DeterV: Verifying Deterministic Parallel Execution

Zhengqing Liu
scofield.liu@imperial.ac.uk
Imperial College London
London, United Kingdom

Marios Kogias
m.kogias@imperial.ac.uk
Imperial College London
London, United Kingdom

Abstract

Deterministic execution is a critical component for replicated systems, where each replica is required to advance its state deterministically in an agreed-upon order even in the presence of concurrent execution. Despite the extensive efforts on the design space, the correctness of deterministic parallel execution engines is not yet backed by provable guarantees. We close this gap with DeterV, a deterministic parallel runtime which uses Verus to verify its correctness and equivalence to serial execution in the input order. DeterV establishes determinism via per-object local reasoning, an effect-boundary contract which limits the determinism enforcement to a single point, and a series of refinements that compose these local guarantees globally. DeterV attains a 3:1 proof-to-code ratio and incurs only trivial runtime overhead compared to an unverified baseline. Despite being a single system implementation, we discuss how DeterV's ideas can generalize to other usecases and designs.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Computing methodologies →** *Parallel computing methodologies*; • **Theory of computation →** *Concurrency*.

Keywords: Determinism, Replicated State Machines, Formal Verification, Verus, Multicore Systems

ACM Reference Format:
Zhengqing Liu and Marios Kogias. 2026. DeterV: Verifying Deterministic Parallel Execution. In *Workshop on Programming Languages and Operating Systems (PLOS '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831586.3838154

1 Introduction

Deterministic systems, i.e., systems that given the same input always produce the same output, have been widely studied by multiple communities, such as databases [18, 42], operating systems [3, 16], programming languages [24], and distributed systems [20] for their properties. Determinism simplifies building replicated fault-tolerant systems, offers reproducibility, enables easier debugging, and can be used to reduce the overhead in checkpoint/restore systems.

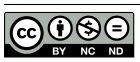
Interest in deterministic parallel execution has been recently revived in the scope of high-throughput payment systems, blockchains, and smart contract execution. These systems typically decompose into a sequencing/consensus layer that produces an agreed-upon total order (a log) and a replicated execution layer that runs the log operations. A naive approach that could guarantee that all replicas end up in the same state is single-threaded execution, assuming that there are no other sources of non-determinism, e.g., random number generators and external events, beyond thread interleavings. However, as sequencing layers have become increasingly high-throughput and low-latency [1, 2, 4, 27], sequential execution can become the system bottleneck. Thus, the need for parallel execution and specifically deterministic parallel execution becomes a timely requirement.

Unfortunately, existing deterministic parallel systems only claim determinism by construction with their correctness guarantees usually justified through informal arguments. If the runtime contains subtle scheduling, validation, or commit-order bugs, the system may silently violate the serial semantics it claims to provide. Beyond causing replica divergence and undermining reproducibility and debuggability, violating the log order in distributed ledgers can allow malicious parties to increase their profits [45, 46].

Program verification offers a promising path to close this gap through machine-checked proofs of systems-level correctness and provide provably deterministic parallel systems that can work under high-throughput and low-latency requirements. Recent advances in practical verification tools (e.g., Verus [28] and Dafny [29]) have substantially lowered the engineering cost of such efforts, with several companies [6, 8] adopting (lightweight) formal methods for their production deployments.

In this work, we aim to build and verify a deterministic parallel execution engine. The main challenge in such an effort is that parallel execution can lead to a very large number of possible thread interleavings, which makes it hard to reason about the global behavior of concurrently executing tasks or transactions [10]. A proof that reasons directly about these global schedules does not scale, so we need a way to establish determinism without enumerating them.

Our first insight to address this challenge is to reason locally for each transaction and object rather than globally. Instead of proving correctness over the full space of thread



PLOS '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2877-8/26/09
https://doi.org/10.1145/3831586.3838154

schedules, we show how to localize correctness properties to individual objects and then build up towards the full system through a series of refinements that prove equivalence to serial execution according to the transaction log order. Each object advances via a version sequence induced by the input log, so the global ordering requirement becomes object-local ordering constraints, and these per-object guarantees compose into whole-execution determinism. Our second insight is that through careful design and state separation, we can concentrate the determinism enforcement to a single point, when a task makes its effects visible to the rest of the system. By giving this boundary a Hoare-style contract that advances the version of each touched object, we can aggressively simplify the complexity and extent of proofs.

