PAXOS Made Transparent

Heming Cui[+*], Rui Gu[*], Cheng Liu[*], Tianyu Chen[x], and Junfeng Yang[*]

[+]The University of Hong Kong [*]Columbia University [x]Tsinghua University

Abstract

State machine replication (SMR) leverages distributed consensus protocols such as PAXOS to keep multiple replicas of a program consistent in face of replica failures or network partitions. This fault tolerance is enticing on implementing a principled SMR system that replicates general programs, especially server programs that demand high availability. Unfortunately, SMR assumes deterministic execution, but most server programs are multithreaded and thus nondeterministic. Moreover, existing SMR systems provide narrow state machine interfaces to suit specific programs, and it can be quite strenuous and error-prone to orchestrate a general program into these interfaces

This paper presents CRANE, an SMR system that transparently replicates general server programs. CRANE achieves distributed consensus on the socket API, a common interface to almost all server programs. It leverages deterministic multithreading (specifically, our prior system PARROT) to make multithreaded replicas deterministic. It uses a new technique we call *time bubbling* to efficiently tackle a difficult challenge of nondeterministic network input timing. Evaluation on five widely used server programs (e.g., `Apache`, `ClamAV`, and `MySQL`) shows that CRANE is easy to use, has moderate overhead, and is robust. CRANE's source code is at `github.com/columbia/crane`.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Threads, Reliability; C.2.4 [**Computer-communication Networks**]: Distributed Systems;

General Terms: Algorithms, Design, Reliability, Performance

Keywords: State Machine Replication, Fault Tolerance, Stable and Deterministic Multithreading, Software Reliability

1. Introduction

State machine replication (SMR) models a program as a deterministic state machine, where the states are important program data and the transitions are deterministic executions of program code under input requests. SMR runs replicas of the program and invokes a distributed consensus protocol (typically PAXOS [42, 44, 65]) to ensure the same sequence of input requests for replicas, as long as a quorum (typically a majority) of the replicas agrees on the input request sequence. Under the deterministic execution assumption, this quorum of replicas must reach the same exact state despite replica failures or network partitions. SMR is proven safe in theory and provides high availability in practice [19, 21, 23, 33, 37, 51, 52, 61].

The fault-tolerant benefit of SMR makes it particularly attractive on implementing a principled replication system for general programs, especially server programs that require high availability. Unfortunately, doing so remains quite challenging; the core difficulty lies in the deterministic state machine abstraction required by SMR, elaborated below.

First, the deterministic execution assumption breaks down in today's server programs because they are almost universally multithreaded. Even on the same exact sequence of input requests, different executions of the same exact multithreaded program may run into different thread interleavings, or *schedules*, depending on such factors as OS scheduling and physical arrival times of requests. Thus, they can easily exercise different schedules and reach divergent execution states – a difficult problem well recognized by the community [14, 33, 34, 37]. To tackle this problem, one prior approach, execute-verify [37], detects divergence of execution states and retries, but it relies on developers to manually annotate states, a strenuous and error-prone process.

Second, to leverage existing SMR systems such as ZooKeeper [6], developers often have to shoehorn their programs into the narrowly defined state machine interfaces provided by these SMR systems. Ideally, experts – those with intimate knowledge of the arcane (think how many papers [23, 42, 44, 52, 65] are needed to explain PAXOS), under-specified [52] SMR protocols and subtle failure scenarios in distributed systems – should build a solid SMR system, which all other developers then leverage. However, an SMR system often has to settle for a specific state and transitional interface because it cannot anticipate all possibilities in which developers structure their programs. For example, Chubby [21] defines a lock server interface, and ZooKeeper a pseudo file system interface. Orchestrating a sever program into such a narrow interface not only requires intrusive and error-prone modifications to the program's structure and code, but also disrupts the SMR system itself at times. For instance, developers abused Chubby for storage [21], causing the Chubby developers to add quota support.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815427

This paper presents CRANE,[1] an SMR system that transparently replicates server programs for high availability. With CRANE, a developer focuses on implementing her program's intended functionality, not replication. When she is ready to replicate her program for availability, she simply runs CRANE with her program on multiple replicas. Within each replica, CRANE interposes on the socket and the thread synchronization interfaces to keep replicas in sync. Specifically, it considers each incoming socket call (e.g., `accept()` a client's connection or `recv()` a client's data) an input request, and runs a PAXOS consensus protocol [52] to ensure that a quorum of the replicas sees the same exact sequence of the incoming socket calls.

CRANE schedules synchronizations using *deterministic multithreading (DMT)* [13–15, 18, 31, 34, 56]. This technique typically maintains a *logical time*[2] that advances deterministically on each thread's synchronization. By serializing thread synchronizations, DMT practically makes an entire multithreaded execution deterministic. The overhead of DMT is typically moderate because most code is not synchronization and can still run in parallel. Specifically, CRANE leverages our prior DMT system PARROT [29], which incurs on average 12.7% overhead on a wide range of 108 popular multithreaded programs on 24-core machines.

A key challenge on realizing SMR for multithreaded executions is that, simply combining PAXOS and DMT is not sufficient to keep replicas in sync, because the physical time that each request arrives at different replicas may still be different, easily leading to divergence of execution states and outputs. (We illustrate this problem using an example in §2.2 and experimental results in §7.2.)

Two prior approaches attempted to tackle this challenge. Execute-agree-follow [33] records a partially ordered schedule of Pthreads synchronizations on one replica and replays it on the other replicas, which may incur high network bandwidth consumption and performance overhead. dOS [14] also leverages DMT for replication, but it determines the logical admission time for each request using two-phase commit. Aside from two-phase commit's known intolerance of primary failures, per-request commit is also costly.

One may consider solving this challenge by leveraging the underlying distributed consensus protocol to determine the logical admission time for each request. Specifically, when running the consensus protocol to decide each request's position in the request sequence, a predicted logical admission time can be carried as part of the decision as well. Unfortunately, predicting a logical admission time for each request accurately is quite challenging because typical server programs have background threads which may frequently tick logical clocks. A too-small predication leads to replica divergence if another replica has already run past the predicted logical time. A too-large predication blocks the system unnecessarily because replicas cannot admit the request before reaching the predicted time.

Our key insight is that many requests need no admission time consensus because their admission times are already deterministic. Hypothetically, if the requests arrive faster than they are admitted at each replica, each request's admission time is fully deterministic because each replica simply admits requests as fast as it can. In practice, requests do not arrive this fast. However, there are still frequent bursts of requests that arrive together. Among replicas, as long as the first request of a burst is admitted at a deterministic logical time, all the other requests in the burst are admitted at deterministic logical times without requiring consensus.

Leveraging this insight, we created an technique called *time bubbling* to enforce deterministic logical times efficiently. It ensures that the first request in a burst is admitted at each replica deterministically by inserting a deterministic wait after the previous burst of requests are all admitted. During this wait, each replica only processes already admitted requests, and does not admit new requests. CRANE negotiates a consistent duration of the wait via the underlying distributed consensus protocol, and enforces this wait at each replica via DMT. These waits are like deterministic time bubbles between bursts of requests (hence the name of the technique), creating the illusion that the requests arrive faster than they are admitted.

In short, by converting per-request admission time consensus to per-burst, time bubbling efficiently combines the input determinism of PAXOS and the execution determinism of DMT. For busy servers, requests in bursts greatly outnumber the other requests. (We observed that 66.65% to 93.88% of requests are in bursts; see §7.3.) They rarely need to invoke time consensus, enjoying good performance. For idle servers, time consensus overhead does not matter much because the servers are idle anyway.

