DORADD: Deterministic Parallel Execution in the Era of Microsecond-Scale Computing

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

Musa Unal
EPFL
musa.unal@epfl.ch

Matthew J. Parkinson
Azure Research
mattpark@microsoft.com

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

Abstract

Deterministic parallelism is a key building block for distributed and fault-tolerant systems that offers substantial performance benefits while guaranteeing determinism. By studying existing deterministically parallel systems (DPS), we identify certain design pitfalls, such as batched execution and inefficient runtime synchronization, that preclude them from meeting the demands of μs-scale and high-throughput distributed systems deployed in modern datacenters.

We present DORADD, a deterministically parallel runtime with low latency and high throughput, designed for modern datacenter services. DORADD introduces a hybrid scheduling scheme that effectively decouples request dispatching from execution. It employs a single dispatcher to deterministically construct a dynamic dependency graph of incoming requests and worker pools that can independently execute requests in a work-conserving and synchronization-free manner. Furthermore, DORADD overcomes the single-dispatcher throughput bottleneck based on core pipelining.

We use DORADD to build an in-memory database and compare it with Caracal, the current state-of-the-art deterministic database, via the YCSB and TPC-C benchmarks. Our evaluation shows up to 2.5× better throughput and more than 150× and 300× better tail latency in non-contended and contended cases, respectively. We also compare DORADD with Caladan, the state-of-the-art non-deterministic remote procedure call (RPC) scheduler, and demonstrate that determinism in DORADD does not incur any performance overhead.

PPoPP '25, March 1-5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03.
https://doi.org/10.1145/3710848.3710872

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**.

Keywords: parallel execution, determinism, runtime scheduling

1 Introduction

A deterministically parallel system (DPS) guarantees that *given exactly the same input, it will produce the same output via parallel execution.* Deterministic parallelism is a widely researched topic, tackled by multiple different communities, i.e., operating systems [7, 20, 46, 57], architecture [16, 22, 33], distributed systems [6, 29], databases [25, 63, 69], and programming languages [34]. In a nutshell, all these approaches try to eliminate the external sources of non-determinism, such as IO or random generators, and predictably control thread interleaving that performs parallel accesses to shared memory, which would otherwise introduce non-deterministic outcomes in multi-threaded applications.

Despite its various other usecases in testing and debugging [46, 56, 61], deterministic parallel execution finds its killer usecase in the space of distributed, replicated, and fault-tolerant systems. State machine replication (SMR) serves as the cornerstone for fault-tolerant systems [20, 29], where each node must execute a pre-agreed log of operations. A naive approach would be to execute all requests in a single thread to avoid diverging execution, yet limiting throughput. Deterministic parallel execution can safely eliminate the single-threaded execution bottleneck. Deterministically parallel log replay can be applied to fast failure recovery and live migration [28, 44, 45] in replicated databases, file systems, and blockchains. Furthermore, deterministic databases reduce the need for two-phase commit across partitions when processing transactions, helping to scale the performance of distributed databases [3, 69].

Most DPS in this space follow a modular approach. A sequencing layer external to the DPS is in charge of ordering and replicating operations. The DPS then execute these operations in parallel, ensuring that the final result is equivalent to a serial execution that follows the input sequence.

Commonly, the sequencing layer has been a performance bottleneck with high latency in the order of *ms* [69].

However, recent research on datacenter systems and μs-scale computing [8] has driven the development of low-latency distributed systems, challenging the previous state of the art in DPS design and implementation. Techniques such as kernel bypass, RDMA, and in-network compute allow distributed systems to achieve high throughput, reaching millions of requests per second, and latency of a few μs. Examples of such systems are Mu [4] and HovercRaft [38] for crash fault tolerance, uBFT [5] for byzantine fault tolerance, and NetChain [32] for primary-backup replication. Consequently, the bottlenecks in DPS have transitioned from the sequencing layer to the execution phase, emphasizing the imperative to revisit deterministic parallel execution for modern datacenter services.

In light of μs-scale systems, we revisit the design and implementation of DPS, discovering that none of the prior works adequately address both high-throughput and low-latency requirements essential for modern datacenter applications. Certain solutions are strictly optimized for throughput using batched execution, thus suffering from high latency [25, 26, 48, 63]. They also pay a high efficiency tax to enforce determinism, wasting CPU cycles and being non-work-conserving due to unnecessary synchronization. We identify that such inefficiencies can be exacerbated in contended [63] and heterogeneous [37] workloads consisting of operations with highly variable service times.

In this paper, we introduce DORADD (i.e., **D**eterministically **OR**dered **A**ccesses with **D**ynamic **D**AGs), a high-performance deterministic parallel runtime for modern datacenter stateful applications, e.g., key-value stores and transactional datastores. DORADD is a multi-core runtime that allows efficient execution of deterministically parallel Remote Procedure Calls (RPCs) with high throughput, low latency, and efficiency guarantees. Through a novel design of its scheduling mechanism, DORADD guarantees that i) request dependencies are respected, ii) incoming requests never have to unnecessarily wait to enforce determinism, thus leading to low latency, and iii) available work will harness idle CPU resources, i.e., work-conserving, even under highly contended cases, thus leading to high throughput. DORADD is a pluggable module, agnostic to the type of system used as a sequencing layer, serving as a high-performance execution engine for an ordered set of operations. For the rest of the paper, the terms RPCs, requests, and operations are used interchangeably to refer to the main execution unit in a DPS.