We realize these insights in DeterV, a deterministic parallel runtime in Verus [28] that we verify by adding proof machinery to a concrete unverified system one step at a time. We represent each object's version as ghost state backed by the authoritative resource algebra of Iris [25], and attach it to a single `commit_effects` operation. The refinement then composes the local commits into global determinism without reasoning about thread interleavings directly (§4.7). DeterV reports a 3:1 proof-to-code ratio and marginal overheads (§5) compared to an unverified baseline. Finally, we discuss how DeterV's approach can generalize to other implementations and designs of similar deterministic execution engines.

2 Background and Motivation

This section revisits the design of deterministic parallel systems and existing verified concurrent systems.

2.1 Deterministic Parallel Execution

In this work, we focus on deterministic parallel systems that take an ordered log as the input, perform parallel execution that guarantees that the resulting effects are identical to those of sequential execution. To enforce the input order, existing systems fall into two main categories: pessimistic and optimistic. In pessimistic designs, i.e., *schedule-execute*, the runtime schedules tasks via dependency analysis or ordered locking, then execution runs concurrently when dependencies are satisfied [33, 42]. In optimistic ones, i.e., *execute-validate-commit*, the runtime speculatively performs parallel execution, then sequentially validates if effects can be committed [12, 19]. Pessimistic approaches rely on the predefined read/write sets, thus as long as execution proceeds, it will commit; however, optimistic ones bypass the programming model assumption, while introducing aborts and re-execution in conflicts. Consequently, optimistic approaches perform well under low contention but suffer high abort rates that degrade performance when many accesses overlap, whereas pessimistic ones avoid aborts and perform better under high contention.

2.2 Verified Concurrent Systems

A line of work applies machine-checked verification to concurrent systems. IronFleet [22] blends TLA-style state machine refinement with Hoare-logic verification. Its successor IronSync [21], built on a linear extension of Dafny [30], verifies concurrent libraries by sharding the state machine into independently owned pieces whose local transitions compose into a whole-system guarantee. More recently, Verus [28] provides a practical verification framework for systems in Rust, which also provides VerusSync that leverages the resource algebra and transition specification to automate multi-thread reasoning. In the setting of transactional systems, vMVCC [11] verifies a multi-version concurrency-control library via prophecy variables given the linearization point happens before the transaction run body, while DaisyNFS [10] verifies a concurrent, crash-safe file system whose operations are atomic and refine a sequential specification. CSPEC [9] takes a complementary route, using mover types and the commutativity of operations to reorder interleavings so that proving a concurrent procedure correct reduces to reasoning about its atomic, sequential execution. Armada [35] lowers proof effort for concurrent programs by machine-generating refinement proofs via reduction, rely-guarantee, and TSO-elimination reasoning. Their correctness targets, however, are the linearizability or serializability of concurrent transactions, whereas DeterV targets determinism, namely that a parallel execution reproduces the exact outcome of a given serial order, a property these systems do not establish.

2.3 Formal Verification of Determinism

Determinism has been formalized at the *model* and *language* level: Kahn networks [26] and the synchronous languages descending from them admit determinacy proofs [39], with Vélus [7] verifying a Lustre compiler that emits sequential code, and the concurrent revisions model carrying a mechanized determinacy proof in Isabelle/HOL [38]. Lingua Franca (LF) [34] achieves deterministic concurrency through its underlying reactor model, whose statically declared reaction dependencies and logical-time semantics fix the observable behavior regardless of physical scheduling. Verification around it, however, stays at the model level: the reactor semantics and its determinism are mechanized in Lean [40], and individual LF programs are model-checked against models derived automatically from their implementations [31], while the runtime enforcing the model remains unverified. Closer to programs, a type system soundly verifies determinism of sequential code at compile time [37], and Musketeer [36] targets internal determinism [5] via program logics, whereas DeterV targets the *runtime implementation* level, orthogonal to the prior work.

3 Insights

In this section, we highlight key observations and insights that guide the design of DeterV.

3.1 Localize by Object, then Compose (I1)

Existing deterministic runtimes target the non-determinism arising from interleaved access to shared mutable state. [1] The input log presents the total order of tasks/transactions, while the deterministic runtime derives the partial order to exploit parallelism while preserving determinism. This partial order exists because tasks share or overlap on objects: two tasks must respect the log order only if they touch a common object. For example, given an ordered log $\langle T_1{:}\,w(x),\ \ T_2{:}\,w(x),\ \ T_3{:}\,w(y)\rangle$, T_2 must follow T_1 because both write x, whereas T_3 is independent and may execute in parallel with either.

