSD provides weak symbols for much of libc, meaning we can override these symbols at link time by providing our own replacement with the same name, e.g., `read`. Other functions can be interposed on dynamically by changing a table of function pointers built into libc for this purpose. Finally, we modified 5 lines of code in libc and libthr to expose symbols we needed to hook.

In self-contained mode (see §2.1), to record Castor executes system calls and writes their return value and side effects to the log. On replay, most system calls are elided, and their values and side effects are returned from the log, similar to a variety of other systems [18, 20, 39, 40, 44]. We interpose on roughly 107 types of events in our current prototype, including system calls, library calls, user level locks, etc.

The Castor runtime maintains a table of file descriptor metadata including their type and per-descriptor locks. To keep this consistent, we interpose on any call that can modify the descriptor space such as `open`, `dup/dup2`, and `fcntl`. Castor uses this table for a variety of purposes. Per-resource locks are used to ensure proper ordering on aliased resources when replaying. Knowing the type of descriptors lets us avoid recording reads and writes on socketpairs or pipes between recorded processes.

For fault tolerance, we replay in pass-through mode, which re-executes most system calls to recreate kernel state on the replica. Replaying system calls makes it possible for the replayed process to *go live* when the original recording host fails. In passthrough mode, we leverage ZFS file system

snapshots to synchronize the initial file system state of the primary and secondary. As a side effect, this improves performance by eliminating the need to log file system reads.

4.2 Go Language/Runtime

Supporting Go required integrating with the Go runtime. Languages such as Objective C and Rust that use LLVM and libc work out of the box with Castor. However, Go developers use their own compiler, linker, system libraries, etc. thus, additional support was required. Our implementation is built on Go 1.4.2.

To support Go, we initially tried recording the lower layers of the Go runtime with one log per OS thread, similar to our C/C++ implementation. However, after some experience, we switched to a different approach: logging higher-level APIs built on the core runtime and keeping one log per Go thread (called *goroutines*). This allowed us to simplify our implementation considerably by ignoring lower-level non-determinism in the runtime, such as the scheduler, network stack, allocator, and garbage collector. It also resulted in logs that are smaller and easier to understand, as they correspond more closely to language-level semantics. Supporting large numbers of goroutines required integrating more closely with the Go scheduler, as discussed in §3.5.

Logging at a higher layer exposed some internal non-determinism. In particular, when finalizers execute can vary from run to run, depending on when garbage collection happens. Finalizers are processed by a goroutine associated with the garbage collector. We modified this goroutine to log when each individual finalizer is run relative to other events in the system. On replay the routine executes the finalizers in the same order they ran during the recording. This sometimes requires the goroutine to force a garbage collection to run if the object being finalized is not ready.

Asynchronous signal handling is simplified by logging at a higher layer. Instead of delivering signals at the OS level, we can leverage existing machinery in Go. In particular, asynchronous signals, e.g., `SIGINT`, are delivered by a goroutine in the language runtime that listen on a channel for signal messages sent by lower layers of the runtime. To record signals, we modified this routine to log and replay signals.

4.3 Event Logging

Castor's log is a FIFO queue. Each entry is the size of an L1/L2 cache line (64 B). The `head` and `tail` live on their own cache lines. The recording agent is a separate process that polls each per-thread queue and drains them into a buffer that gets flushed to disk or over the network. As discussed in §3.2, Castor uses transactional memory and time stamp counters for contention-free logging. These are available as TSX and `RDTSC` on x86-64 processors. When transactional memory is not supported or fails, we fall back to user-level sharded (per-object) locks.

Transactional Memory To log an event, the runtime first checks that the next log entry is free by examining `head` and `tail`. It then executes a transaction in which it atomically: checks the sharded lock, executes the logged operation, reads the time stamp counter, writes the log entry, and bumps the `tail` pointer.

Listing 1 shows a simplified version of the code generated for recording an atomic exchange (the `XCHG` instruction). The time stamp is read (`RDTSC`) in the same transaction as the atomic exchange, which provides a causal ordering of the log entry relative to another changes on the same atomic variable.

```
# %rcx = Pointer to free log entry
# %rbx = Per-thread log tail
# %r8 = Pointer to sharded lock
    movq ENTRYTYPE_ATOMIC_XCHG,(%rcx)
    xbegin record_slowpath
    cmpl $0,(%r8) # Read sharded lock
    jne lock_notheld
    xabort # Abort if others are in slow path
lock_notheld:
    xchg %r9,(%r10) # Atomic exchange
    rdtsc
    mov %eax,8(%rcx)
    mov %edx,12(%rcx)
    incq (%rbx)
    xend
```

Listing 1: A simplified version of the code executed for the record path.