We implement DORADD in C++ and use it to build a series of applications including a deterministic in-memory database and primary-backup system. We evaluate DORADD in a series of micro- and macro-benchmarks and compare it with the current state-of-the-art deterministic and non-deterministic systems. Compared to Caracal [63], the state-of-the-art deterministic database, DORADD achieves up to 2.5× better throughput, due to work-conservation, while being able to do so with 12 fewer cores than Caracal in the YCSB and a modified TPC-C benchmark. DORADD also achieves 150× and 300× better tail-latency in non-contended and contended cases compared to Caracal. To examine the cost of determinism, we compare DORADD with Caladan [27], the state-of-the-art non-deterministic RPC scheduler, serving as an upper bound in achievable performance. DORADD can achieve the same throughput under latency SLAs as Caladan while guaranteeing determinism, i.e., it enables zero-overhead deterministic execution.

This paper makes the following contributions:

- An efficient scheduling scheme that decouples request dispatching from execution and enforces determinism while avoiding unnecessary synchronization.
- A scalable and reusable design for the single-dispatcher architecture [21, 31, 35] based on core pipelining.
- A high-performance implementation of DPS that can provide both high throughput (millions of requests per second) and low latency (at μs-scale).

DORADD is open-source and can be found at https://github.com/doradd-rt.

2 Background and Motivation

In this section, we revisit prior approaches to achieving deterministically parallel execution. We describe the common system model most DPS follow, break down their execution layer, and explain how they achieve determinism. Through this analysis, we identify several common pitfalls in existing schemes that lead to poor performance and inefficiency.

System Model: Most DPS targetting distributed and replicated systems assume an existing ordered sequence of operations they have to execute in parallel, ensuring that the final result will be equivalent to the sequential execution of the operations in their original order. Hence, the system remains agnostic to the input source and focuses on the execution. This modular approach of *order-execute* allows better flexibility since the sequencing layer and the execution engine can be chosen independently. The sequencing layer can be a state machine replication system, e.g., Raft [58], Paxos [41], or PBFT [12]. Apart from sequencing, this layer is also in charge of durably logging the input operations for fault-tolerance and recovery [48], thus removing the need for such functionality from the DPS. Recent research in datacenter systems [4, 5, 32, 38, 42] has shown that the sequencing layer can scale to millions of transactions per second and achieve μs-scale latency using kernel-bypassing, RDMA, programmable switches, and persistent memory. DORADD also assumes the existence of such a sequencing layer, the design of which goes beyond the scope of this paper.

There are also other models deployed in DPS that mingle the sequencing and execution layer, such as Rex [29] and Eve [36]. However, those are application-specific and not

modular, thus making them more difficult to deploy and adapt to different scenarios. For the rest of the paper, we focus on this modular design that separates the sequencing layer from the execution layer.

Deterministic Execution: The mechanisms to implement a deterministically parallel execution that respects a serial order of operations can be divided into two main categories: optimistic and pessimistic. Optimistic approaches [28, 40, 48] are reactive, i.e., they execute first and then periodically check whether different execution units access the same resources. In cases where conflicting accesses occur, parallel execution is aborted, and the system rolls back, resorting to either serial execution or retries. Optimistic approaches have to operate on batches of operations that indicate the boundaries in which the system checks for conflicts and rolls back; hence achieving low latency is challenging. While optimistic approaches perform well in low contention cases, their throughput and latency degrade significantly due to high abort rates when there are many overlapping accesses to the same resources.

Pessimistic approaches, on the other hand, are proactive and need to ensure that accesses to the shared resources indeed happen according to the agreed order. They follow a *schedule-execute* model. The scheduling phase determines which operations must be executed in a specific order and which can be run in parallel. For instance, when two operations modify shared resources, they are interdependent and cannot be processed in parallel; instead, they must follow the specified order to ensure determinism. During the execution phase, operations run respecting the constraints and order determined by the scheduling phase, leading to no conflicts and aborts due to races. As a result, pessimistic approaches excel in high-contention scenarios [17, 63], as they avoid costly aborts.

In the *schedule-execute* DPS, dependency analysis must be performed during the scheduling phase to identify conflicting operations. To achieve this, they depend on a predefined read-write set for each operation given by the user. This is a widely used pattern in deterministic databases [25, 26, 63, 69], concurrency control schemes for one-shot transactions [52, 53], and various blockchain systems for their smart contracts [68, 72]. While this requirement imposes additional complexity on the programming model with certain application logic, it greatly simplifies system design and enhances overall throughput by preventing conflicts and avoiding aborts.

Based on the dependency analysis, pessimistic schemes derive a deterministic schedule that is enforced in different ways, i.e., lock managers [69], multi-versioning [25, 63], and dependency graphs [26, 30, 39]. Calvin [69] uses a centralized lock manager to establish a lock order and ensure that shared resources are accessed according to the pre-defined order during the execution phase. Bohm [25] and Caracal [63] use multi-versioning with writes creating new versions and reads

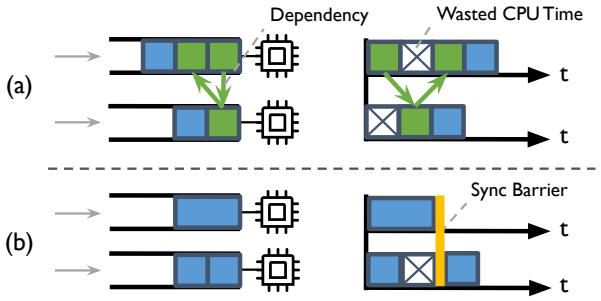

Figure 1. Inefficient runtime synchronization in prior works. The left column denotes the incoming requests. The right column denotes the execution in time series. Case (a) shows core stalling due to inter-dependent requests and case (b) shows so due to variance of service time (i.e., stragglers). For visualization, we assume single-transaction epochs in (b).

accessing specific old versions decided during scheduling. PWV [26], Kuafu [30], and CBASE [39] build a dependency graph of operations based on their conflicts and operation order. The resulting directed acyclic graph (DAG) defines the partial order of execution.