This observation suggests a way to avoid reasoning directly about the full space of thread interleavings (C1). Instead of proving correctness from the global task view, we reason from the local object view. For each object, the projection of the input log onto tasks that access that object induces a sequential order of relevant accesses. Local correctness then requires the runtime to respect this per-object order, rather than the entire global task order.

DeterV makes this object view explicit by treating each object as advancing through its version sequence $v_0 \longrightarrow v_1 \longrightarrow \cdots$ induced by the input log. Each write to an object produces a new version which corresponds to the next step in that object's version sequence. This turns a global ordering requirement into many object-local ordering constraints: ensuring each object progresses via its version sequence.

Given the object-local guarantees, we can compose the global determinism, i.e., determinism of the whole execution follows from determinism of these per-object sequences. If every object observes the same sequence of updates and reaches the same final state as in the sequential reference execution, then the composed shared state is also the same as the sequential state. Thus, DeterV reduces a global reasoning problem over many possible thread interleavings to local reasoning over individual objects, followed by a composition step over all objects.

3.2 Verify at the Effect Boundary (I2)

Deterministic parallel runtimes diverge in design choices, pessimistic or optimistic (§2.1), but converge structurally: in both schemes, every task is gated by a dependency-resolution mechanism that must be satisfied before the task's effects become visible to others. Equivalently, each task has an effect boundary: the moment at which the runtime commits its externally visible writes on shared state and unblocks successor tasks. This boundary provides a natural linearization point for verification. Although runtimes differ in how they enforce dependencies, they ultimately publish effects by exposing a common commit interface, such as `commit_effects`, whose arguments include the task's write set.

We identify that this interface is where we can unify various design choices (C2). DeterV turns this interface into the verification hook via a Hoare-style contract [23], when we treat this effect boundary as the only operation allowed to publish state changes. The precondition encodes exactly the ordering facts that must already hold, such as for about-to-commit objects, this commit should only be allowed to make if the previous versions match. The postcondition then advances each touched object to its next version, simultaneously discharging the dependency slots of successor tasks and establishing their preconditions.

This verified interface isolates the proof of determinism from the runtime's internal design choices. Pessimistic runtimes may establish the precondition by construction before execution, while optimistic runtimes may establish it through validation after speculative execution. Once the precondition holds, both designs use the same verified boundary to publish effects.

4 Verifying a Deterministic Runtime

We show how to build a verified deterministic parallel runtime by starting from a concrete unverified system and adding verification machinery one step at a time. We choose this as it is a Rust implementation inspired by DORADD [33], the state-of-the-art deterministic parallel runtime which has high throughput and low latency without batching, and it has been used to build execution engines for smart contracts [32].

4.1 The Unverified Runtime

The runtime achieves deterministic parallel execution through a single dispatcher and a pool of worker threads. The dispatcher processes the input log in reference order, derives each task's read/write object sets, assigns object versions, and spawns the task. Once spawned, a worker runs the task concurrently: it waits for its dependency predecessors to finish (i.e., wait for certain object versions to be ready to consume), executes the task body, applies its writes to the store, and signals its successors. Because the dispatcher is the only component that consumes the input log and assigns versions, it serializes the reference order and thereby enforces deterministic outcomes.

```
// Dispatcher: runs sequentially given the log order
while let Some(&mut task) = incoming.recv() {
  assign_versions(task);
  let (pre_deps, post_deps) = resolve_deps(task);

  spawn_async(move || {
    // worker: runs concurrently on the thread pool
    pre_deps.await();  // 1. wait for predecessors
    task.run();        // 2. execute task body
```

[1] Other sources of non-determinism, such as timers and randomness, are orthogonal and outside the scope of this project.

```
    post_deps.signal();// 3. unblock successors
  });
}
```

Listing 1. Main structure of the unverified runtime.

The runtime implements the task dependency DAG using `tokio::sync::Notify` [13], a lightweight synchronization primitive that carries no payload and serves purely for readiness signaling. Concretely, the runtime maintains one `Notify` instance per object version. When a task arrives with its required versions explicitly annotated, the runtime spawns an asynchronous task that (1) `await`s on the `Notify` instances for all required input versions; (2) once these dependencies are satisfied, executes to completion on the thread pool; and (3) signals the `Notify` instances for the versions it produces, unblocking dependent tasks. Since each version is consumed exactly once, the runtime garbage-collects both the `Notify` entry and the object version immediately after consumption. Each `Notify` object has a unique name within the worker, and the runtime creates it atomically on the first encounter with either the producer or the consumer task. This design folds dependency enforcement directly into the asynchronous runtime, yielding deterministic parallel execution without invasive changes to the underlying scheduler (i.e., Tokio [41]).