Performance We measured the transaction abort rate and throughput during the development process using a tight logging loop. We found most aborts were due to interrupts. However, use of a time stamp counter is crucial; using a shared in-memory counter would cause most transactions to abort due to conflicts on the shared counter. We noticed a modest benefit from increasing log entries to the L3 cache line size (128 B) vs. the L1/L2 size (64 B), but decided against doing so to avoid the impact on log size. We saw a slight uptick in spurious aborts with the use of hyper-threading, which we attribute to hyper-threads sharing an L1/L2 cache. We speculate that cache lines lack a bit to record whether they belong to one or both hyper-threads' read sets, though in practice such aborts have not been an issue.

Sharded Locks and Nested Events Sometimes Castor cannot use transactional memory, either because a processor does not support it or because it is incompatible with a particular operation. System calls always trigger aborts, as well as certain instructions including `CPUID` and `PAUSE`.

When Castor cannot use transactional memory, the atomicity of time stamps and logged events is enforced through user level sharded locks. Locks are sharded by unique resource identifier such as memory address or file descriptor.

Recording higher-level interfaces sometimes requires the use of nested events, since we still want to record and replay ordering at lower layers properly. For example, when a mutex is statically initialized with `PTHREAD_MUTEX_INITIALIZER`, the first call to `pthread_mutex_lock` dynamically allocates memory. In this case `pthread_mutex_lock`, which is logged, calls into `malloc`, which also uses locks.

To record something like this, Castor acquires a sharded lock and logs a *start event* recording when it acquired the outer lock. It then performs the target operation, including generating log entries for nested events. Finally, it generates an *end event* recording the target's non-deterministic results (if any), before releasing the lock. To support nested events on the same resource, we built user-level recursive mutexes using C11 atomics.

5. Evaluation

Our experience using Castor is discussed in §5.1. Performance on heavy server workloads is presented in §5.2, and multi-core scalability via parallel benchmarks from SPLASH-2 and PARSEC is presented in §5.3. We conclude with microbenchmarks, and some general observations on performance with Castor's.

Our measurements were taken using two machines running FreeBSD 10.3. Our first machine has a 3.4 GHz Intel i7-6700 Skylake processor with 4 cores/8 hyper-threads, 8 GiBs of memory, and transactional memory (TSX) support. Our second machine has a dual-socket Intel Xeon E5-2658 v3 Haswell processor with 12 cores/24 hyper-threads per socket and 128 GiBs of memory, but no TSX support. All measurements were done in debugging and analysis mode (see §3.6) using 4 MiB/thread buffers with a single recording agent.

5.1 Using Castor

We built all workloads evaluated in this paper by recompiling and relinking unmodified source code with Castor, supporting our thesis that a custom compiler pass can avoid the need for manual instrumentation.

We used Castor with unmodified debugging tools including `gdb` and `lldb` for replay debugging, as well as `valgrind`. Small runtime detours that insert or consume log entries are the only visible difference while debugging.

With performance tools like `pmcstat` we see the time spent in the record-replay infrastructure and the impact of eliding system calls as we made no effort to hide these details from the tool.

Castor supports replaying modified binaries, allowing us to perform many tasks *retroactively*, i.e., by recompiling and relinking before replay to change libraries or add instrumentation, or even change application source code. So long as changes do not impact the log, this works just fine.

To use dynamic analysis tools like ThreadSanitizer and AddressSanitizer, we recompile and relink our application with the appropriate flags, then replay as usual. This works because the runtimes for these checkers only make raw system calls so they do not interact with Castor's event log.

We also created our own versions of `assert` and `printf` that bypass logging. We can add these to our programs after the fact to debug at replay time.

To deploy fault tolerance, we provide a special command `cft` that launches our program in passthrough mode, either recording or replaying. When using `cft`, the record agent transmits the log over the network to the replay agent. We use CARP (Common Address Redundancy Protocol) support in FreeBSD to fail over an IP address from one server to another on hardware failure. For UDP based applications, this works seamlessly.