We implemented CRANE by interposing on the POSIX socket and the Pthreads synchronization interfaces. It intercepts operations along these interfaces by hijacking dynamically linked library calls for transparency. It implements the PAXOS protocol atop libevent [47] for distributed consensus, and leverages our PARROT system for deterministic multithreading. Unlike prior SMR systems with narrow interfaces, CRANE's checkpoint and recovery must work with general programs. To this end, it leverages CRIU [28] to checkpoint and restore process states, and LXC [2] for file system states. An additional benefit of using the LXC container is that CRANE isolates the replicated server program from the environment, avoiding nondeterministic systems resource contentions (§5.2).

We evaluated CRANE on five widely used server programs, including HTTP servers `Apache` and `Mongoose`, an anti-virus server `ClamAV`, a uPnP multimedia server

[1] CRANE stands for Correctly ReplicAting Nondeterministic Executions. It is also our hope that our system is as elegant as the identically named bird.

[2] Though related, the logical time in DMT is not to be confused with the logical time in distributed systems [43].

`MediaTomb`, and a database server `MySQL`. Our results on popular performance benchmarks show that Crane works with all the servers easily (three servers require no modification, and the other two servers each require only two lines of Parrot hints [29] to improve performance); that Crane's performance overhead is moderate (an average of 34.19% overhead at the servers' peak performance setups on our 24-core machines); and that Crane is robust on replica failures.

Our key conceptual contribution is the idea of transparent SMR for general programs, which has the potential to expand SMR's adoption and improve availability of many systems. This idea also applies to other replication concepts (e.g., byzantine fault tolerance [22, 38]). This idea has other broad applications as well (§6.2). Our engineering contributions include the Crane system and its evaluation on widely used server programs. All Crane's source code (including a standalone, libevent-based Paxos implementation), benchmarks, and evaluation results are available at `github.com/columbia/crane`.

In the remainder of this paper, §2 introduces Crane's architecture and an example. §3 describes how Crane enforces order for synchronizations in a server. §4 illustrates the work flow of the time bubbling technique. §5 describes implementation details. §6 discusses the limitations and applications of Crane. §7 presents evaluation results. §8 discusses related work, and §9 concludes.

2. Crane Overview

Crane is deployed as a typical SMR system. A set of three or five replicas is set up within a LAN, and each replica runs an Crane instance containing the same server program. On the Crane system starts, one replica becomes the *primary* (or leader) replica which proposes the order of requests to execute, and the others become backup replicas which follow the primary's proposals. A number of clients in LAN or WAN send network requests to the primary and get responses. If the primary machine fails, the other replicas run a leader election (§5.1) to elect a new primary.

This section first presents Crane's architecture, including its consensus interface and a Crane instance's main components, and then uses an example to show how Crane works with server programs.

2.1 Architecture

To support general server programs transparently, Crane chooses the POSIX socket API as its consensus interface. Crane enforces two kinds of orders for socket calls. First, for client programs' out going socket calls (e.g., `connect()` and `send()`), Crane enforces that all replicas see the same sequence of client socket calls with Paxos. Crane does not need to order the clients' blocking socket calls because Crane is not designed to replicate clients. Second, for a server program's blocking socket calls (e.g., `poll()`, `accept()`, and `recv()`), Crane enforces that these calls are scheduled and returned in the same sequence of logical times across replicas. Crane responses to the clients only using the server program on the primary, and it drops the responses of the server programs on backups.

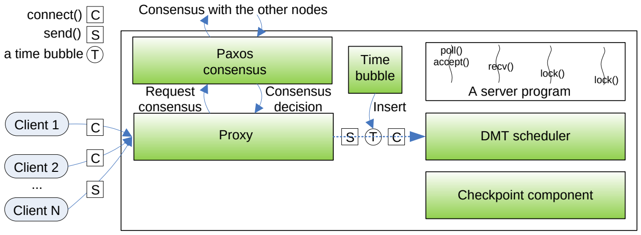

Figure 1: ***The* Crane *Architecture.* Crane components are shaded (and in green).**

For a server program's outgoing socket calls (e.g., `send()`), Crane simply schedules them using DMT and does not invoke consensus. The reason is that these calls readily have consistent contents via enforcing the same logical admission times of input requests and the same thread schedules for server programs across replicas.

Figure 1 shows a Crane instance running on the primary. The instance contains five main components, the proxy, the Paxos consensus, the DMT scheduler, the time bubbling component that enforces the same logical clocks for servers' blocking socket calls across replicas via inserting time bubbles, and the checkpoint component that periodically checkpoints the server program. A server program runs transparently in a Crane instance without being aware of Crane's components. A backup replica runs the same Crane instance except that its proxy does not accept connections from clients and does not invoke consensus.

The proxy component is a Crane instance's gateway. It accepts socket requests from clients and forwards the requests to the server program on its own replica. It accepts responses from the server program and forwards the responses to the clients. Once the proxy receives a client socket request, it invokes the Paxos consensus component running on its own replica for this request. The proxy does not block-wait for this decision which may take a while to reach. Once the proxy is notified by the Paxos component that some requests' decisions are made, it forwards the requests in decision order to the server program.

The Paxos consensus component is a Paxos protocol that receives a client socket request from its own proxy and invokes a consensus process on this request. This component is also the only Crane component that communicates among different replicas. Crane's Paxos implementation is based on a well-known and concise protocol [52]. More details on our Paxos implementation are given in §5.1. After Crane's Paxos components reach consensus on a client socket call, each Paxos component notifies its own proxy to forward this call to its server program.

The DMT component runs within the server program's process. Crane leverages Parrot [29] as the DMT scheduler because Parrot runs fast on a wide range of 108 popular multithreaded programs. Specifically, PARROT uses a runtime technique called LD_PRELOAD to dynamically intercept Pthreads synchronizations (e.g., pthread_mutex_lock()) issued by an executable and enforces a well-define, round-robin schedule on these synchronization operations for all threads, practically eliminating nondeterminism in thread synchronizations.

Although PARROT is not designed to resolve data races deterministically, CRANE's replication tolerates data races that have fail-stop consequences, and can further catch the other data races by running a race detector on a backup replica (see §6.1). CRANE augments the DMT component to schedule the return points of socket calls in server replicas, too, to ensure that requests are admitted at consistent logical times across replicas.

The time bubbling component sits between the proxy and the DMT's processes, and it is invoked on two conditions. First, on a server's bootstraps, CRANE invokes time bubble insertions to make sure that the server programs across replicas reach the same initial state and wait for the first input request. Second, if the DMT component has not received any input request from the proxy for a physical duration $W_{timeout}$, a time bubble insertion is invoked as the boundary of two request bursts. To ensure the same sequence of inserted time bubbles across replicas, the same PAXOS consensus as that for client socket calls is invoked. For each time bubble, each replica's DMT scheduler promises to run a number of N_{clock} synchronizations and not to admit any client socket call.

If the DMT scheduler exhausts the logical clocks in a time bubble, it either admits new client socket call (if any) or inserts another time bubble. If the scheduler does not exhaust the logical clocks after serving current requests, PARROT has a mechanism to exhaust them rapidly (§3.1). More details on the time bubbling technique are given in §4, and discussions on the values of the two parameters $W_{timeout}$ and N_{clock} are given in §7.5.

To recover from replica failures or add new replicas, the checkpoint component is invoked every minute on a backup replica. It checkpoints the server process running with DMT. While one can always start a server replica from scratch and replay the entire sequence of socket calls, this replay can be extremely time-consuming for long-running servers. Prior SMR systems rely on narrow state machine interfaces for checkpoint and recovery, which does not work for general server programs. Instead, CRANE leverages two popular open source tools: CRIU, to checkpoint process state such as CPU registers and memory; and LXC, to checkpoint the file system state of a server program's current working directory and installation directory.