After studying prior works on DPS, we identify two major pitfalls that render them inappropriate for high-throughput and low-latency datacenter applications.

Problem 1: High latency due to batched execution (P1). Most of prior works on DPS adopt batched execution. Optimistic systems need well-defined batch boundaries to check for conflicts, while the reasons for this design choice in pessimistic systems vary. Calvin [69] assumes that the sequencing layer uses batches for scalability and inherits this choice. Caracal [63] and Bohm [25] perform parallel scheduling and require a batch of transactions to split across all threads. We study the impact of batching on throughput and latency for Caracal in §5.1. Batching, despite increasing throughput, incurs significant latency, thus masking the μs-scale benefits coming from emerging datacenter systems. We argue that this performance trade-off is not acceptable when building a high-performance DPS which can be user-facing, hence requiring low tail-latency.

Problem 2: Inefficient runtime synchronization (P2). By only focusing on deterministic execution, many prior DPS forgo an efficient use of the underlying CPU resources. First, several pessimistic approaches depend on inefficient busy-waiting to guarantee the correct order of accesses. For example, in Caracal [63] that implements a multi-versioned execution, a thread executing a transaction with a read-after-write dependency has to actively wait till the read version is ready, despite there being other transactions that this thread could run without any determinism violation. Second, many of the prior works, e.g., Bohm [25] and Granola [18] implement a static mapping of transactions to threads, which leads

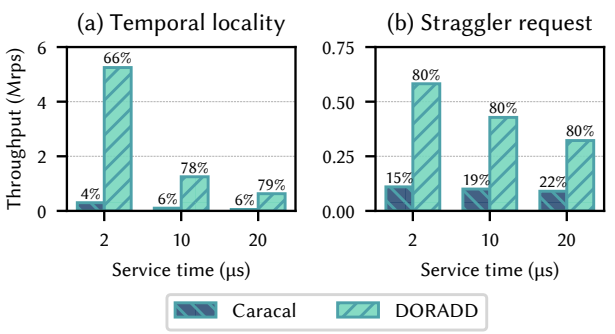

Figure 2. Caracal and DORADD throughput in synthetic read-spin-write workloads. The percentage denotes the ratio of actual throughput to the ideal throughput.

to imbalance with skewed workloads and non-conservation, i.e., there are idle threads despite the presence of work that can be executed without violating determinism. Finally, the use of batching and epochs introduces inefficiencies when dealing with highly variable service times, for example, straggler transactions [37, 74]. Despite running on a single thread, a long-running operation can prevent all other threads from starting the next batch due to the strict synchronization barriers.

Figure 1 illustrates some of the cases of inefficient CPU usage. It shows two CPU cores with their request queues, the assigned requests per queue, and how these requests are executed over time. In Figure 1a, the dependency among the green requests interconnected with arrows, combined with the core assignment, leads to wasted CPU time in both cores. Figure 1b shows how inter-core synchronization, e.g., epoch boundaries, combined with long-running transactions or load imbalance wastes CPU cycles in the second core.

To further demonstrate the effect of P2, we use Caracal [63], the state-of-the-art DPS, with two synthetic read-spin-write workloads, each corresponding to the cases in Figure 1. Each request needs to access 10 keys and spins for a configurable amount of time. We fix the number of CPU cores to 16 and measure the maximum achieved throughput. We also measure the equivalent achieved throughput for DORADD for the same workloads and compare it to the ideal throughput. In an ideal scenario, all CPU cycles are dedicated to processing the request, i.e., the throughput per worker core should be the reciprocal of the average service time. We assume such an ideal throughput where there are no dependencies across requests and it scales linearly with the number of cores. Figure 2 summarises the results.

First, we consider a workload with high contention. Arriving batches of requests show temporal locality, i.e., access a common key, thus leading to serialization within the batch, whereas requests belonging to different batches do not conflict. We assume batches of 100 requests. In this case, Caracal suffers extremely low CPU utilization with nearly sequential execution, hence 6% (1/16 cores) of the ideal throughput.

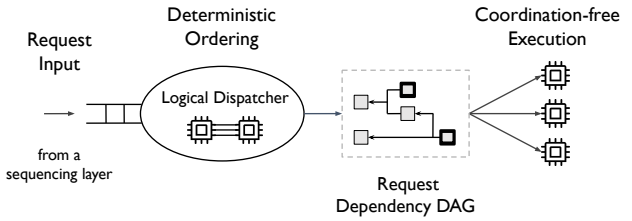

Figure 3. DORADD Architecture. The cores are split into dispatcher and worker cores. The pipelined dispatcher cores construct a dependency DAG of requests, while worker cores execute them in a coordination-free manner.

Even though the next batch of requests does not depend on the previous one, almost all cores busy-wait for dependencies within the previous batch. In contrast, DORADD manages to concurrently process requests that do not conflict on all the available cores, while conflicting requests run on the same core.