5.2 Network Server Workloads

We evaluated a variety of network server workloads with Castor. As with other workloads in our evaluation, replay times were substantially lower than record times, as we discuss in §5.5.

Nginx We benchmarked Nginx [35] on our Haswell machine, using the *wrk* HTTP load generator configured with 12 threads and 100 connections for a duration of 30 seconds. Nginx scaled well with approximately 1% overhead until 10 cores. At 10 cores we were attempting to handle 160K requests per second and approximately 2 million log messages per second. At this point we began to run up against log throughput as a limiting factor. As predicted, overhead jumped to around 9%, roughly what we expect to see at 10 cores, where our peak log throughput is 20M events/per second as shown in Figure 2b.

Memcached We measured record overhead of memcached on our Skylake machine, using mutilate [31] to generate a maximum load of 119K requests per second. On average we saw 29 events per request. Multithreaded memcached [48] with a single worker thread uses a listener thread and 5 additional helper threads, for 7 threads total on our 4 core/8 hyper-thread machine. Overhead again increased proportional to log bandwidth, with a roughly 7.5% drop in throughput at 119K requests per second.

Lighttpd We measured record overhead for lighttpd [4] on our Skylake machine with load generated by `ab` [46], which we tuned to generate the maximum requests lighttpd could handle. Lighttpd is a multiprocess webserver and as such record/replay has no effect on scalability. Thus, performance is mainly dependent on the logging rate. Turning on recording incurred a roughly 3% reduction in throughput.

PostgreSQL We measured record overhead for PostgreSQL [38] on our Skylake machine. Using `pgbench` with a read/write intensive workload for 10 minutes with 4 client threads and two worker threads, we measured a roughly 1% drop in throughput.

LevelDB We measured record overhead for LevelDB [21] on our Skylake machine. Using `db_bench` with default settings, we saw a 2.6% slowdown from recording, while CPU consumption increased 9%.

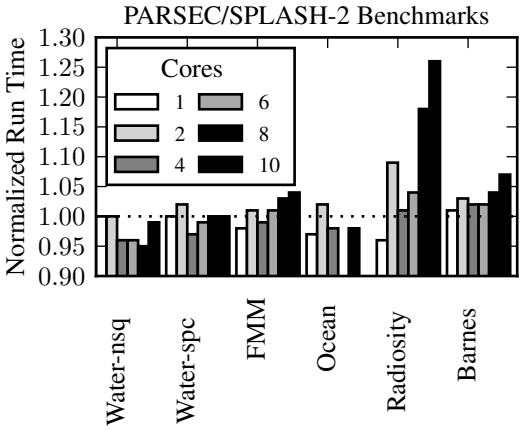


Figure 3: Recording PARSEC/SPLASH-2 on a 12-core Xeon Haswell with a single recording agent—execution times are normalized against number of threads with standard deviation below 2%. Only Barnes, Radiosity, and FMM exhibit measurable overheads. Many benchmarks executed faster with Castor due to changes in memory layout. For Radiosity, a single recording agent is insufficient to keep up with workload, illustrating the need for multiple recording agents to cope with heavier workloads.

Go Caddy (HTTP) We evaluated Caddy [32], a popular HTTP server written in Go, on our Haswell machine. Our Go implementation is currently unoptimized and is impacted by limitations in Go's C compiler. Record overhead for Caddy was 15% or less, scaling from one to twelve cores. At one core, we saw 5,255 req/s when recording, vs, 6,175 req/s normally, a 15% overhead. At twelve cores, we saw 36,658 req/s when recording, vs, 39,156 req/s normally, a 6.6% overhead. Replay was 1.9× faster than record.

Go Memcache We evaluated a commercial memcached implementation written in Go [33] on our Haswell machine. Record overhead was 5.7% or lower while scaling from one to twelve cores. This memcached implementation did not scale well due to its use of coarse-grained locking. We saw 72,708 req/s when recording, vs. 77,109 req/s normally, a 5.7% overhead. At twelve cores, we saw 112,270 req/s when recording, vs, 117,213 req/s normally, a 4.2% overhead. Replay was 1.3× faster than record.

Running our implementation in a production setting serving live traffic we observed an additional 5% of CPU was required to run record and approximately 220 MB of uncompressed log was generated per hour.