Each checkpoint in CRANE is associated with a global index in PAXOS's consensus order, so if one replica needs recovery, CRANE ships the latest checkpoint from a backup replica, restores the process running DMT and the server program, and re-executes socket calls starting from this index. The proxy and consensus components do not require checkpoints because we explicitly designed their execution states independent to the server's process (§5.2).

```
void main(int argc, char *argv[]) {
  int done = 0; // Be 1 when receives a kill signal.
  int nworkers = atoi(argv[1]);
  pthread_create(..., NULL, listener, NULL);
  for (i = 0; i < nworkers; ++i)
    pthread_create(..., NULL, worker, NULL);
  ...; // Wait for threads to exit.
}
void *listener(void *arg) {
  ...; // Call bind() and listen().
  while (!done && poll(...)) {
      int sock = accept(...);
      worklist.add(sock);
  }
}
void *worker(void *arg) {
  while(!done && int sock = worklist.get()) {
    recv(sock, buf, ...);
    lock(m);
    ret = process_req(buf);
    unlock(m);
    send(sock, ret, ...);
    ...;
  }
}
```

Figure 2: ***A server example based on*** Apache.

```
void main(argc, char *argv) {
  ...; // Get server IP:port from argv[ ].
  int sock = socket(...);
  connect(sock, ...); // Connect to IP:port.
  send(sock, ...); // Send a http request.
  recv(sock, ...); // Wait for server's response.
  close(sock);
}
```

Figure 3: ***A client example based on*** curl.

2.2 Example

Figure 2 shows an example based on the Apache HTTP server. For clarity, the example uses worklist synchronization, and the actual servers use Pthreads mutex locks and conditional variables which CRANE readily handles. The main thread creates a listener thread to accept client requests and a number of worker threads to process client requests in parallel. The listener listens on a port with poll(). When a new client connection comes, the listener calls accept() and appends the accepted socket descriptor to a worklist. Each worker thread blocks on a worklist.get() function until the worklist is not empty. It then dequeues an accepted socket, processes the request with a mutex lock acquired, and then sends a response. Figure 3 shows an example based on client programs such as curl. This client connects to the server, sends one HTTP request, waits for the server's response, and then closes the connection.

```
//    worker 1                  worker 2
1:  recv("PUT a.php");
2:  lock(m);
3:  ret = process_req();
4:  unlock(m);
5:  send(ret);
6:                              recv("GET a.php");
7:                              lock(m);
8:                              ret = process_req();
9:                              unlock(m);
10:                             send(ret); //200 OK
```

Figure 4: ***HTTP GET request got the valid page due to the two requests' large arrival interval.***

```
//    worker 1                  worker 2
1:  recv("PUT a.php");
2:                              recv("GET a.php");
3:                              lock(m);
4:                              ret = process_req();
5:                              unlock(m);
6:  lock(m);
7:  ret = process_req();
8:  unlock(m);
9:  send(ret);
10:                             send(ret); //404 Not Found
```

Figure 5: ***HTTP GET request didn't get the page due to the two requests' small arrival interval.***

Let's say a CRANE system with three replicas is set up, and each replica runs this server; two clients start simultaneously, and each sends a HTTP PUT and GET request respectively on the same URL "a.php" to the primary.

This server has three major sources of nondeterminism, which can easily cause its execution states across replicas to diverge. The first source (for short, S_1) is that clients' requests may arrive at different replicas in different orders. Second (S_2), within the server, the nondeterministic Pthreads synchronizations may easily lead to different schedules. For instance, the `worklist.add()` called by the listener may wake up any worker blocking on `worklist.get()`.

Third (S_3), even if clients' requests arrive at different replicas in the same order, the physical time interval of each two consecutive requests can still be largely different across replicas depending on each request's physical arrival time. This variance may cause client socket calls to be admitted at inconsistent logical clocks across replica and lead to divergent execution states.

For instance, Figure 4 and Figure 5 show two replicas' schedules, and each is a total order of executed socket calls or Pthreads synchronizations in workers. Although the PUT and GET requests arrive at the two replicas in the same order, these requests' interval in the first schedule (Figure 4) is much larger than that in the second one, causing the first one to return a page and the second one "404 Not Found".

```
1: connect();
2: connect();
3: send("PUT a.php");
4: send("GET a.php");
5: a time bubble;
6: close();
7: close();
```

Figure 6: ***A sequence of client socket calls enforced by* CRANE.**

```
//   Listener          worker 1               worker 2
1:   poll();
2:   accept();
3:   worklist.add();
4:                     worklist.get();
5:   poll();
6:   accept();
7:   worklist.add();
8:                                            worklist.get();
9:                     recv("PUT a.php");
10:                                           recv("GET a.php");
11:                                           lock(m);
12:                                           ret = process_req();
13:                                           unlock(m);
14:                    lock(m);
15:                    ret = process_req();
16:                    unlock(m);
17:                    send(ret);
18:                                           send(ret);//404 Not Found
```

Figure 7: ***A schedule of the server example enforced by* CRANE *across replicas.***

CRANE works as follows. First, depending on the order the primary's proxy receives these requests, CRANE eliminates S_1 with PAXOS and ensures the same request sequence for all replicas. Second, CRANE's DMT scheduler eliminates S_2 by ensuring a deterministic order of Pthreads synchronizations.

Third, depending on the time intervals of client socket calls observed by the primary, CRANE eliminates S_3 by dividing this sequence of calls into bursts with time bubbles. Figure 6 shows a sequence. Let's say the primary observes that the intervals of the `connect()` and `send()` calls are all smaller than $W_{timeout}$, and the interval between the `send()` at Line 4 and the `close()` at Line 6 in the sequence is larger than $W_{timeout}$. Then, CRANE inserts a time bubble at Line 5 and divides the sequence into two bursts. For the calls in the first burst, all replicas admit them as is using DMT even if some of their time intervals are larger than $W_{timeout}$ on some backups, consistently maintaining the logical admission times for these calls.

For the time bubble in this sequence, all replicas' DMT schedulers promise to do only N_{clock} Pthreads synchronizations and then admit a client socket call, consistently maintaining the logical admission times for the `close()` calls. Given this sequence, Figure 7 shows CRANE's consistent schedule and output "404 Not Found" across replicas. We ran CRANE with `Apache` and used `curl` to spawn concurrent PUT and GET requests, and we observed that `Apache`'s network outputs were consistent across replicas (§7.2).

In addition to enforcing consistency, the time bubbling technique is also efficient because it does per-burst consensus instead of per-request consensus. In this example,

```
void get_turn();
void put_turn();
void wait(opaque obj);
void signal(opaque obj);
```

Figure 8: ***The* PARROT *DMT runtime's scheduler primitives.***

if more `connect()` calls arrive simultaneously and each client connection does more `send()` calls, the ratio of inserted time bubbles versus the total number of socket calls in the sequence may be even lower, then CRANE may become more efficient. Evaluation on popular servers and workloads shows that this ratio is often low and that CRANE has moderate overhead (§7.3).

3. CRANE's Synchronization Wrappers for a Server

This section describes how CRANE handles a server program's synchronizations, including Pthreads synchronizations and blocking socket calls. Because how to handle these synchronizations is tightly relevant to the PARROT DMT scheduler we leverage, in this section, we first introduce some background on the PARROT scheduler, including its primitives and wrappers. And then we describe how CRANE leverages PARROT's primitives and wrappers to implement its own synchronization wrappers.

3.1 Background: the PARROT Scheduler

PARROT [29] is a DMT system that uses the LD_PRELOAD trick to intercept Pthreads synchronizations at runtime and enforces a well-define, round-robin order for these operations. In this round-robin manner, PARROT first lets one runnable thread do one synchronization operation; and then, for the left runnable threads, PARROT lets the next thread do one synchronization operation; and then the next runnable thread, until all runnable threads having done one synchronization operation. Then PARROT repeats. To enforce this schedule, PARROT maintains a queue of runnable threads (*run queue*) and another queue of waiting threads (*wait queue*), like a Linux OS scheduler.

PARROT enforces an important invariant: only the thread at the head of the run queue can do one actual synchronization operation and manipulate the run queue and wait queue. After the head thread does one operation, it rotates itself to the tail of the run queue and wakes up the new head thread of the run queue. Conceptually, threads within PARROT pass a global token (the run queue head) around. A thread will be put into the wait queue if the synchronization object it requires is not available, and it will be put back to the run queue when this object becomes available.