4.2 Verification Target and Specification

Before verifying anything, we must formally specify what the runtime should do. The runtime's promise is that parallel execution is equivalent to serial execution that follows the log order, so we make that promise concrete with a reference spec model: `sequential_exec`(log) runs the log one task at a time, in order, and produces a final state. The property we want, *global determinism*, is then simply that the store state the runtime produces, under any schedule, equals the state `sequential_exec`(log) produces. We assume each task is a deterministic function of the values it reads, with no other sources of non-determinism.

4.3 A Per-Object View of Correctness

Proving global determinism directly is hard. With many workers committing concurrently, there is a very large number of possible interleavings, and a proof that reasons directly about global thread schedules does not scale (C1).

DeterV tackles this by leveraging per-object local reasoning (I1), zooming-into the object view to reason about correctness. The runtime already expresses dependencies as object versions: a task waits only on the specific versions it reads. For a single object, the projection of the log onto tasks that write that object is a sequence: version v_0, then v_1, then v_2, and so on. Correctness for that object means the runtime advances it along exactly this sequence: every committed write moves the object to the next version in log order, never skipping or reordering. If every object individually walks its own sequence, the global store is correct by composition, which we discharge in §4.7. This reframes the goal. Instead of one global proof over interleavings, we owe one local proof per object: each object's committed version always matches the version the sequential reference would have produced. The remaining steps build the machinery that makes this local reasoning sound under concurrency.

4.4 Ghost State for Object Versions

To reason per object, a worker needs to talk about "object x is at version v." Under concurrency this is not a fact a worker can simply assert and rely on: another worker committing to x could change its version between the moment our worker checks it and the moment it acts. A plain shared variable, read and then used, gives an unsound proof here. We need to carefully design the ghost state, i.e., proof-only state erased at compile time, with three properties. A worker's claim about an object's version must be (i) owned by that worker, so it can be used in step-by-step reasoning; (ii) consistent with a single global truth about that object; and (iii) stable, meaning no other worker can silently invalidate it.

DeterV obtains all three from the authoritative resource algebra of Iris [25]. Each object x gets a single ghost version cell, conceptually Iris's `Auth(Ex(Version))`, where `Ex(·)` is the exclusive resource algebra that admits no valid second copy. This cell splits into two complementary halves, realized in Verus by the library pair `GhostVarAuth<Version>` and `GhostVar<Version>`. The authority $\bullet v_x$ (the `GhostVarAuth`) is the unique global record of x's current version number, one per object, incremented by one on each write, and it lives inside a per-object invariant I_x. Any worker thread may open I_x, but only under the discipline it enforces: $\bullet v_x$ always agrees with x's current value in the store, and the version may be advanced only by a worker that opens I_x while holding the matching fragment. The fragment $\circ v_x$ (the `GhostVar`) is that worker's local/fragmental owned view of the version. Because it is a non-`Copy` linear value, its exclusivity, and hence the stability we needed, comes from Verus's type system rather than runtime bookkeeping, so no other worker can hold a conflicting claim.

A potential alternative is to pass directly each object's version as a single exclusive resource with no invariants. However, this makes our theorem difficult to prove, which concerns the shared store state: the relation between ghost version and physical state must hold at every instant from every thread's view, and an owned token in transit is invisible to everyone else. The authority inside the invariant provides this always defined record, and collectively the authorities form the global state that the refinement in §4.7 requires.

4.5 Encode Ghost State into the Runtime

The ghost state says how versions are represented; we still need a place in the runtime where versions are checked and advanced. In the runtime, the moment that matters is when

a task publishes its writes, making the effects visible, the only point where shared state changes and successors are unblocked. DeterV encodes verification logic around one operation, `commit_effects` by giving it a Hoare-style contract: a precondition that must hold on entry and a postcondition guaranteed before return. This already presents natively in Line 9 of Listing 1, in which the task performs computation, produces the effects, then commits to the shared object store.

```
exec fn commit_effects<S: Store>(
  store:            &StoreHandle<S, ReadWrite>,
  writes:           &Vec<(ObjId, S::Value)>,
  Tracked(inv):     Tracked<ObjInvariantsList>,
  Tracked(permits): Tracked<PermitsList>,
) → (advanced: Tracked<PermitsList>)
requires
  forall|o| permits_match_inv(permits[o], inv[o]),
ensures
  forall|o| advanced@[o].v == permits[o].v + 1, {
  let applied: Map<ObjId, Version> = store.apply(writes);
  commit_effects_ghost(inv, permits, Ghost(applied))
}
```

Listing 2. `commit_effects` API in Verus.