5.3 Multi-Core Scalability

We evaluated Castor with benchmarks from the PARSEC and SPLASH-2 multiprocessing benchmark suites [9, 51] on our Haswell machine. PARSEC/SPLASH-2 are commonly used to evaluate multi-core record/replay systems [19, 29, 30, 49], as their heavy use of synchronization stresses scalability and their ubiquity facilitates apples-to-apples comparisons among systems.

The PARSEC distribution includes SPLASH-2. We excluded the SPLASH-2 kernels from our graph, along with several other benchmarks that exhibited no noticeable overhead.

We dedicated a whole core to the recording agent and thus excluded the 12 core numbers. Logs were written to the null device to eliminate storage performance interactions. SPLASH-2 uses pthread mutexes. On replay, Castor enforces lock ordering that almost eliminates the cases where the pthread mutex code must call into the kernel.

In Figure 3 we see the normalized execution time for these benchmarks; a value of 1.00 is ideal. The benchmarks perform identical work regardless of the number of cores. Execution time is normalized against the number of benchmark threads, standard deviations were generally below 2%.

Water-nsq, *Water-spc*, and *Ocean* had nearly zero overhead. In these and other benchmarks, we found that cache effects due to changes in memory layout measurably changed performance, sometimes leading to performance gains, which is why many benchmarks are actually faster with recording enabled. We start seeing overhead in FMM and Barnes at 8 cores, but still below 5%.

Only Radiosity showed significant overhead. This was due to two factors: First, this benchmark was particularly sensitive to added cache pressure. Second, during certain phases of the benchmark large numbers of events are generated on all cores simultaneously, so log bandwidth becomes a limiting factor at 8 cores and beyond. In particular, the single logging agent was unable to keep up with draining the log. Adding an additional recording agent is required to continue to scale.

5.4 PARSEC

We ran *pbzip2* on 6 cores with approximately 4% overhead. We ran several benchmarks from PARSEC versions 2 and saw results similar to our fastest SPLASH-2 benchmarks, i.e., zero overhead and small variations due to cache effects.

5.5 Replay Performance

Replay times were consistently either as fast as or faster than recording, and often faster than unrecorded execution. There are a number of reasons for this.

First, the replay agent is able to append to queues in $O(1)$ time, as it knows the target thread given a log entry. In contrast, the recording agent drains queues in $O(n)$ time where n is the number of running threads, as it has to read the head of every thread until it finds the next log entry.

Second, replay generally can read ahead in the global log to feed the individual per-thread queues. This means we can potentially fill all replay queues of all threads, so if an interrupt occurs on the processor with the replay agent, there may be more replay buffer available. On recording, interrupts delivered to the recording agent are more likely to lead to small delays in execution.

Third, replay logs obviate the need to execute recorded operations, thus we can avoid re-executing certain system calls, reducing IO latency in some cases. For example, for fault-tolerance we replicate network IO to the destination.

5.6 Microbenchmarks

Per-Event Record Overhead Record overheads were quite predictable. We measured roughly 80–200 cycles for each non-deterministic event logged. Overhead scaled roughly linearly as a function of the number of events logged. Cache effects were the only other significant source of variation.

Go Our Go implementation with a single core can log 7.3M events per second while 12 goroutines on our 12-core box tops out at 9.0M events per second. These lower numbers are due to limitations of the C compiler used by Go, a lack of transactional memory support in our Go implementation, and our currently unoptimized integration with the Go scheduler that requires iterating and dereferencing three structures per dequeue. On average, it costs 350 cycles to enqueue an event, with 130 of those cycles spent inside the critical section when logging locks.

Log Size For the SPLASH-2 benchmarks log sizes ranged from 14 kiB to 43 MiB and compression ratios using `xz` ranged from 23× to 133×. Since our logs tend to have so much padding it is not surprising that they are easily compressible.

5.7 Observations

Predictable Performance: Our experience with Castor has been that workloads behave quite predictably. At lower event rates cache effects are noticeable, but record overheads remain consistently low. Overheads were often under a percent for both server and parallel workloads as shown in §5.2 and §5.3. As event rates increase, log throughput becomes the limiting factor, and overheads are a predictable function of log throughput.

Queue Size vs. Cache Pressure: Some of our server workloads perform better with larger queue sizes. In contrast, SPLASH-2 benchmarks often performed worse due to increased cache pressure. We also saw a related effect, where the impact on memory layout of adding record/replay actually improved performance over normal, as seen in Figure 5.3.