To implement this round-robin schedule in a compact way, PARROT provides a monitor-like internal interface, shown in Figure 8. The `get_turn()` function waits until the calling thread becomes the head of the run queue. The `put_turn()` function rotates the calling thread to the tail of the run queue and wakes up the next thread which now is the head of the run queue. The `wait()` function puts the calling thread from run queue to wait queue and blocks on a opaque object (e.g., a mutex or a socket descriptor), until another thread makes this object available and calls a `signal()` on this object. When a thread returns from a `wait()` function, it becomes the head of the run queue. Both the `wait()` and the `signal()` functions require getting the global turn.

```
int pthread_mutex_lock_wrapper(mu) {
    DMT.get_turn();
    check_add_timebubble(); // NOP in Parrot. Called in Crane.
    while (pthread_mutex_trylock(mu))
        DMT.wait(mu);
    DMT.put_turn();
    return 0; // Error handling code ommitted for clarity.
}
```

Figure 9: **PARROT*'s wrapper for* `pthread_mutex_lock()`*.***

These set of primitives are highly optimized for multicore. Each thread has an integer flag and condition variable. The `get_turn()` function spin-waits on the current thread's flag for a while before blocking on the condition variable. The `wait()` function needs to get the turn before it returns, so it uses the same combined spin- and block- wait strategy as the `get_turn()` function. The `get_turn()` and `signal()` functions signal both the flag and the conditional variable of the next thread. In common case, these operations acquire no lock and do not block-wait, thus the number of synchronization context switches in PARROT is much smaller than that in traditional Pthreads synchronizations, yielding faster performance in PARROT than in the Pthreads runtime for some programs [29].

Figure 9 shows the `pthread_mutex_lock()` wrapper in PARROT. This wrapper uses try-lock to avoid deadlock: if the head of the run queue is blocked waiting for a lock before giving up the turn, no other thread can get the turn.

When all threads of a program block, which is common case in a server program, PARROT puts an internal *idle thread* to the run queue, which simply does repetitive `get_turn()` and `put_turn()` operations. This idle thread ensures that PARROT's run queue always has threads and that PARROT's logical clock keeps ticking.

PARROT's blocking socket calls are nondeterministic because it is a DMT system for eliminating nondeterminism in Pthreads synchronizations. A blocking socket call's wrapper in PARROT works as follows. When a thread calls a blocking socket call, the thread calls `get_turn()`, passes the global token to the next thread in the run queue, removes itself from the run queue, and then calls into the actual socket call. When the thread returns from the actual call, it appends itself to a *socket queue*. Each thread at the run queue head moves the threads in this socket queue back to the run queue. This move-back is nondeterministic because threads may return from blocking socket calls nondeterministically and thus may be added to the socket queue in various orders.

```
1:  void check_add_timebubble(mu) {
2:    while (paxos_seq.empty()) {
3:      usleep(...);
4:      request_time_bubble();
5:    }
6:    if (paxos_seq.head().type == TIME_BUBBLE)
7:      paxos_seq.head().decrease();
8:    else
9:      DMT.signal(paxos_seq.head());
10: }
```

Figure 10: CRANE's *`check_add_timebubble()`* function.

```
int recv_wrapper(sockfd, ...) {
  DMT.get_turn();
  DMT.wait(sockfd);
  int nbytes = recv(sockfd, ...);
  paxos_seq.dequeue(nbytes);
  DMT.put_turn();
  return nbytes;
}
```

Figure 11: CRANE's *wrapper for `recv()`.*

3.2 CRANE' Synchronization Wrappers for a Server

CRANE wraps a rich set of common blocking socket operations, including `select()`, `poll()`, `epoll_wait()`, `accept()`, and `recv()`. CRANE also modifies the wrappers of Pthreads synchronizations. These wrappers are sufficient for the server programs in our evaluation.

CRANE needs to modify the `pthread_mutex_lock()` wrapper in Figure 9 to do three things. First, if the PAXOS request sequence has been empty for a physical duration $W_{timeout}$, CRANE requests a time bubble with N_{clock} logical clocks. Second, if the head of the PAXOS sequence is a time bubble, CRANE decreases the logical clock in the time bubble by one, or it removes this bubble if zero clock is left. Third, CRANE signals a thread that blocks on a socket operation (e.g., `recv()`) if there is a matching client socket call (e.g., `send()`) at the head of the PAXOS sequence. To do these three things, CRANE calls the `check_add_timebubble()` function (defined in Figure 10) at Line 3 of the `pthread_mutex_lock()` wrapper in Figure 9.

An important data structure in CRANE's wrapper is the PAXOS sequence which contains clients' socket calls and inserted time bubbles. This sequence sits between the proxy and the server's processes, and it is implemented with Boost [1] shared memory. CRANE uses `lockf()` to ensure mutual exclusion on this sequence because the two processes may concurrently manipulate this sequence. For clarity, these lock and unlock operations are omitted in the pseudo code.

CRANE also needs to modify PARROT's idle thread mechanism because sometimes this thread is the only thread in the run queue, and CRANE needs to frequently check whether a new client socket call comes or a time bubble insertion is needed. To do so, CRANE replaces PARROT's `get_turn()` and `put_turn()` primitives within the idle thread to be mutex lock and unlock operations, then the idle thread also runs the function defined in Figure 10 to check and insert time bubbles.

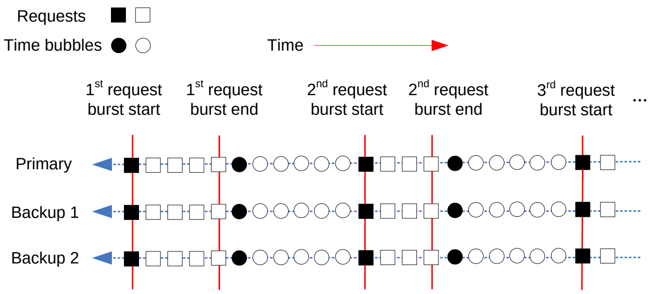

Figure 12: *The request and time bubble flow.*

Figure 11 shows CRANE's wrapper for the `recv()` call. This wrapper ensures that the `recv()` calls of server programs across replicas return at consistent logical times. The other blocking socket calls' wrappers are similar. A thread calling `recv()` in CRANE simply calls `get_turn()` and blocks on the socket descriptor using PARROT's `wait()` primitive. When a client `send()` call that matches this `recv()` becomes the head of the PAXOS sequence, the `pthread_mutex_lock()` wrappers wakes up the server thread blocking on `recv()` with the `signal()` call at Line 9 in Figure 10. The waken up thread dequeues a number of matching `send()` calls from the PAXOS sequence according to the actual bytes received. Also, for clarity, the lock and unlock operations for the PAXOS sequence are omitted in this `recv()` wrapper.

4. The Time Bubbling Technique

Figure 12 shows the time bubbles inserted by the time bubbling technique. The technique groups clients' socket operations as bursts. A request burst can be a group of real socket requests (rectangles), or can be a time bubble with a fixed number of logical clocks (circles). In this figure, black requests are the first operation for each burst.

In a conceptual level, CRANE uses three rules to enforce the same sequence of logical times for socket requests (rectangles) and thus the same schedules across different replicas. First, CRANE uses PAXOS to ensure the same sequence of client socket calls as well as inserted time bubbles as a "PAXOS request sequence" for each replica, as shown in each horizontal arrow. Second, CRANE uses DMT to guarantee that it only ticks logical clocks (i.e., schedules Pthreads synchronizations or socket operations) when this sequence is not empty. Third, the time bubbling technique ensures that this sequence is not empty, otherwise it inserts a time bubble.