The API separates concrete runtime state from ghost state. Only `store` and `writes` are real physical values: `store` exposes read/write access to the shared object store, and `writes` is the write set the task produced when it finished executing. The `Tracked` arguments are ghost-only: `inv` carries the handles to *shared* per-object invariants guarding the authorities, and `permits` carries the fragments *owned* by the task. We elaborate how fragments get traversed in §4.6.

The contract states the obligation directly. The precondition, `permits_match_inv`, requires that every fragment the task holds agrees with the authority inside that object's invariant, equivalently, that for each written object $\bullet v_x$ and $\circ v_x$ are equal. The postcondition states that each written object's authoritative version becomes exactly its fragment's version plus one. The body performs the physical write via `store.apply`, then calls `commit_effects_ghost`. In this ghost fn, it opens each invariant, proves the agreement, and advances both $\bullet v_x$ and $\circ v_x$ to v_{x+1}. In addition, it also checks whether the physical state aligns with the ghost state update via the applied versions returned from the physical writes.

Two points are worth noting. First, the version match is enforced by the resource algebra, not by ad-hoc runtime bookkeeping: agreement is forced the instant the invariant is opened with the fragment in hand. Second, DeterV opens the invariants for the whole write set together, so the $|W|$ per-object updates of a single task form one atomic ghost transition, and no other worker can observe a task whose write set is only partially committed.

4.6 Fragment Construction and Lifecycle

The `commit_effects` contract assumes the committing task already holds the right fragment. Recall that the authority $\bullet v_x$ is the single source of truth for x's version, kept inside x's invariant, while the fragment $\circ v_x$ is a task's transferable claim that x stands at a given version. Although the dispatcher fixes every object version in advance, the fragment for version v exists only after the authority has advanced to v. The authority advances only at the commit point, after its current fragment matches the authority, and that match at version v is what allows advancing to v+1 and producing the fragment for v+1. Hence, pre-generating these fragments for known versions is not plausible.

Moving fragments is the difficulty. Ownership of the fragment must therefore pass from the producing task to the consuming task, which might run on separate threads and are connected only by the dependency edge between them. The transfer must occur exactly once along that edge: neither copied nor observed by a third task. Simply wrapping the fragment via a lock or shared cell does not fit this pattern, since those primitives provide shared, repeated access under mutual exclusion, whereas a fragment needs a single transfer of sole ownership tied to the readiness signal already exchanged between dependent tasks.

To tackle this, we leverage the `tokenized_state_machine` in VerusSync to specify the transfer of fragments and its `storage` notion [43] to hold linear fragments. For each object version we define two operations: the producer may *deposit* the fragment once, and the consumer may *withdraw* it once. We model it as a state machine with states `empty`, `deposited`, and `withdrawn`, traversed once and in order, and we grant the producer only a deposit capability and the consumer only a withdraw capability. The dispatcher distributes these capabilities, since it already knows which task produces and which consumes each version. Because `withdrawn` is reachable only through `deposited`, a successful withdraw is itself evidence that the deposit occurred.

Concretely, we manage the fragment passing via a *slot*: a mailbox for a single (*object*, *version*) pair. The producer places its fragment in the slot when it commits, and the consumer removes it when ready to commit in turn. The slot reuses the per-version `Notify` the runtime already employs for readiness (§4.1): the deposit travels with the signal the producer sends, and the withdraw happens when the consumer's wait returns, so the existing happens-before edge orders the two steps and the consumer never withdraws from an empty slot. The dispatcher allocates one slot per version a task and hands out the matching deposit and withdraw capabilities, after which each worker withdraws its predecessors' fragments when their signals arrive and deposits its advanced fragments before signaling.

```
transition!{
  deposit(frag: Frag) {
```