Multiple Recording Agents: Record log throughput becomes a bottleneck with sufficient load. Often the limiting factor is ability of a single recording agents ability to rapidly drain the log. We saw this both with Radiosity in SPLASH-2, and in our server workloads, e.g., as we scaled NGINX to 10 cores. Thus, to scale beyond a certain point, dedicating multiple cores to logging is necessary.

6. Data Races and their Impact on Replay

Modern languages such as C11, C++11, and Go, provide explicit mechanisms, such as the `_Atomic` qualifier, to notify the compiler of variables that are shared between threads or otherwise accessed concurrently. If a variable that does not employ these mechanisms is operated on concurrently with no intervening inter-thread synchronization and at least one operation is a write, it is a data race.

Data races are a source of non-determinism, as their effects may depend on the relative order of instructions executed across different CPU cores. Since cross-core instruction ordering may vary from one execution to another, and since additionally the order does not depend on any inter-thread synchronization (which Castor would log), Castor does not have enough information to replay data races identically. Thus, we can only guarantee deterministic replay up to the first data race. Castor always lets us detect that first data race by retroactively applying a *happens-before* race detector [7].

Past work has insisted on precisely reproducing all data races, on the grounds that the non-determinism they introduce can impact reproducing bugs or fault tolerance [29, 30]. Our perspective is that the practical impact of data races on record/replay in Castor is largely negligible, for a variety of reasons.

In the past, "benign data races," data races that were intentionally included for optimization reasons, were a recognized practice. Benign data races have always been dangerous and non-portable practice as their semantics were not well defined, thus, the compiler could always interpret them in a manner that would yield bizarre bugs [10].

At present, data races are explicitly prohibited and yield undefined behavior in languages such as C11, C++11, and Go. Moreover, data races no longer offer performance benefits on modern architectures [11]. Programs that need scheduler-induced shared-memory non-determinism can achieve it with relaxed-memory-order atomics, which we record and replay. Thus, from a language perspective, it makes little sense to pay a high price for precisely replaying undefined behavior. That said, legacy code often uses the `volatile` qualifier for variables subject to intentional data races. In this case, Castor does precisely reproduce benign data races, as it records and replays `volatile` variables.

We still want to consider unintentional data races, since programmers make mistakes, and we want to understand the impact. In the following sections we consider how frequently these bugs commonly occur, and, when they do, how they affect replay.

6.1 Data Race Frequency

We found limited data in the existing literature on how frequently data races occur and how effective existing tools are at early detection.

As a first step to understanding these issues, we looked at the Chromium web browser bug database. We chose

Chromium for a variety of reasons: its large size, prodigious use of multi-threading, and active development, all lend themselves to the presence of data races; its use of ThreadSanitizer [42], a state of the art race detector, let us examine the impact of dynamic race detection on catching these bugs; and its large well annotated bug database simplified research.

We looked at the period from January 1, 2012 – January 1, 2016. During this time the code base grew from roughly 6M LOC to 13M LOC. We considered only bugs that were closed and fixed, to eliminate noise from false positives. While this is a limited data set, it suggests several conclusions.

First, data race bugs were quite rare relative to other bugs. Out of 65,861 total bugs fixed during this period, 165 were data races, or about 1 in 400. Next, current development practices and tools seem effective at catching and eliminating data races before they reach production. Out of 165 data race bugs, 131 were caught by ThreadSanitizer.

Notably, none of bugs were in release builds (development, beta or stable). Thus, it seems likely that data races would have little effect on record/replay of production code in similar modern development settings. Moreover, data races should have almost no impact on Castor's ability to capture reproducible bug reports in production code in the field.

It seems that software bugs leading to downtime in general would dwarf any impact of data races on fault tolerance. In Chrome for example, there were 211 bugs in stable release builds that caused crashes, and 10 that caused the application to hang.

6.2 When do data races impact replay?

Data race bugs generally manifest non-deterministically. If a race interferes with replay regularly, it will quickly be detected. If it manifests infrequently, it is unlikely to impact record/replay in its task, e.g., infrequent divergences in fault tolerance are easily handed by resynchronizing (see §3.9).