Figure 13 shows the work flow of our time bubbling technique with four steps. Each replica's DMT just waits for a physical duration $W_{timeout}$, if no further requests come, (1) the DMT requests its own proxy to insert a time bubble. (2)

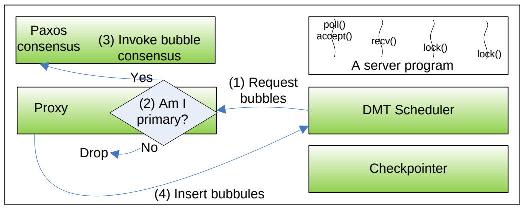

Figure 13: ***The work flow of inserting a time bubble.***

The proxy then checks whether it sees itself as the primary in the Paxos protocol. If so, it asks (3) the consensus component to invoke consensus on whether inserting this bubble; otherwise it drops this request. After a consensus on this bubble insertion is reached, (4) each machine's proxy simply inserts the bubble into the Paxos sequence, granting N_{clock} logical clocks to the DMT scheduler.

If a server has not exhausted the logical clocks in a time bubble after serving current requests, Parrot's idle thread mechanism (§3.1) exhausts these clocks rapidly. Then, the server can continue to process further requests in time.

5. Implementation Details

5.1 The Paxos Protocol

The Paxos consensus component (§2.1) is a critical component to enforce a consistent total order of socket calls from client programs. Although there are already a number of open source Paxos implementations [6, 26, 60], we reimplemented a Paxos protocol in order to incorporate our new socket-API consensus interface.

Our Paxos implementation is based on a well-known and concise approach [52]. In normal case, only the primary invokes consensus, thus this approach reaches consensus efficiently. In exceptional cases such as primary restarts, a Paxos leader election is invoked to resolve conflicts. In Crane, we implemented this election via making the primary send a heart beat message to all the backups every second, and if backup replicas have not receive any heart beat message for three seconds, these replicas start to elect a new leader. The leader election contains three steps [52]: (1) backups proposing a new view, which is a standard Paxos two-phase consensus [42], (2) the proposer that wins the view proposing itself as a primary candidate, another standard Paxos two-phase consensus, and (3) the new leader announcing itself as the new primary.

In our implementation, each socket call from the client is assigned a global, monotonically increasing viewstamp (or *global index*) to associate with each checkpoint (§5.2). Upon consensus on a socket call, each consensus component persistently stores the call type, arguments, and global index into a Berkeley DB storage [17] on SSD.

Although our current Paxos implementation focuses on supporting socket consensus interface, this Paxos protocol logic is independent of the types and arguments of socket operations, so our Paxos implementation can be applied to other types of consensus interface as well.

5.2 Checkpoint and Restore

To recover or add a new replica, Crane leverages two popular open source tools: CRIU [28], to checkpoint process state such as CPU registers and memory; and LXC [2], to checkpoint the file system state of a server program's current working directory and installation directory. These two directories are sufficient to capture files modified by the server programs in our evaluation.

Incorporating LXC into Crane has two extra practical benefits. First, the server process is ran within an LXC, which provides the server the same and clean initial systems state and mitigates contentions on systems resources (e.g., file descriptors) with other processes. Second, LXC snapshots make Crane easy to deploy on multiple replicas without worrying about slight differences of the systems environments such as kernel and library versions. We just built Crane on one replica once, did a LXC snapshot, and then copied the snapshot to other replicas.

A Crane checkpoint operation contains three steps. First, Crane uses CRIU to checkpoint the server's process running within the LXC container and dumps the checkpoint to the process's current working directory. CRIU needs to modify systems files (e.g., ns_last_pid), but LXC's default isolation configuration does not permit these modifications, so we configure LXC to run in "unconfined mode". Second, Crane stops the container, uses "`diff --text`" to generate a patch of current working directory and the server's installation directory against an LXC snapshot prepared before any server starts. This file system checkpoint patch is incremental and thus efficient (§7.6). Third, Crane restarts the container, and restores the server process with CRIU.

Such a Crane checkpoint operation is done every minute on one backup replica without affecting the other replicas' performance. We explicitly design Crane's proxy and consensus component stateless and they do not require checkpoints. A Crane restore operation reverts these steps.

One main issue on checkpointing a server process is that it constantly accepts socket connections, but checkpointing and restoring TCP stacks are notoriously difficult. Our trick to avoid this difficulty is based on an observation: even busy server programs have some idle moments. For instance, consider `Apache`, even running with its standard performance-stress benchmark `ApacheBench`, we observed that in some moments the server has no alive socket connections. Thus, during a checkpoint operation, Crane simply checks whether the server has alive connections. If so, Crane backs off for a few seconds and then retries until the server has no alive connections. Since checkpoint periods do not have to be precise, this trick runs well (§7.6).

6. Discussions

This section first discusses CRANE's limitations and then introduces its applications.

6.1 Limitation

CRANE leverages PARROT to make synchronizations deterministic. PARROT is explicitly designed not to handle data races. However, in the context of CRANE, data races are less harmful because, if they cause backups to crash, CRANE can still operate and recover as long as a quorum of the replicas is still alive. Moreover, leveraging CRANE's replication architecture, one can deploy a race detector on a backup replica [30], achieving both good CRANE performance and full determinism.

There are other sources of nondeterminism besides thread scheduling and request timing. These other sources of nondeterminism may cause backups to diverge, too. For example, backups may do different things based on their IP addresses, data read from `/dev/random`, addresses returned by `malloc`, physical time observed via `gettimeofday`, or delivery time of signals. Prior work has shown how to eliminate these sources of nondeterminism using record-replay [40, 46] or OS-level techniques [14], which CRANE can leverage. Another solution is to treat all these sources as inputs and leverage distributed consensus to let all replicas observe the same input. We leave these ideas for future work. We inspected server programs' network outputs among replicas, and we found that these outputs were consistent in CRANE except physical times (§7.2).

For a server program that spawns multiple processes which communicate via IPC, CRANE currently does not make these IPC operations deterministic. We expect that it should be easy to support deterministic IPC in CRANE because it already makes socket API deterministic. In addition, dOS [14] and DDOS [34] have many effective techniques for tackling this problem, which CRANE can leverage.

6.2 Applications

We envision three applications for CRANE. First, CRANE can be leveraged by other replication concepts (e.g., byzantine fault tolerance [22, 38]) and record-replay [39, 41, 46] because they also suffer from nondeterminism. Second, promising results in REPFRAME [30] have shown that CRANE's transparent replication architecture can enable multiple types of program analysis tools within one execution, making a server program enjoy benefits of multiple analyses. Third, CRANE's determinism as well as its time bubbling technique alone can be applied to mitigate timing channels [10, 11, 70].

7. Evaluation

Our evaluation was done on a set of three replica machines, with each having Linux 3.13.0, 1Gbps bandwidth LAN, 2.80 GHz dual-socket hex-core Intel Xeon with 24 hyper-threading cores, 64GB memory, and 1TB SSD.

We evaluated CRANE on five widely used server programs, including HTTP servers `Apache` [9] and `Mongoose` [54]; `ClamAV` [24], an anti-virus scanning server that scans files in parallel and deletes malicious ones; `MediaTomb` [8], a uPnP multimedia server that uploads, shares, and transcodes pictures and videos in parallel; and `MySQL` [3], an SQL database. Although `MySQL` has a replication feature [4], this feature is mainly for improving read performance, not for providing SMR fault tolerance.

SMR's high availability and fault-tolerance are attractive to these servers programs, because these programs provide on-line service and contain important in-memory execution states and storage (e.g., `ClamAV`'s security database, `MediaTomb`'s SQLite [5] database, and `MySQL`).