Second, we consider the straggler request scenario, following the similar setting in [48, 74], in which each batch of 10k requests contains a 20ms straggler. Caracal suffers a significant slowdown due to stragglers while DORADD stays resilient. Note that DORADD uses three cores for dispatching which we illustrate in §4 while the rest are worker cores for request processing, thus leading to maximum ratio as 81% (13/16 cores).

3 The DORADD Design

We design DORADD, a deterministically parallel runtime targeting *µs*-scale datacenter services. We set the following design requirements. DORADD should: i) follow the same *order-execute* pattern used in prior works, leveraging an external sequencing layer, while remaining agnostic to it; ii) allow for low latency deterministic responses, i.e., online latency-critical datacenter services should run on top of DORADD without compromises; iii) have a work-conserving design without unnecessary synchronization that achieves high efficiency and supports high throughput.

To accommodate low latency and highly contended workloads, DORADD adopts the pessimistic *schedule-execute* model and introduces a scheduling scheme that enforces deterministic execution, while tackling the two problems, P1 and P2. DORADD's programming model makes the same assumptions as prior works, in which users explicitly declare the read and write set of each operation.

3.1 High-level System Design

DORADD completely decouples request scheduling from request execution. There are dedicated cores to dispatch and schedule incoming requests and others to execute those requests. There is a single input queue to the system that consumes the input from the sequencing layer and acts

as a serialization point. The dispatcher cores operate as a *single logical dispatcher* and are the ones in charge of enforcing determinism at scheduling time. The job of the worker cores is to follow and respect the execution plan determined by the dispatcher cores. For now one can assume a single-core dispatcher for simplicity. §3.4 describes how to scale up the dispatcher without violating determinism. Such a design, inspired by state-of-the-art μs-scale RPC schedulers [21, 31, 35, 59], allows the two parts, scheduling and execution, to work completely independently in parallel and serve a contiguous flow of incoming requests, instead of alternating between scheduling and execution phases [28, 63, 69]. Consequently, DORADD operates without epochs, enabling low latency and avoiding P1.

The core of the deterministic scheduling logic is a dynamic Directed Acyclic Graph (DAG) that organizes incoming requests in a partial order based on the shared resources they require to access during their execution. For a given serial order of requests, there is a unique DAG. The dispatcher is in charge of constructing this DAG. Similar to dataflow programming, requests that need to access the same resource have a dependency on each other, and the order of these dependencies follows the order the dispatcher processes them. Several prior works use similar dependency analysis to derive parallel execution. However, without the focus on μs-scale datacenter services, they fall short in providing an efficient and high-throughput implementation [30], and in how they interleave the dependency analysis with execution, since they depend on epoch-based approaches that lead to high latency [26]. In contrast, DORADD builds a dynamic DAG in an epoch-free manner and bypasses inefficiencies of prior works through its pipelined dispatcher (illustrated in §3.4).

Worker cores can execute a request from the DAG when it has no dependencies, i.e., when all previous requests that access overlapping shared resources have finished their execution. Requests run to completion in a coordination-free manner, i.e., they do not need to use locks or similar synchronization primitives, since the deterministic scheduling by the dispatcher guarantees exclusive access to those shared resources by design. Once a request finishes its execution, its dependent requests become available for execution on the worker cores. This architecture ensures a work-conserving execution, i.e., as long as there are requests with no dependencies, they will be executed immediately if there is an idle core. Also, worker cores will never have to inefficiently busy-wait to guarantee determinism, thus solving P2.

Figure 3 summarizes the above description. Requests come into the dispatcher from the sequencing layer. The dispatcher serializes their order on its single queue and dynamically constructs the requests' DAG based on their dependencies. Workers consume this DAG without needing to synchronously communicate with the dispatcher. Bold boxes represent requests that can be executed, i.e., dependencies are resolved.

```
// Transfer procedure with src/dst resources to write to.
void transfer(Resource* src, Resource* dst,
  uint64_t amount) {
  src->balance -= amount;
  dst->balance += amount;
}

// Balance procedure with account resource to read from.
uint64_t balance(Resource* account) {
  return account->balance;
}
```

Listing 1. Bank example in DORADD.

3.2 Programming Model

Following the majority of DPS approaches discussed in §2, DORADD adopts a programming model that explicitly bundles the RPC logic with the shared resources—of any form and granularity as determined by the programmer—that the RPC needs to access. This design is crucial for decoupling scheduling from execution, enabling more efficient resource management and execution flow.

DORADD leverages two main abstractions: ***procedures*** and ***resources***. A *procedure* is a unit of single-threaded execution and implements the RPC logic. A *resource* is any part of the system state for which the order of accesses could interfere with the system's deterministic execution. In its simplest form, a *resource* is a piece of memory, e.g., a C++ object. A *resource* can not be accessed concurrently by multiple threads. *Procedures* take *resources* as arguments and cannot access any other *resources* beyond their argument list because this would be a determinism violation. When a *procedure* is executed in DORADD, it has guaranteed exclusive access to its *resources*. Consequently, DORADD also avoids data races and deadlocks resulting from error-prone implementation when coupling concurrency control with scheduling. Currently, there is no difference between read and write resources, as they are both treated equally as a dependency in DORADD. We leave this optimization as future work.

Programmers can write applications on top of DORADD as RPC services. Each DORADD application exposes a set of RPC endpoints. Each RPC type corresponds to a *procedure* that takes *resources* as arguments. Resource names are known to clients and there is a mapping from names to the actual resources taking place during scheduling in DORADD.