When races do manifest as non-determinism, they can impact replay in three ways: *log divergence* (see §3.8), where execution changes sufficiently that the log is not adequate to complete replay, *value divergence*, where intermediate values of an execution change, and *output divergence*, where the user visible result of execution changes, which in Castor are treated as log divergences (see §3.8).

Log divergences caused by data races seem infrequent. We have only encountered a couple, caused by deliberate (benign) data races in the Go runtime that were easily found and fixed. Others building replay systems reported similar experiences, e.g., a couple of benign data races [23], or races that had already been caught [18].

Two studies classifying data races suggest most races would not cause log divergence. One, studying 68 data races in Windows Vista and Internet Explorer, found roughly half had no measurable impact on execution [34], i.e., no change in control flow or register live outs. Of the remaining half, most would not impact replay, while the remainder could possibly lead to log divergence. Similar results were reported in a separate study [25].

The impact of value divergences depends on use case. For example, for decoupled analysis, such as running a memory error detector or race detector, a value divergence is still a potential program path, and thus little is lost.

For replay debugging, in the worst case, we would not be able to reproduce a bug that occurred after the data race. However, we have two options: we can fix the race and redeploy, or employ some strategy to reproduce the bug from the existing log—this could be as simple as re-running a few times with partially ordered replay (see §3.7), or as complex as using speculative execution [8, 29, 37, 49] to find an interleaving that reproduces our bug [18].

For fault tolerance, value divergences are irrelevant (explained in §3.9), since we only care about what an external observer can see. Output divergences are detected as discussed in §3.8, and dealt with like any other form of log divergence, by resynchronizing with the primary. This approach is similar to what VMware's FT [41] and other commercial solutions do to cope with hardware non-determinism.

Resynchronizing introduces a delay and a small downtime window, so having it happen frequently is not desirable. For reasons discussed above, having this occur often due data races would be very unusual. Further, for a loss of availability to result, the downtime window would need to overlap with the downtime window of the protected hardware. Put another way, replication failure and hardware failure would need to coincide.

Finally, should one wish to fix these bugs when they occur in the field, others have described how data races can be detected and fixed automatically [18].

7. Related Work

Shared memory non-determinism is a key challenge to multi-core record/replay. Systems have coped in two ways. Some reproduce all data races precisely. Others like Castor only record and replay explicit synchronization.

Reproducing All Data Races Systems that reproduce all data races tend to scale poorly as shared memory activity increases. Often this is due to the increased contention these systems induce by adding synchronization.

Multi-core record/replay systems that work at the OS, VMM, or binary translator level need to reproduce all data races, since synchronization information is often lost in the compilation process. For example, on x86 a relaxed-memory-order atomic store is indistinguishable from any other word aligned store at the ISA level.

VMM[19] and OS level [26] replay systems have implemented CREW [28] (concurrent read, exclusive write), a cache coherency protocol at the page granularity that provides determinism by leveraging the MMU to add extra synchronization. In these systems, overheads from page faults

and inter-processor interrupts (IPIs) are high, compounded by false sharing from the large page size.

For example, SMP-Revirt [19] saw slowdowns ranging from 2×–4× for 2 CPUs, and up to 8× for 4 CPUs on SPLASH-2. For applications with very little sharing, these costs can be avoided or amortized [26], and CREW can be more performant in emulated hardware [15]. However, high overheads and poor scalability make CREW generally unsuitable for production workloads.

Several approaches work by simultaneously replaying a copy of the main process while recording, to find a recording that is deterministic [52]. ReSpec detects divergences that occur when logging is insufficient, and is able to rollback the execution and/or restart the replayed process. DoublePlay goes further, allowing offline replay by finding an execution schedule for the threads that can replay correctly on a single core. DoublePlay has average overheads of 15% and 28% for two and four threads, while ReSpec has overheads of 18% and 55%. Both require parallel execution, and thus twice the CPU resources. Worst case SPLASH-2 slowdowns for both exceed 2× for 4 threads.

Other systems attempt to reproduce all data races out of the desire to eliminate non-determinism on replay for debugging or fault tolerance. Chimera ensures synchronization by analyzing program source code with a static race detector and instrumenting it with *weak locks*. Added contention introduces significant slowdowns on multi-core. On SPLASH-2, Chimera saw overhead ranging from 1.6× for two cores to over 3× for 8 cores. Similar systems exhibited substantially larger overheads [24] or only considered small and easily parallelizable programs [43].