For `Apache` and `Mongoose`, we used `Apache`'s own concurrency stress testing benchmark `ApacheBench` to invoke concurrent HTTP requests for a PHP page, which takes about 70 ms for a PHP interpreter to generate the page contents. For `ClamAV`, we used its own client utility `clamdscan` to request the server to scan `ClamAV`'s own source code and installation directories in parallel. For `MediaTomb`, because it has a web interface, we used `ApacheBench` to invoke concurrent requests which use `mencoder` [53] to transcode a 15MB video from AVI to MP4. For `MySQL`, we used `SysBench` [7] to generate random select queries. These workloads triggered 8$\sim$12 threads in each server program to process requests concurrently at peak performance on our machines. These popular benchmarks and workloads cover CPU, network, and file-IO bounded operations.

CRANE has two parameters for the time bubbling technique. The first parameter, $W_{timeout}$, is the physical duration that the primary's DMT scheduler waits before it requests consensus on a time bubble insertion. To prevent this parameter significantly deferring responses, CRANE sets its default value 100us, two orders of magnitudes smaller than the workloads' response times and wide-area network latencies.

The second parameter, N_{clock}, is the number of logical clocks within each time bubble. CRANE sets its default value 1000, because we observed that the amounts of executed Pthreads synchronizations to process each request in most of the evaluated servers are closed to this scale. We used these default values in all evaluations unless explicitly specified. A sensitivity evaluation on these two parameters showed that their default values were reasonable choices (§7.5).

To mitigate network latency, benchmark clients were ran within the replicas' LAN. Larger latency will mask CRANE's overhead. We measured each workload's response time as it has direct impact on users. For each data point, we ran 1K requests for 20 times and then picked the median value.

The rest of this section focuses on these questions:

§7.1: Is CRANE easy to use?

§7.2: Compared to nondeterministic executions, does CRANE consistently enforce the same sequence of network outputs among replicas?

§7.3: What is Crane's performance overhead compared to nondeterministic executions?

§7.4: When the default schedules enforced by the Parrot DMT scheduler are slow, how much optimization can Parrot's performance hints bring to Crane?

§7.5: How sensitive are the two time bubbling parameters to Crane's performance?

§7.6: How fast are Crane's checkpoint and recovery components on handling replica failures?

7.1 Ease of Use

All five servers we evaluated were able to be transparently plugged and played in Crane without modification. For `ClamAV`, `MediaTomb`, and `MySQL`, we did not need to modify any line of code and they already have moderate performance overhead compared to the un-replicated nondeterministic executions. For `Apache` and `Mongoose`, the default schedules serialized parallel computations. For each of the two servers, we added two lines of soft barrier performance hints invented by Parrot [29] to line up parallel computations as much as possible and compute efficient DMT schedules (cf §7.4).

7.2 Consistency of Network Outputs

To verify whether the server programs running in different replicas maintain the same execution states, we compared each server program's network outputs logged in three replicas. Network outputs imply a server's execution states, including the outcomes of ad-hoc synchronizations and data races, which synchronization schedules can not capture. We ran the performance workloads and logged the order and contents of server programs' outgoing socket calls, including `send()`, `sendto()`, `sendmsg()`, `write()`, and `pwrite()`. These calls are sufficient to capture all network outputs of the evaluated programs. We then used `diff` to compare the logs across replicas.

We designed two experiment plans. In plan I, we ran Crane with the programs. In plan II, we disabled only the time bubbling component in Crane for three reasons: (1) we wanted to know whether time bubbling is needed to keep replicas in sync, (2) enabling Paxos made us easy to ship the same workload to replicas, and (3) enabling Parrot made us easy to intercepted and logged network outputs.

Among the five programs, three server programs, `Apache`, `MediaTomb`, and `Mongoose`, used `ApacheBench` to spawn workloads. In plan I, Crane's logs from all three replicas had the same order and contents of outputs except physical times in the responded HTTP headers. In plan II, despite that we disabled only the time bubbling component, the logs' order of responded HTTP headers and contents across replicas were different. Two server programs, `ClamAV` and `MySQL`, used specific benchmarks to spawn workloads. In plan I, the logs showed that Crane enforced the same network outputs. In plan II, the orders of the outputs across replicas were different. These experiments suggest that simply combining Paxos and DMT is not sufficient to keep replicas in sync, and the time bubbling technique is needed.

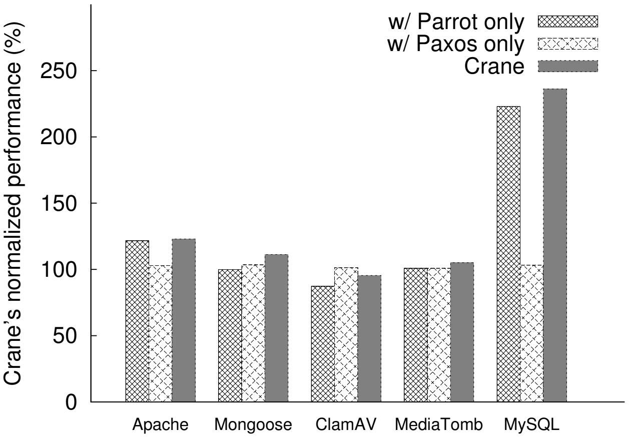

Figure 14: Crane*'s performance normalized to un-replicated nondeterministic execution.*

To diagnose consistency of network outputs more concisely, we wrote a micro-benchmark for `Apache`. We used the `curl` utility to spawn two concurrent HTTP requests: a PUT request of a PHP page and a GET request on this page, and then we inspected the outcome of the GET request. We ran `Apache` in Crane with this micro benchmark for 100 times and found that three replicas consistently reported the same GET result in each run, either "200 OK" or "404 Not Found", depending on the order of the PUT and GET request arriving at the primary's proxy. And then we ran `Apache`'s un-replicated execution for 100 times on each replica, and three replicas reported "404 Not Found" for 6, 8, 11 times respectively.

7.3 Performance Overhead in Normal Case

To understand the performance impact of Crane's components, we divided Crane's components into two major parts: the DMT part ran by Parrot; and the proxy (with Paxos) part which enforces the same sequence of client socket calls across replicas. Each part ran independently without the other part. The proxy part represents the performance overhead of invoking Paxos consensus for client socket calls, and the DMT part represents the Parrot DMT scheduler's overhead.

Figure 14 shows the servers' performance running in Crane normalized by their un-replicated nondeterministic executions. The mean overhead of Crane for the five evaluated programs is 34.19% due to two main reasons. First, except for `MySQL`, which does fine-grained, per-table mutex and read-write locks frequently, the DMT schedules were efficient on the other four servers. The reason is that Parrot's scheduling primitives are already highly optimized for multi-core (§3.1). The proxy-only part incurred 0.82%∼3.46% overhead, which is not surprising, because the number of socket calls is much smaller than the number of Pthreads synchronizations in these programs. In short, CRANE's performance mainly depends on the DMT schedules' performance.

`MediaTomb` incurred modest speedup because its transcoder `mencoder` had significant speedup with PARROT. We inspected `MediaTomb`'s micro performance counters with the Intel `VTune` [66] profiling tool. When running in CRANE, `MediaTomb` only made 6.6K synchronization context switches, while in the Pthreads runtime it made 0.9M synchronization context switches. This saving caused `MediaTomb` running with PARROT a 12.76% speedup compared to its nondeterministic execution. The PARROT evaluation [29] also observed a 49% speedup on the `mencoder` program.

The time bubbling technique saves most of needs on invoking consensus for the logical times of clients' socket operations, confirmed by the low frequency of inserted time bubbles in Table 1. `Apache`, `MediaTomb`, and `Mongoose` uses `ApacheBench` as its benchmark, and each request contained a `connect()`, `send()`, and `close()` call. `ClamAV` uses its own `clamdscan` benchmark, and each request contained 18 socket calls. `MySQL`'s benchmark contained 6~7 socket calls for each query. The ratio of inserted bubbles is merely 6.12%~33.35%. `MediaTomb` had the highest ratio of time bubbles because it took the longest time (9,703ms) to process each request.

Note that the number of inserted time bubbles across replicas is the same within the same run of CRANE. Within different runs of CRANE, this number can be different because $W_{timeout}$ is a physical duration.