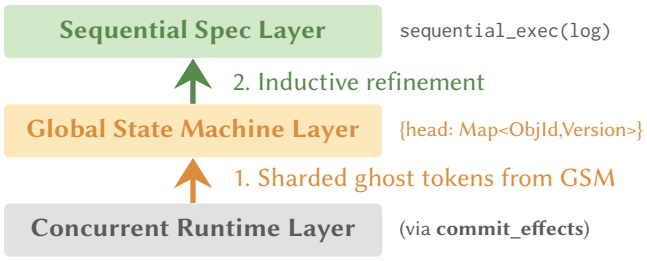


Figure 1. DeterV proof architecture.

```
    require(pre.phase == Phase::Empty);
    remove deposit_cap -= Some(());
    update phase = Phase::Deposited;
    deposit frag += Some(frag);
  }
}
```

Listing 3. Fragment `deposit` pseudocode in VerusSync.

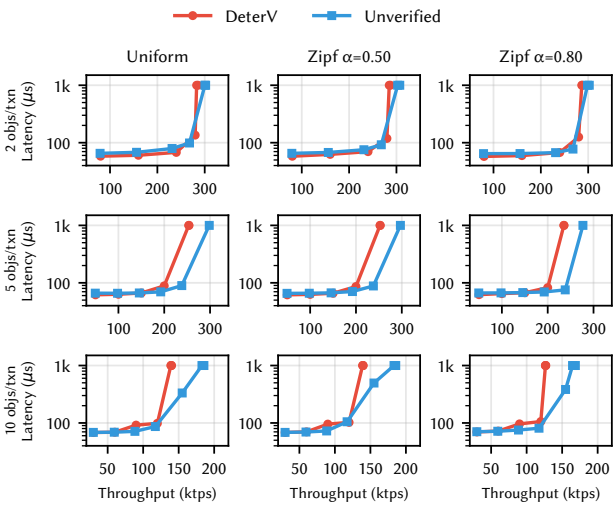


Figure 2. DeterV performance comparison with its unverified version.

4.7 Global Refinement

We now turn the local commits of each worker into one global guarantee: under any schedule, the final store equals the store produced by `sequential_exec(log)`. Following the methodology of IronSync [21], we phrase this as a refinement through three layers (Figure 1): a concurrent runtime, a global state machine (GSM), and a sequential specification. This has a similar shape as the node replication example (proves the linearizability via `UnboundedLog` and `SimpleLog`) in IronSync, specialized to deterministic execution.

The sequential spec layer. The top layer is the trusted reference model `sequential_exec(log)`: its state is a store and a log position, and its only step applies the task at that position to the store and advances by one. The step is deterministic because each task is a deterministic function of the values it reads, and the reference outcome is the store reached once the position passes the end of the log.

The GSM layer. The middle layer is a state machine whose state is one version counter per object, i.e., `head`. It carries the shared ghost tokens, i.e., invariants maintaining the authorities sharded across objects. The map `head`, which bookkeeps each object to its committed version, is just a read projection of the authorities. Its single step, `advance_head`, takes a committing task and bumps the version of every object in its write set by one, in one atomic step.

From the runtime to the GSM. This is similar to how IronSync connects the local-transition systems to the global one. The concurrent runtime carries the GSM's ghost token in pieces, run `commit_effects` with the ghost per-object invariants and advances authorities. A fragment is a worker's owned claim that one object sits at a given version, and it must agree with that object's authority. A worker reaches `commit_effects` holding the fragments for its whole write set, and presenting them is what lets it open the matching invariants and run one `advance_head` step. Because all the relevant invariants open together, a task's per-object updates commit as one step, and no worker ever observes a half-committed write set. The atomicity of `commit_effects` eases the refinement later to the spec, since a run's commit trace is already a serialization of tasks, whereas non-atomic one splits the tasks into interleaved substeps which demands further proof work for refinement.

From the GSM to the spec. We show the GSM connects to the specification by refinement and induction. The abstraction reads each object's committed value off the GSM and assembles the store, related to the specification one object at a time. We order a run's commits into a trace and induct over the log order, so each `advance_head` step moves a written object one version along its writers. This holds because each fragment is held by one task at a time, so the check `permit.v = head(o)` stays stable under parallel commits. Projecting pointwise composes these equalities, so every schedule ends in the store of `sequential_exec(log)`: global determinism, no cross-object obligation.

5 Evaluation & Discussion

In this section, we report our experience verifying DeterV using Verus, measure its runtime overhead against the unverified baseline, and discuss how its approach generalizes to other deterministic designs.