Recording Explicit Non-Determinism (Synchronization) With source code, shared memory non-determinism can be recorded by instrumenting well defined interfaces for synchronization, modulo data races.

RecPlay [39] was an early system that recorded synchronization to enable replay debugging and race detection. It intercepted OS-level synchronization through shared library interposition.

Record overheads were quite low, around 2% for SPLASH-2 on a 4 core Sun SPARCServer 1000 [13]. Replay overheads could be quite high, ranging from 1.1×–4×. This would preclude its use for fault tolerance and similar applications [16]. With RecPlay, user level synchronization was manually converted to OS level synchronization, which could also add significant overhead.

Since RecPlay predated the C11 spec, benign data races were still an accepted form of optimization, race detectors were not yet common, etc. Thus, eliminating data races was part of embracing the RecPlay programming model.

R2 [22] recorded non-determinism at different layers of abstraction for improved comprehensibility and performance, e.g., at the MPI or SQL interface, for a broad selection of Win32 applications. Workloads were run on a dual-core Xeon, without parallel benchmarks. Non-determinism was recorded at the library interface using a system of annotations. R2's approach of capturing non-determinism at specific interfaces is probably best viewed as an optimization, complementary to our approach of recording language level non-determinism in situ as a default.

Arnold [18] explored eidetic computing, continuously recording all applications to make the lineage of any past byte available. Arnold reported SPLASH-2 numbers for up to 8 cores. Record overheads were often low, at worst around 20%. It incorporated the ability to use a race detector to dynamically instrument a binary to reproduce a single race. Synchronization was recorded through manual instrumentation. Rex [23] used record/replay to implement Paxos. It also relied on manually instrumenting synchronization.

Castor is largely complementary to this prior work in that all systems could incorporate techniques from Castor for scalability, performance, and transparently instrumenting user level synchronization.

8. Conclusion

We have presented Castor, a system that provides consistently low overhead recording and real-time replay for modern multi-core workloads, often with no modification to applications beyond recompilation. This makes it practical to leave recording on by default, to capture bugs in production, or to recover from hardware failures.

Castor offers several lessons for future implementers of record/replay systems. *Hardware optimized logging:* log bandwidth is the critical limiting factor for performance in this class of system. Castor's approach to hardware optimized logging can make an order of magnitude difference in log bandwidth, which is directly proportional to the amount of load a record/replay system can handle with low overhead. *Recording unmodified applications:* by interposing at the compiler level we can instrument user level non-determinism automatically, thus eliminating the tedious and error prone task of manually instrumenting applications for record/replay, and easing adoption. *Data races have negligible impact on replay:* for a variety of reasons data races generally have little practical impact on replay. Thus, its worth considering whether the benefit of trying to reproduce them precisely is worth the cost.

Acknowledgments

Dawson Engler, Phil Levis and Jim Chow offered helpful comments and discussion. Reviews of drafts by Russ Cox, Austin Clements, Laurynas Riliskis, and Edward Yang were very helpful for improving the content and presentation. Special thanks to Riad Wahby for his editorial help. This work was supported by funding from DARPA, ONR, and Google. And the Gnome.

References

[1] UndoDB Live Recorder: Record/Replay for Live Systems. http://jakob.engbloms.se/archives/2259, Nov. 2015.

[2] Reverse Debugging With Replay Engine. http://docs.roguewave.com/totalview/8.15.7/pdfs/ReplayEngine_Getting_Started_Guide.pdf, 2015.

[3] Simics 5 is Here. http://blogs.windriver.com/wind_river_blog/2015/06/simics-5-is-here-more-parallel-than-ever.html, June 2015.

[4] Lighttpd - fly light. https://www.lighttpd.net/, Jan 2017.

[5] VMware vSphere: What's New - Availability Enhancements. http://www.slideshare.net/muk_ua/vswn6-m08-avalabilityenhancements, Jan 2017.

[6] xxHash - Extremely fast non-cryptographic hash algorithm. http://cyan4973.github.io/xxHash/http://cyan4973.github.io/xxHash/, Jan. 2017.

[7] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 234–243, New York, NY, USA, 1991. ACM. ISBN 0-89791-394-9. doi: 10.1145/115952.115976. URL http://doi.acm.org/10.1145/115952.115976.

[8] G. Altekar and I. Stoica. ODR: Output-deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629594. URL http://doi.acm.org/10.1145/1629575.1629594.