As an example, consider a simple bank application (Listing 1) that only allows for two transaction types, one to get the current account balance and one to transfer money between accounts. In this scenario, each account is a *resource*, the transaction to get the account balance is a *procedure* with a single *resource*, and the transaction for account transfers is a *procedure* with two *resources*.

Figure 4 shows an ordered stream of requests for the above bank application coming into the dispatcher. Every request explicitly states the procedure name, `balance` or `transfer`,

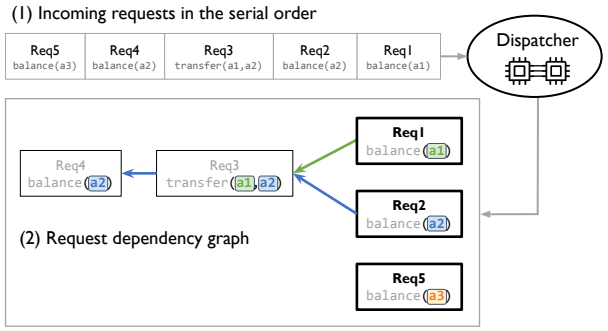

Figure 4. From serial to partial RPC order based on their *resource* dependencies. Bold requests on the right (Req1, Req2, and Req5) are ready to run.

and the set of *resources*, i.e., accounts a_i, it needs to access and use as arguments in the procedure call. The dispatcher dynamically constructs the DAG seen in the same figure by inspecting the overlapping dependencies. Req1, Req2, and Req5 can execute immediately since they have no dependencies to wait for. Req3 needs to wait for the termination of Req1 and Req2, since they have overlapping accesses to the same *resources*, i.e., a_1 and a_2. Similarly, Req4 needs to wait for Req3 because of a_2. We note that these are only scheduling dependencies enforced by the dispatcher and DORADD's workers will never have to actively wait for those. As long as there are requests to be executed, i.e., Req1, Req2, and Req5, they can run on any worker core.

Limitation: A limitation of this programming model, which is similar to all other pessimistic DPS, is that it cannot support all types of RPC services. Instead, it requires that all the shared resources of a request are known at dispatching time. Despite seeming limiting, prior works have shown that such a programming model can express a variety of stateful applications, such as deterministic databases [25, 26, 63, 69], one-shot transactions in concurrency control schemes [52, 53], a wide range of blockchain systems for their smart contracts [68, 72], and even certain types of microservices, e.g., carefully crafted CRUD (Create Read Update Delete) RESTful APIs [1].

3.3 DORADD Scheduling Scheme

DORADD introduces a novel scheduling scheme that is specifically designed for low-latency, without batching requirements. It is work-conserving, which increases efficiency, without violating determinism, and reduces the need for coordination between the dispatcher and workers, resulting in lower overhead. It follows a hybrid approach in which both the dispatcher and the workers participate in scheduling, i.e., the process deciding which worker core will run which procedure and when. This hybrid approach substantially reduces the dispatcher load, whose main duty is to construct the DAG, and not schedule *procedures* to cores, thus improving the dispatcher throughput.

When a request enters DORADD, it is first processed by the dispatcher, which identifies its corresponding *procedure* and required *resources*. The dispatcher then adds the *procedure* to the DAG. Each *resource* required by the *procedure* creates a dependency on the most recently scheduled *procedure* that also needs the same resource. For example, in Figure 4, Req3 depends on both Req1 and Req2, meaning it cannot be executed until both have completed. Adding a new *procedure* to the DAG can lead to an unconnected node, e.g., Req5. Such *procedures* do not have any dependencies and can be executed immediately. The dispatcher adds such *procedures* to the *runnable procedures set*.

The *runnable procedures set* is an unordered set of *procedures* that has no unresolved dependencies, allowing them to be executed in any order without violating DORADD's determinism guarantees. Bold boxes in Figure 3 and Figure 4 indicate *procedures* in the *runnable procedures set*. Workers can then actively remove from this set and execute them without needing to coordinate with the dispatcher. Any worker can execute any procedure from this set and the dispatcher does not participate in this decision. The dispatcher only adds *procedures* to this set, yet the workers both add and remove, forming DORADD's hybrid scheduling scheme.

A worker constantly pulls *procedures* from the *runnable procedures set* and runs them to completion, without needing to wait on any synchronization primitives. When a *procedure* finishes its execution, the worker inspects any *procedures* from the DAG that depend on the finished one. If they are eligible for scheduling, i.e., if they have no unresolved dependency, the worker adds those procedures to the *runnable procedures set*, so that any worker can execute them. For example, after the worker core finishes executing Req3 (Figure 4), it also adds Req4 to the *runnable procedures set*, because Req4 depends solely on Req3. The worker then pulls a new *procedure* from the set and continues in the same way.

This decoupled and hybrid scheme offers DORADD its performance benefits. First, unlike any other previous DPS, it operates in a contiguous streaming manner and does not require epochs that hinder latency. Second, *procedures* are eagerly added to the *runnable procedures set* as long as they have no pending dependencies without being affected by any stragglers, thus leading to low latency. Finally, workers can independently pull from this set if idle without coordination with the dispatcher, thus achieving work-conservation and reducing expensive inter-core communication.

3.4 Scalable Pipelined Dispatcher

The single-dispatcher design is the cornerstone of DORADD's deterministic execution, since it guarantees the creation of a unique DAG given a sequence of requests. However, such a design raises concerns about the dispatcher becoming a potential throughput bottleneck [63] if implemented on a single

core. DORADD leverages two mechanisms to mitigate this limitation. First, it employs a pipelined [66] dispatcher architecture, which leverages multiple cores to increase throughput. This approach effectively extends the single-core dispatcher into single logical dispatcher pipeline, preserving determinism while enhancing performance. Second, it carefully eliminates pipeline stalls through prefetching.