5.1 Verification Experience

Porting Experience. Our unverified runtime is a heavily asynchronous Rust system: it builds on `Tokio` for `async`

Table 1. A summary of lines of code.

Metric	Unverified runtime	DeterV proof	DeterV exec	DeterV trusted	DeterV total
LOC	371	1,467	469	374	2,310

scheduling, uses `Notify` to wake tasks waiting on dependencies, and stores dependencies in `DashMap` [17], a popular concurrent hashmap that lets many threads read and write shared state without a global lock. None of these are supported or easily expressible in `Verus`[2] standard library. Porting therefore means rebuilding on a synchronous thread pool with refactoring those unsupported components. The genuinely hard part was not proving the logic correct but reshaping the code to satisfy the proof. DeterV realizes in the 3:1 proof-to-code ratio and the verification runs in 4.5 seconds.

Trusted Scope. DeterV treats the following components as trusted and leaves them unverified: `crossbeam-channel` [15] and `std::thread` for the thread pool, `Mutex`/`Condvar` for one-shot blocking signals, and a widely used concurrent hash map for dependency management. These components are either widely adopted or part of Rust's standard library, and they are not the primary correctness target of DeterV. One of the key limitations of DeterV is that it verifies only safety, not liveness.

5.2 Performance

We compare the performance of DeterV with its unverified version via a synthetic YCSB microbenchmark [14]. Each transaction contains a fixed number of read-spin-write operations, each on a unique key. We use a 10M keyspace, and vary the number of keys accessed per transaction under both uniform and zipfian distributions. Each transaction spins 50 μs to emulate application execution. We run the experiments on an AWS `m5d.8xlarge` instance with 16 physical CPU cores and 128GB memory.

Figure 2 shows that DeterV incurs no latency overhead across all settings. For throughput, it has negligible overhead for transactions that access two objects. As the number of objects per transaction increases, the throughput overhead increases. For example, the maximum throughput drops by 24% when each transaction accesses 10 objects. The major overhead comes from replacing the unverified runtime's `Notify` with heavier `Mutex` and `CondVar` blocking primitives for one-shot signaling. Thus, the throughput gap widens as each transaction accesses more objects.

5.3 Generalizability

We believe DeterV's verified core, the versioned ghost state, the `commit_effects` boundary and the refinement above it, can be reused across deterministic designs. A pessimistic runtime would establish it by construction. For an optimistic runtime, which executes first then sequentially check and validate, we can still use the same API (I2) to check the version matching, thus reusing the local reasoning (I1). We have not yet implemented the optimistic path, which we leave to future work.

[2]Verus only recently supports async primitives [44].

6 Conclusion & Future Work

We present DeterV, the first formally-verified deterministic parallel system. It localizes correctness to per-object version sequences and verifies a single effect boundary, then composes these local guarantees into global determinism via refinement. In the future, we plan to verify an optimistic deterministic parallel system, measuring the additional effort it demands beyond DeterV's observation and how much of our work can be reused and unified. We also hope to extend this approach beyond the single-node multicore setting to a distributed one.

Acknowledgments

We sincerely thank anonymous reviewers, Azalea Raad, Sacha-Élie Ayoun, and Opale Sjöstedt for their feedback. This work is supported by a Sui Academic Research Award.

References

[1] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 599–616.

[2] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. ubft: Microsecond-scale bft using disaggregated memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 862–877.

[3] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-Enforced Deterministic Parallelism. In *OSDI*. USENIX Association, 193–206.

[4] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2025. Mysticeti: Reaching the Latency Limits with Uncertified DAGs. In *32nd Annual Network and Distributed System Security Symposium (NDSS 2025)*. Internet Society, San Diego, CA, USA, 1–18. doi:10.14722/ndss.2025.240929

[5] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 181–192.

[6] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 836–850.

[7] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 586–601. doi:10.1145/3062341.3062358

[8] Claudia Cauli, Timo Lang, Shuo Chen, Sebti Mouelhi, Xin Jin, Subhajit Bandopadhyay, Xusheng Chen, Yazhi Feng, Haoze Song, Linhua Tang, et al. 2026. Lessons Learned from Incorporating Formal Methods in Huawei Cloud Reliability. In *Proceedings of the 21st European Conference on Computer Systems*. 2223–2238.

[9] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying Concurrent Software Using Movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 306–322.

[10] Tej Chajed, Joseph Tassarotti, Mark Theng, M Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 447–463.

[11] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. 871–886.

[12] Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2541–2554.