7.4 Optimization of PARROT's Performance Hints

In general, a DMT schedule may be slow in some cases [29, 48], because this schedule may *serialize* some major computations that can run in parallel in the Pthreads runtime. For instance, when we ran CRANE's DMT scheduler PARROT with `Apache` and `Mongoose`, we observed that PARROT's default schedules serialized the PHP interpreters.

Fortunately, PARROT creates a set of easy to use, intuitive soft barrier hints [29] which tell the DMT runtime to switch to faster schedules. These hints are just "soft" barriers; they timeout deterministically and can tolerate different number of concurrent incoming requests. They just make a (deterministic) effort to line up computations that tend to run in parallel. In addition, these hints can be safely ignored by the PARROT runtime without affecting a program's logic.

In our evaluation, we added two lines of hints for each of the `Apache` and `Mongoose` servers' source code, and the pattern was general: one line was added at the server's `main()` function to initialize the soft barrier, and the other before a PHP interpretation's start to tell the DMT scheduler "these are the major computations to line up". The performance optimization effects of these hints are shown in Figure 15. These hints reduces `Apache`'s overhead from a 424% to 22.99%, and `Mongoose`'s from a 643% to 5.09%.

Program	# client socket calls	# time bubbles	%
`Apache`	3,000	450	13.04
`ClamAV`	18,000	1,173	6.12
`MediaTomb`	3,000	1,501	33.35
`Mongoose`	3,000	448	12.99
`MySQL`	6,750	573	7.82

Table 1: ***Ratio of time bubbles in all* PAXOS *consensus requests.***

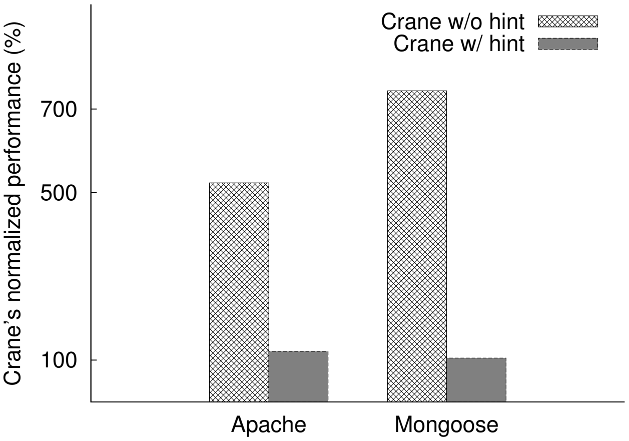

Figure 15: Effects of PARROT's soft barrier performance hints.

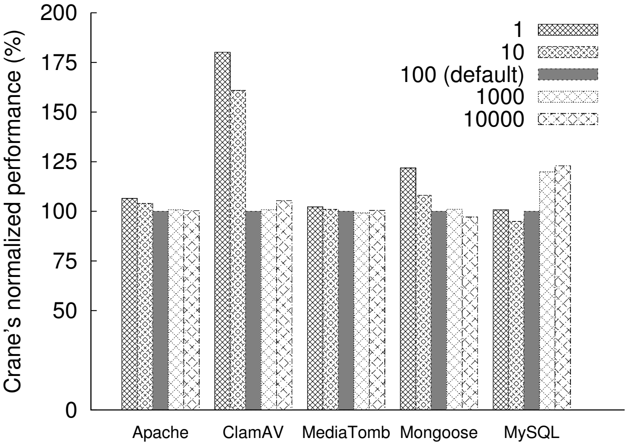

Figure 16: CRANE*'s performance with different settings on* $W_{timeout}$ *(us).* Normalized with the default parameter.

7.5 Sensitivity of Time Bubble Parameters

The two parameters $W_{timeout}$ and N_{clock} for time bubbling have trade-off on performance. This trade-off also depends on each server program as well as its performance workload. A smaller $W_{timeout}$ means the DMT scheduler can wait less time and then proceed with granted logical clocks with inserted time bubbles, but it also means that more time bubbles and thus more PAXOS consensus are involved. A smaller value also means time bubbling runs similar to a per-request consensus approach. Figure 16 shows CRANE's performance by only adjusting this parameter. CRANE's default

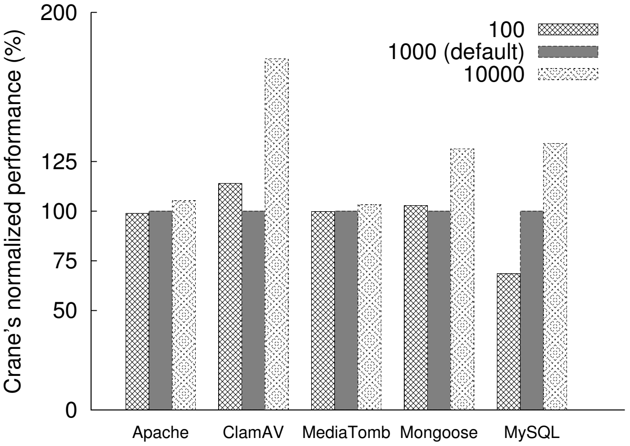

Figure 17: CRANE*'s performance with different settings on* N_{clock}. Normalized with the default parameter.

setting got the best result for both Apache and ClamAV, and it got the second best result for the other three programs.

The N_{clock} parameter also faces trade-off on performance. A smaller value means that servers can exhaust clocks in a time bubble sooner, but if a server does lots of Pthreads synchronizations to process a request, more time bubbles and thus more PAXOS consensus are involved. Figure 17 shows CRANE's performance by only adjusting this parameter. CRANE's default setting got the best result for ClamAV, MediaTomb, and Mongoose, and the second best result for the other two programs.

7.6 Checkpoint and Recovery

To handle replica failures, CRANE periodically invokes a checkpoint operation on one backup (§5.2). Each CRANE checkpoint operation contains four time consuming parts: (1) using CRIU to dump the state of a server process (and its child processes, if any); (2) stopping and restarting a LXC container; (3) doing an incremental checkpoint on a server's current working directory and installation directory between the LXC stop and start; and (4) restoring a process's state after the LXC restart.

Table 2 shows time costs for each process and file system checkpoint operation, and all are median values with 20 runs. In sum, a process checkpoint or restore took at most 415ms, and a file system checkpoint or restore took less than 7s except MySQL. MySQL took about one minute to checkpoint its file system because SysBench generated a large database in MySQL's installation directory. For each program, a file system restore operation took much less time than its checkpoint operation because a restore operation patches only files modified by the server program. A common LXC stop and restart operation took 2∼5s depending on the daemon processes' bootstrap progress within the container. Although each of these four steps in a CRANE checkpoint operation costs time, such a checkpoint is done on only one backup replica, its performance impact was negligible in our evaluation (the other replicas formed a quorum).

Program	C p (ms)	R p (ms)	C fs (ms)	R fs (ms)
Apache	33	48	3,069	237
ClamAV	415	353	6,963	6,128
MediaTomb	17	27	2,852	213
Mongoose	15	31	1,294	169
MySQL	88	81	53,473	712

Table 2: ***Average time cost for* CRANE*'s checkpoint and restoring component.* "C p" means "Checkpoint process", "R p" means "Restore process", "C fs" means "Checkpoint file system", and "R fs" means "Restore file system".**

To evaluate the speed of CRANE's PAXOS protocol on replica failure and recovery, we manually restarted the primary replica running a Mongoose server. The other two backups in the system then invoked a leader election with three steps (§5.1), which took 1.97ms. After the old primary's machine restarted, CRANE restarted the proxy and the consensus component, extracted the latest Mongoose checkpoint on the local machine and restored the Mongoose process and its file system. On the full restore of this CRANE instance, it received the new primary's heart beat message in 0.36s and downgraded itself to a backup. Overall, both the PAXOS leader election and the restarted old primary's self-downgrading took sub-seconds.