The rationale behind a deterministic pipeline architecture is the following. In a single-core system where tasks are divided into independent sub-tasks with each taking t_i time, the throughput is $\frac{1}{\sum t_i}$. Organizing these tasks in a pipeline will increase the system throughput to $\frac{1}{max(t_i)}$. Consequently, breaking the dispatcher logic into deterministic sub-tasks that do not interfere, i.e., access disjoint resources, and assigning those tasks to a set of cores can help the dispatcher scale up to multiple cores, thus improving its throughput.

The throughput of the dispatcher pipeline is determined by its slowest processing stage. DORADD optimizes each dispatcher component and eliminates any expensive DRAM accesses on the dispatcher datapath, which significantly impacts performance. When adding a *procedure* to the DAG, the dispatcher must access at least one cache line per *resource*. For a large *resource* namespace with uniform access patterns, this could potentially result in an equal number of DRAM accesses to the number of *resources* in each RPC. For example, for requests with 10 *resources* each and a DRAM access latency of 100 *ns*, the dispatcher throughput cannot be more than 1 Mrps. To mitigate that, DORADD dedicates a core in the dispatcher pipeline to prefetch *resources* requested by each RPC, such that when later pipeline stages need to access those they will be found in cache.

The current dispatcher pipeline consists of three main components. First, the `Indexer` performs name resolution and finds the address of the target *resources* in each request. Then, the `Prefetcher` prefetches those *resources* from DRAM to the CPU cache. Finally, the `Spawner` is in charge of constructing the DAG based on the *resources* each *procedure* requires. Each component can run on a separate CPU core and communicate with adjacent ones through bounded SPSC queues. Cores process all incoming requests in the same order and move forward almost in tandem for efficient cache usage.

4 Implementation

We implemented DORADD in 6.3k LoC of C++. This includes a heavily modified Verona runtime [51], used in BoC [13], to manage the workers. DORADD eliminates dynamic memory allocations in the dispatcher with memory pools and uses huge pages to reduce page walk overheads and cache pollution due to TLB misses.

Runnable procedures set: DORADD organizes the *runnable procedures set* as a group of per-worker queues, instead of implementing a shared data structure that could become a bottleneck due to concurrent accesses from all the cores. The per-worker queue is a lock-free multi-producer multi-consumer (MPMC) queue [51], facilitating fast request scheduling and work-stealing. Hence, the *runnable procedures set* is distributed across these per-worker queues. When a worker needs to add to the set, it assigns to its own queue. When the dispatcher needs to add to the set, it selects a worker queue in a round-robin manner. Workers pull from the set by first checking their own queue, and if empty, they steal work from other worker queues.

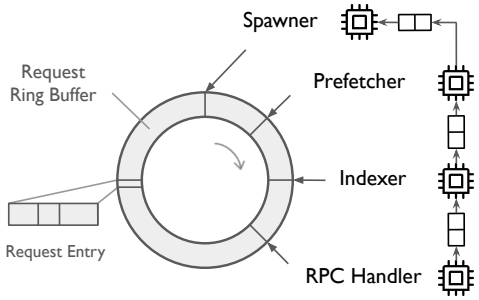

Figure 5. The logical dispatcher pipeline. Each core operates a sub-task in dispatching and communicates via a bounded SPSC queue. Request ring buffer is shared across all cores.

Dispatcher pipeline: DORADD uses a ring buffer for incoming requests, which is shared across all cores in the pipeline. Every core processes requests in place and uses a bounded SPSC queue to signal the next core in the pipeline to take over. An additional pipeline stage at the beginning serves as the RPC handler that processes the input from the sequencing layer and prepares the request entry in the buffer, i.e., identifying the target *procedure* and *resource* names, followed by the `Indexer`, the `Prefetcher`, and the `Spawner`. Figure 5 describes the dispatcher pipeline architecture.

To reduce inter-core communication and further improve the dispatcher pipeline throughput, DORADD leverages *adaptive bounded batching* [9] in the scheduling phase. Each pipeline stage communicates to the next stage over the SPSC queue the number of ring entries it should process next, instead of communicating one entry at a time. Once a core completes processing a small batch of requests, it pushes the batch size into the SPSC queue for the next core. The push operation is blocking if the queue is full, ensuring backpressure when necessary. The first core in the pipeline (RPC handler) determines the size of each batch, i.e., 1 to the maximum batch size, based on the available requests at the system input. It never waits for a full batch; instead, it adaptively processes all available requests up to the batch limit to maintain low latency. In the evaluation, we use bounded queues of 4 entries and a maximum batch size of 8, since this configuration performs better for the target workloads. Notably, though DORADD leverages adaptive batching in scheduling as a performance optimization, DORADD executes requests

without batching, unlike prior works (P1), where batched execution groups thousands of requests per epoch, significantly increasing latency.

The number of dispatcher cores is statically decided at the launch of the system, based on the target workload. We study the impact of the number of pipeline cores in §5.4. Dynamic assignment of cores is left as future work.