[13] Tokio Contributors. [n. d.]. Notify in tokio::sync. Rust crate documentation on docs.rs. https://docs.rs/tokio/latest/tokio/sync/struct.Notify.html Accessed: 2025-12-31.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[15] Crossbeam Contributors. [n. d.]. `crossbeam-channel`: Multi-Producer Multi-Consumer Channels for Message Passing. Rust crate documentation on docs.rs. https://docs.rs/crossbeam-channel Accessed: 2026-06-03.

[16] Heming Cui, Jirí Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *SOSP*. ACM, 388–405.

[17] DashMap Contributors. [n. d.]. `dashmap`: A Blazingly Fast Concurrent Map in Rust. Rust crate documentation on docs.rs. https://docs.rs/dashmap Accessed: 2026-06-03.

[18] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201. doi:10.14778/2809974.2809981

[19] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *PPoPP*. ACM, 232–244.

[20] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. 2014. Rex: replication at the speed of multi-core. In *EuroSys*. ACM, 11:1–11:14.

[21] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 911–929.

[22] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 1–17. doi:10.1145/2815400.2815428

[23] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[24] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *OOPSLA*. ACM, 97–116.

[25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* 50, 1 (2015), 637–650.

[26] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Information Processing 74: Proceedings of the IFIP Congress 74*. North-Holland, 471–475.

[27] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–17.

[28] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. 438–454. doi:10.1145/3694715.3695952

[29] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[30] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear types for large-scale systems verification. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–28.

[31] Shaokai Lin, Yatin A. Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng-Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia. 2023. Towards Building Verifiable CPS using Lingua Franca. *ACM Transactions on Embedded Computing Systems* 22, 5s, Article 155 (2023), 24 pages. doi:10.1145/3609134

[32] Zhengqing Liu, Alberto Sonnino, Igor Zablotchi, Eleftherios Kokoris Kogias, and Marios Kogias. 2026. Remora: Scale-out Deterministic Execution for Smart Contracts. *Proceedings of the VLDB Endowment* 19, 11 (2026), 3076–3090. doi:10.14778/3836663.3836674

[33] Zhengqing Liu, Musa Unal, Matthew J Parkinson, and Marios Kogias. 2025. DORADD: Deterministic Parallel Execution in the Era of Microsecond-Scale Computing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 282–296.

[34] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems* 20, 4, Article 36 (2021), 27 pages. doi:10.1145/3448128

[35] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. ACM, 197–210. doi:10.1145/3385412.3385971

[36] Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 748–776.

[37] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. 2021. Verifying Determinism in Sequential Programs. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021)*. IEEE, 37–49. doi:10.1109/ICSE43902.2021.00017

[38] Roy Overbeek. 2020. Formalizing Determinacy of Concurrent Revisions. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. ACM, 258–269. doi:10.1145/3372885.3373820

[39] Christine Paulin-Mohring. 2009. A Constructive Denotational Semantics for Kahn Networks in Coq. In *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin (Eds.). Cambridge University Press, 383–413. doi:10.1017/CBO9780511770524.018

[40] Marcus Rossel, Shaokai Lin, Marten Lohstroh, Jeronimo Castrillon, and Andrés Goens. 2024. Provable Determinism for Software in Cyber-Physical Systems. In *Verified Software: Theories, Tools and Experiments (VSTTE 2023), Revised Selected Papers (Lecture Notes in Computer Science, Vol. 14095)*. Springer, 85–107. doi:10.1007/978-3-031-66064-1_6

[41] The Tokio Team. [n. d.]. Tokio: an asynchronous Rust runtime. https://tokio.rs/. Accessed: 2025-09-08.

[42] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[43] Verus Project. [n. d.]. Verus tutorial: SPSC Queue. https://verus-lang.github.io/verus/state_machines/examples/producer-consumer-queue.html. Accessed: 2026-06-03.

[44] Verus Project. 2025. Support async functions. GitHub pull request #1993, `verus-lang/verus`. https://github.com/verus-lang/verus/pull/1993 Accessed: 2026-06-03.

[45] Yunhao Zhang, Haobin Ni, Soumya Basu, Shir Cohen, Maofan Yin, Lorenzo Alvisi, Robbert van Renesse, Qi Chen, and Lidong Zhou. 2025. Ordered Consensus with Equal Opportunity. arXiv:2509.09868 [cs.DC] https://arxiv.org/abs/2509.09868

[46] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 633–649.