Transparent State Machine Replication for General Programs

Authors

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

We have built 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 PARROT, a deterministic multithreading system we built, 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 has the potential to grealty improve the reliability and availability of general programs. CRANE's source code is at `github.com/columbia/crane`.

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 [36, 34, 46]) 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 [18, 41, 16, 45, 15, 40, 29, 27].

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 [13, 29, 28, 27]. To tackle this problem, one prior approach, execute-verify [29], 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 [5], 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 [36, 34, 46, 18, 41] are needed to explain PAXOS), under-specified [41] 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 [16] 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 [16], causing the Chubby developers to add quota support.

We present 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 [41] to ensure that a quorum of the replicas sees the same exact sequence of the incoming socket calls.

To address nondeterminism, we have built three *deterministic multithreading (DMT)* systems [25, 24, 23]. DMT [14, 26, 44, 11, 13, 28, 12] 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 PARROT [23], our latest DMT system. PARROT incurs merely 12.7% overhead in average 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.

Two prior approaches attempted to tackle this challenge. Execute-agree-follow [27] 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 [13] 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 §3.3.) 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 [38] 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 [20] to checkpoint and restore process states, and LXC [1] for file

[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 [35].

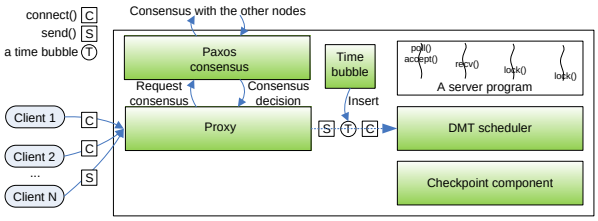

Figure 1: *The* CRANE *Architecture.* CRANE components are shaded (and in green).

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.

We evaluated CRANE on five widely used server programs, including HTTP servers `Apache` and `Mongoose`, an anti-virus server `ClamAV`, a uPnP multimedia server `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 [23] 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.

In the remainder of this report, §2 introduces CRANE's architecture and assumptions. §3 presents evaluation results. §4 concludes, and §5 presents recommendations.

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 [41] 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.

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 [41]. 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 [23] 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 §2.2). 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. More discussions on the values of the two parameters $W_{timeout}$ and N_{clock} are given in §3.5.

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.

2.2 Assumptions

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 [21], 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 [32, 37] or OS-level techniques [13], 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 (§3.2).

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 [13] and DDOS [28] have many effective techniques for tackling this problem, which CRANE can leverage.

3 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` [8] and `Mongoose` [43]; `ClamAV` [19], an anti-virus scanning server that scans files in parallel and deletes malicious ones; `MediaTomb` [7], a uPnP multimedia server that uploads, shares, and transcodes pictures and videos in parallel; and `MySQL` [2], an SQL database. Although `MySQL` has a replication feature [3], 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 [4] 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` [42] to transcode a 15MB video from AVI to MP4. For `MySQL`, we used `SysBench` [6] 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 (§3.5).

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:

§3.1: Is CRANE easy to use?

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

§3.3: What is CRANE's performance overhead compared to nondeterministic executions?

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

§3.5: How sensitive are the two time bubbling parameters to CRANE's performance?

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

3.1 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 [23] to line up parallel computations as much as possible and compute efficient DMT schedules (cf §3.4).

3.2 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,

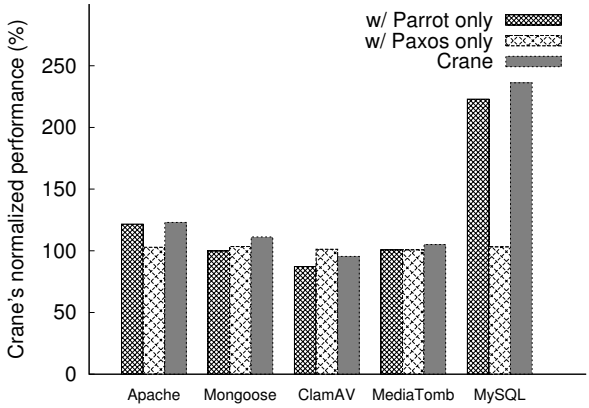

Figure 2: CRANE*'s performance normalized to un-replicated nondeterministic execution.*

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.

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.