5 Evaluation

Our evaluation aims to answer the following questions:

- How does DORADD compare with state-of-the-art DPS? (§5.1)
- What is the cost of enforcing determinism in RPC services and how does DORADD compare with state-of-the-art non-deterministic μs-scale RPC schedulers? (§5.2)
- How can DORADD improve the performance of fault-tolerant applications? (§5.3)
- What is the performance contribution of each dispatcher optimization and what are the limitations of pipelined architecture? (§5.4)

Testbed: We run our evaluation on a mix of CloudLab [24] and local testbed machines. Our local testbed consists of two directly connected machines with an Intel Xeon Gold 5318N CPU with 24 cores and 1 NUMA zone running at 3.40 GHz, 128 GB of DRAM, and 100G Mellanox ConnectX-6 NICs. On CloudLab we used 3 `d6515` servers interconnected by a switch. All machines run Ubuntu 22.04 kernel with hyper-threading disabled.

5.1 Performance Comparison with other DPS

Methodology: We choose Caracal [63] as a baseline representing other DPS. Caracal is the current state-of-the-art deterministic database, achieving millions of transactions per second and surpassing prior works [18, 25, 26]. Caracal uses multi-version concurrency control and batches transactions into epochs. Each epoch contains two phases, i.e., one concurrency control phase to determine conflicts and one execution phase. To compare with Caracal, we use DORADD to implement an in-memory database. We run all experiments on one of our local testbed machines and dedicate one core to generate requests based on an open-loop Poisson process by replaying a memory-mapped pre-generated request log containing 1M requests. In our experiment, there is no sequencing layer to order the input; we assume that logging is handled by that layer before the request enters our system. The open-sourced codebase [64] of Caracal relies on statically generating transactions for each core and each epoch before runtime. To align the experiment setting for Caracal, we modified 300 LoC to enable it to receive incoming requests from the same generator core. We also disable syscall-involved logging in Caracal to make a fair comparison with an in-memory database built via DORADD.

		YCSB	TPCC-NP
Transaction Types		10 keys from 10M keys (Number of hot keys: 77)	NewOrder Payment
Contention	**No**	8 reads, 2 writes 0/10 hot keys	23 warehouses
	Mod	All writes 3/10 hot keys	8 warehouses
	High	All writes 7/10 hot keys	1 warehouse

Table 1. YCSB and TPCC-NP configurations.

Experiments: We run the YCSB [17] and TPCC-NP [2] benchmarks configured as seen in Table 1. In YCSB, we adopt the same settings as in Caracal. We group 10 unique key accesses in a single request and configure each row with 900 bytes. A read operation reads the entire row, while a write operation updates the first 100 bytes. Both uniform and skewed access patterns are considered: the former is represented by `No Contention` case in Table 1 and the latter is modeled by selecting hot keys to represent contention, as same as the settings in Caracal. Hot keys are chosen from 77 rows that are spaced 2^{17} apart in the 10M key space, while other keys are picked uniformly. In TPCC-NP, following the same modeling approach as in prior work [11, 65, 71, 73], we use an equal mix of NewOrder and Payment transactions, which account for 88% of all transactions in standard TPC-C. The default TPC-C sets the number of warehouses equal to the number of CPU cores and has low contention. As we dedicate one core as the client load generator, we have 23 cores available in our testbed as worker cores, therefore we set 23 warehouses to represent a non-contention scenario.

Results: Figure 6 shows the latency vs throughput curves for Caracal and DORADD in the YCSB and TPC-C benchmarks. Overall, DORADD significantly outperforms Caracal in both latency and throughput. It achieves similar throughput for non-contended cases yet up to 2.5× better throughput for contended ones. It achieves more than 150× and 300× lower tail latency for non-contended and contended cases. Throughput benefits are due to the DORADD's work conservation, while Caracal suffers from unnecessary spinning under contention. Caracal will block the current worker core until the target version of objects is ready, the effects of which are pronounced in high contention cases, resulting in efficiency reduction. On the contrary, DORADD harnesses all worker cores for useful work. More importantly, DORADD achieves significantly better tail latency thanks to the decoupled architecture which allows the dispatcher and the workers to operate independently. Reducing the epoch size in Caracal does improve latency, but also significantly reduces throughput.