[9] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] H.-J. Boehm. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2001252.2001255.

[11] H.-J. Boehm. Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 9–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1632-3. doi: 10.1145/2414729.2414732. URL http://doi.acm.org/10.1145/2414729.2414732.

[12] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375591. URL http://doi.acm.org/10.1145/1375581.1375591.

[13] K. D. Bosschere. Personal communication, May 2016.

[14] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224058. URL http://doi.acm.org/10.1145/224056.224058.

[15] Y. Chen and H. Chen. Scalable Deterministic Replay in a Parallel Full-system Emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 207–218, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442537. URL http://doi.acm.org/10.1145/2442516.2442537.

[16] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1404014.1404015.

[17] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013. URL http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[18] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 525–540, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685090.

[19] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. doi: 10.1145/1346256.1346273. URL http://doi.acm.org/10.1145/1346256.1346273.

[20] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267359.1267386.

[21] S. Ghemawat and J. Dean. GitHub - google/leveldb. https://github.com/google/leveldb, Jan 2017.

[22] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855755.

[23] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the Speed of Multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 11:1–11:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592800. URL http://doi.acm.org/10.1145/2592798.2592800.

[24] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International*

Symposium on Foundations of Software Engineering, FSE '10, pages 207–216, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882323. URL http://doi.acm.org/10.1145/1882291.1882323.

[25] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150997. URL http://doi.acm.org/10.1145/2150976.2150997.

[26] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1): 155–166, June 2010. ISSN 0163-5999. doi: 10.1145/1811099.1811057. URL http://doi.acm.org/10.1145/1811099.1811057.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563.

[28] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4): 471–482, Apr. 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.1676929. URL http://dx.doi.org/10.1109/TC.1987.1676929.

[29] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 77–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736031. URL http://doi.acm.org/10.1145/1736020.1736031.

[30] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254119. URL http://doi.acm.org/10.1145/2254064.2254119.

[31] J. Leverich. Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate, Jan 2017.

[32] Matt Holt. Caddy: The HTTP/2 web server with automatic HTTPS. https://caddyserver.com/, May 2016.

[33] MemCachier Inc. MemCachier. https://www.memcachier.com/, May 2016.

[34] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250738. URL http://doi.acm.org/10.1145/1250734.1250738.

[35] NGINX Inc. NGINX — High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/, Jan 2017.

[36] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Lightweight User-Space Record And Replay. *CoRR*, abs/1610.02144, 2016. URL http://arxiv.org/abs/1610.02144.

[37] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629593. URL http://doi.acm.org/10.1145/1629575.1629593.

[38] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/, Jan 2017.

[39] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999. ISSN 0734-2071. doi: 10.1145/312203.312214. URL http://doi.acm.org/10.1145/312203.312214.

[40] Y. Saito. Jockey: A User-space Library for Record-replay Debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 69–76, New York, NY, USA, 2005. ACM. ISBN 1-59593-050-7. doi: 10.1145/1085130.1085139. URL http://doi.acm.org/10.1145/1085130.1085139.

[41] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, 44(4):30–39, Dec. 2010. ISSN 0163-5980. doi: 10.1145/1899928.1899932. URL http://doi.acm.org/10.1145/1899928.1899932.

[42] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. doi: 10.1145/1791194.1791203. URL http://doi.acm.org/10.1145/1791194.1791203.

[43] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 227–240, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787501. URL http://doi.acm.org/10.1145/2785956.2787501.

[44] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247415.1247418.

[45] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC,

USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.13. URL http://dx.doi.org/10.1109/SP.2013.13.

[46] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, Jan 2017.

[47] The Golang Team. The Go Memory Model. https://golang.org/ref/mem, May 2014.

[48] The Memcached Team. Memcached: A distributed memory object caching system. http://memcached.org/, May 2016.

[49] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950370. URL http://doi.acm.org/10.1145/1950365.1950370.

[50] Wikipedia Foundation, Inc. Intel SHA extensions. https://en.wikipedia.org/wiki/Intel_SHA_extensions, Jan 2017.

[51] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM. ISBN 0-89791-698-0. doi: 10.1145/223982.223990. URL http://doi.acm.org/10.1145/223982.223990.

[52] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1. URL http://dl.acm.org/citation.cfm?id=774861.774871.