3.3 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 2 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 [23]. 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` [47] profiling tool. When running in CRANE, `MediaTomb` only made 6.6K synchronization context switches, while in the Pthreads runtime it made 0.9M synchro-

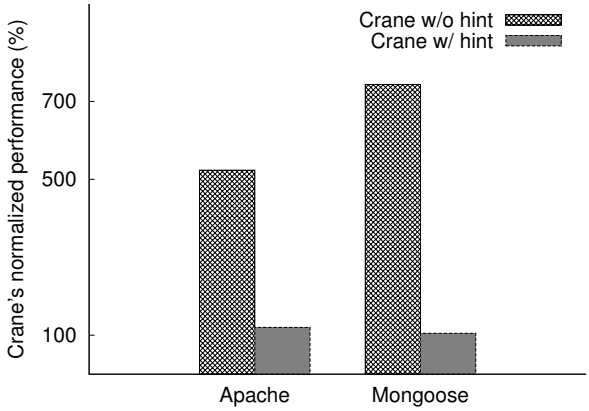

Figure 3: Effects of PARROT's soft barrier performance hints.

nization context switches. This saving caused `MediaTomb` running with PARROT a 12.76% speedup compared to its nondeterministic execution. The PARROT evaluation [23] 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.

3.4 Optimization of PARROT's Performance Hints

In general, a DMT schedule may be slow in some cases [23, 39], 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 [23] 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 3. These hints reduces `Apache`'s overhead from a 424% to 22.99%, and `Mongoose`'s from a 643% to 5.09%.

3.5 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

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.***

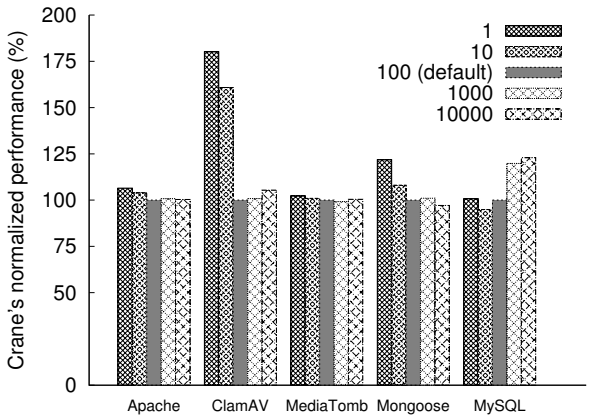

Figure 4: CRANE*'s performance with different settings on $W_{timeout}$ (us).* Normalized with the default parameter.

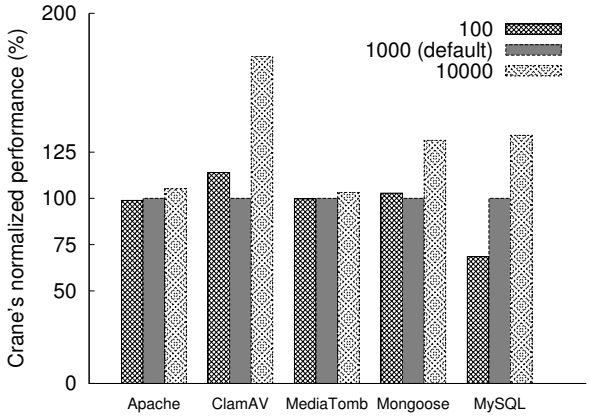

Figure 5: CRANE*'s performance with different settings on N_{clock}.* Normalized with the default parameter.

per-request consensus approach. Figure 4 shows CRANE's performance by only adjusting this parameter. CRANE's default 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 5 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.

3.6 Checkpoint and Recovery

To handle replica failures, CRANE periodically invokes a checkpoint operation on one backup. 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

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".

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).

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 [41], 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.

4 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`.

5 Recommendation

We envision three applications for CRANE. First, CRANE can be leveraged by other replication concepts (e.g., byzantine fault tolerance [17, 30]) and record-replay [33, 31, 37] because they also suffer from nondeterminism. Second, promising results in REPFRAME [21] 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 [9, 50, 10].

Moreover, our systems, techniques, and tools developed for CRANE has broad applications on improving software reliability, including bypassing concurrency bugs [48], detecting security rule violations [22], and improving precision of static analysis [49].

References

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

[2] MySQL. `http://www.mysql.com/`.

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

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

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] `http://www.clamav.net/`.

[20] Criu. `http://criu.org`, 2015.

[21] 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.

[22] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[23] 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.

[24] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[25] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

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

[37] 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.

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

[39] 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.

[40] 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.

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

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

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

[44] 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.

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

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

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

[48] 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.

[49] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 205–216, June 2012.

[50] 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.