8. Related Work

State machine replication (SMR). SMR has been studied by the literature for decades, and it is recognized by both industry and academia as a powerful fault-tolerance technique in clouds and distributed systems [43, 63]. As a common practice, SMR uses PAXOS [42, 44, 65] and its popular engineering approaches [23, 52] as the consensus protocol to ensure that all replicas see the same input request sequence. Since consensus protocols are the core of SMR, a variety of study improve different aspects of consensus protocols, including performance [45, 55] and understandability [57]. Although CRANE's current implementation takes a popular engineering approach [52] for practicality, it can also leverage other consensus protocols and approaches.

At a system implementation level, SMR typically takes the "agree-execute" approach: replicas first "agree" on a total order of input request as a input sequence, and then "execute" the requests that have reached this consensus. Such typical systems include Chubby [21], ZooKeeper [6], and the Microsoft PAXOS [44] implementation, and they have been widely used to maintain critical distributed systems configurations (e.g., group leaders, distributed locks, and storage meta data). SMR has also been applied broadly to build various highly available services, including storage [19, 27, 61] and wide-area network [51]. Hypervisor-based Fault Tolerance [20] leverages a hypervisor to build a primary-back system for single-core machines. Unlike CRANE, these systems are not designed to transparently replicate general multithreaded server programs. Nevertheless, CRANE takes the typical "agree-execute" approach.

In order to support multi-threading in SMR, Eve [37] introduces a new "execute-verify" approach: it first executes a batch of requests speculatively, and then verifies whether these requests have conflicts that cause execution state divergence. If so, Eve rolls back the program to a state before executing these requests and re-execute these requests sequentially. Both Eve's execution divergence verification and rollbacks require developers to manually annotate all shared states, which is time consuming and error-prone.

Rex [33] addresses the thread interleaving divergence problem with a "execute-agree-follow" approach: it first records thread interleavings on the primary by executing requests, and then replays these interleavings on the other backups. If the executed interleavings in the primary may not be agreed on the other replicas, then Rex rollbacks the primary's states. These rollbacks/checkpoints also require developers' manual efforts for every program. Furthermore, Rex requires frequently shipping thread interleavings across replicas, which may be slow. Unlike CRANE's transparent checkpoint-restore mechanism, Rex requires program developers to implement the checkpoint-restore logic.

To improve performance, some SMR systems [6, 22, 25, 37, 38] perform read-only optimization on request interface and allow these requests to be processed rapidly without consensus. CRANE currently does not explore this direction mainly for two reasons. First, CRANE's performance overhead is already moderate in our evaluation. Second, some read requests may still modify programs' internal execution states (e.g., `Apache`'s internal HTTP cache) and affect outputs. Thus, ensuring whether a request is indeed read-only for a general server program may require understanding or crafting the program significantly, which may trade off transparency. However, exploring the trade-off between CRANE's transparency and performance is an interesting direction.

DMT and StableMT systems. In order to make multithreading easier to understand, test, analyze, and replicate, researchers have built two types of reliable multithreading systems: (1) stable multi-threading systems (or StableMT) [12, 16, 48] that aim to reduce the number of possible thread interleavings for program all inputs, and (2) deterministic multi-threading systems (or DMT) [13–15, 18, 31, 34, 56] that aim to reduce the number of possible thread interleavings on each program input. Typically, these systems use deterministic logical clocks instead of nondeterministic physical clocks to make sure inter-thread communications (e.g., `pthread_mutex_lock()` and accesses to global variables) can only happen at some specific logical clocks. Therefore, given the same or similar inputs, these systems can enforce the same thread interleavings and eventually the same executions. These systems have shown to greatly improve software reliability, including coverage of testing inputs [15] and speed of recording executions[14] for debugging.

Typical DMT systems, including Kendo [56], COREDET [13], and COREDET-related systems [14, 34], improve performance by balancing each thread's load with low-level instruction counts, so they are unstable to input perturbations. DDOS [34] demonstrates that a distributed system can be made deterministic. However, our CRANE approach is more flexible, because we can choose to focus on replicating servers' execution states only and discard clients' states, then CRANE has fewer scheduling constraints and can be more efficient.

Concurrency. CRANE are mutually beneficial with much prior work on concurrency error detection [32, 49, 50, 62, 69, 71], diagnosis [58, 59, 64], and correction [35, 36, 67, 68]. On one hand, these techniques can be deployed in CRANE's backups and help CRANE detect data races. On the other hand, CRANE's asynchronous replication architecture can mitigate the performance overhead of these powerful analyses [30].

9. Conclusion

We have presented CRANE, a SMR system that transparently replicates general server programs without requiring server developers' intervention. It provides a new state machine interface compatible to socket API, and it leverages deterministic multithreading to enforce the same schedules for a multithreaded server program across replicas. CRANE creates a time bubbling technique to efficiently enforce consistent logical times on admitting network requests across replicas.

Evaluation on five widely used server programs shows that CRANE is easy to use, has moderate overhead, and provides practical recovery support. CRANE has the potential to expand the adoption of SMR and to provide transparent fault-tolerance support for general server programs. CRANE's source code is at `github.com/columbia/crane`.

Acknowledgments

We thank Marcos K. Aguilera (our shepherd), Yinzhi Cao, Adrian Tang, David Williams-King, and anonymous reviewers for their many helpful comments. This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

References

[1] Boost C++ Libraries. `http://www.boost.org/`.

[2] LXC. `https://linuxcontainers.org/`.

[3] MySQL. `http://www.mysql.com/`, .

[4] MySQL Replication. `https://dev.mysql.com/doc/refman/5.0/en/replication.html`, .

[5] SQLite. `https://www.sqlite.org/`.

[6] ZooKeeper. `https://zookeeper.apache.org/`.

[7] SysBench: a system performance benchmark. http://sysbench.sourceforge.net, 2004.

[8] MediaTomb - Free UPnP MediaServer. http://mediatomb.cc/, 2014.

[9] Apache. Apache web server. http://www.apache.org, 2012.

[10] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, Oct. 2010.

[11] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW '10)*, Oct. 2010.

[12] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[13] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[14] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[15] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 677–692. ACM, 2013.

[16] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.

[17] Berkeley DB. http://www.sleepycat.com.

[18] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[19] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[20] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[21] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[22] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, Oct. 1999.

[23] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[24] Clam AntiVirus. http://www.clamav.net/.

[25] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009.

[26] concoord. Openreplica. http://openreplica.org/download/, 2015.

[27] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. Oct. 2012.

[28] criu. Criu. http://criu.org, 2015.

[29] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[30] H. Cui, R. Gu, C. Liu, and J. Yang. Repframe: An efficient and transparent framework for dynamic program analysis. In *Proceedings of 6th Asia-Pacific Workshop on Systems (APSys '15)*, July 2015.

[31] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[32] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[33] 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 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[34] N. Hunt, T. Bergan, , L. Ceze, and S. Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[35] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, 2012.

[36] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[37] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[38] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[39] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

[40] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 155–166, 2010.

[41] O. Laadan, N. Viennot, C. che Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[42] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[43] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[44] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[45] L. Lamport. Fast paxos. Fast Paxos, Aug. 2006.

[46] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, Mar. 2010.

[47] libevent. libevent. `libevent.org/`, 2015.

[48] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[49] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[50] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[51] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[52] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[53] mencoder. Mencoder. `https://www.mplayerhq.hu/`, 2015.

[54] Mongoose. `https://code.google.com/p/mongoose/`.

[55] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[56] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[57] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[58] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[59] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[60] M. Primi. LibPaxos. `http://libpaxos.sourceforge.net/`.

[61] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, Jan. 2011.

[62] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[63] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[64] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[65] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[66] VTune. `http://software.intel.com/en-us/intel-vtune-amplifier-xe/`.

[67] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[68] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[69] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, Oct. 2005.

[70] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*, Oct. 2011.

[71] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.