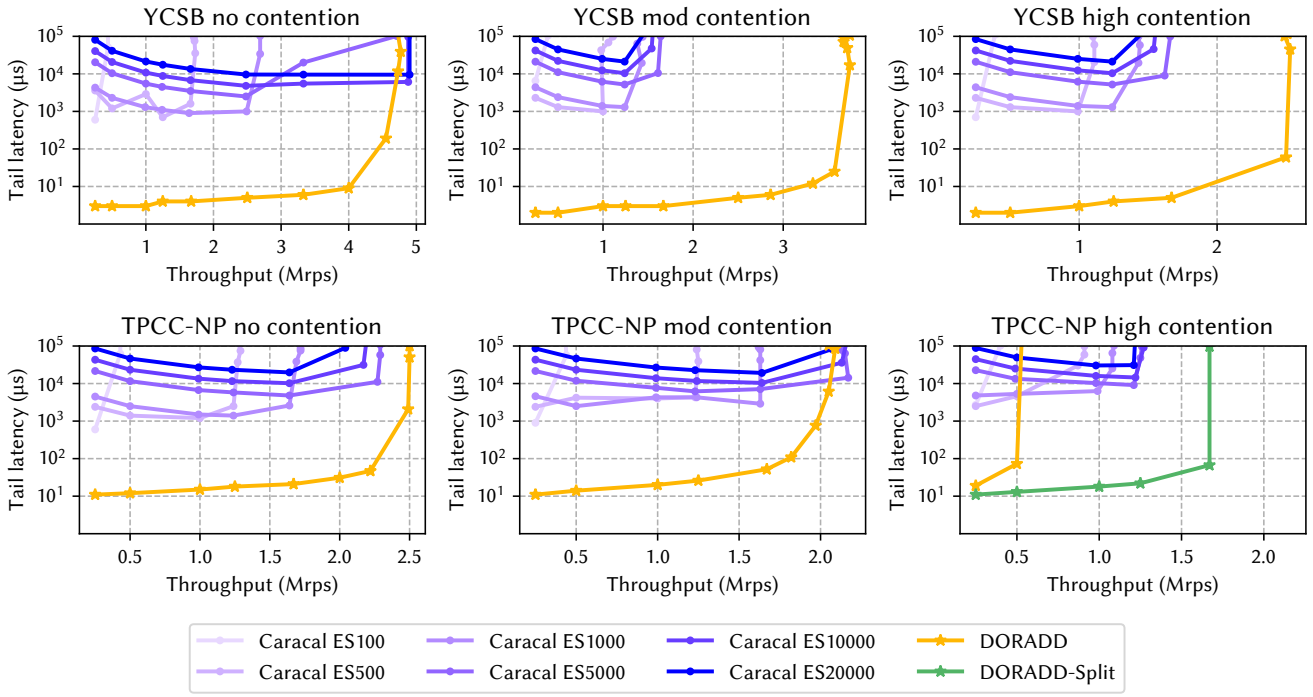

Figure 6. YCSB and TPCC-NP Latency v.s. Throughput. Caracal ES represents different epoch sizes in number of transactions.

An interesting scenario that showcases the flexibility of DORADD's programming model appears in the highly contended TPC-C experiment. High-contended TPC-C uses a single warehouse, in which all requests require the same *resource*. Consequently, a naive implementation of the benchmark leads to a serial execution of all requests, which suffers in terms of throughput as seen by the DORADD curve. By studying the TPC-C transaction implementation we identified that warehouses are updated independently, thus the transaction logic can be split to the contended warehouse update part, and the rest of the transaction logic. In this way, DORADD's dispatcher atomically schedules two subtransactions that can execute in parallel. After splitting, DORADD achieves 1.65Mrps (indicated as DORADD-split in the same figure) while Caracal only reaches 1.2Mrps.

Efficiency: During the evaluation, we experimented with reducing the number of cores for both Caracal and DORADD. We observed that DORADD achieved the above performance even with 8 worker cores, while adding more workers did not change the maximum throughput, since the dispatcher or the workload itself can limit scalability. Caracal required all 23 workers to achieve that performance, and for reference achieves 3.8Mrps in non-contended YCSB and 1.6Mrps in non-contended TPC-C, which are 0.7x of the performance with 23 cores.

5.2 The Cost of Determinism

Methodology: To study the cost of determinism, we compare DORADD with Caladan [27], the state-of-the-art RPC scheduler, which is inherently non-deterministic by design. We reuse the synthetic applications from Caladan artifacts and made 100 LoC changes. Upon receiving a request, each worker core operates in a two-phase locking manner: 1) acquires a specified number of locks, 2) performs some simulated work, equivalent to a fixed amount of service time, and 3) releases the locks. To guarantee deadlock-freedom, workers acquire locks with lower IDs first and release them in reverse order. Caladan implements an asynchronous user-level mutex which yields the thread execution if the mutex is already held during an acquisition attempt. The first baseline runs Caladan with its asynchronous mutex. We also implement a spinlock-based non-deterministic baseline. We use UDP RPCs to both Caladan and DORADD with minimum single-packet requests and responses.

Experiments: We run experiments in which each RPC accesses 10 different locks from a 10M keyspace following a uniform or a Zipfian distribution with different θ parameters and spins for 5 or 100 μs. We choose these two service times to study the impact of determinism both in terms of system overheads with short requests and in terms of inherent parallelism reduction at the workload level with longer requests that diminish the effects of system inefficiencies. For both Caladan and DORADD, we use 8 worker cores and 1 dispatcher core. Given that Caladan introduces mechanisms to

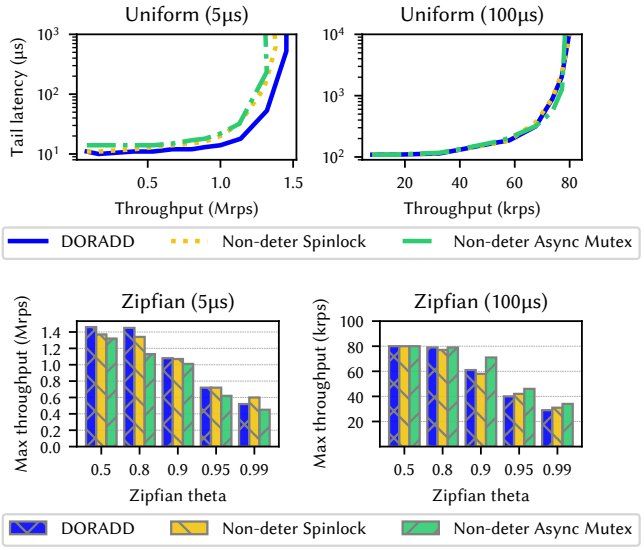

Figure 7. Performance comparison with non-deterministic systems.

trade efficiency with tail latency, we configure it in the performance mode (set optimize build mode and use spinning kthreads) to make a fair comparison. We run this experiment on our local testbed with one client and one server, both equipped with a DPDK networking stack. We measure the tail latency under different load levels and the maximum throughput each configuration can achieve.

Results: We first focus on the latency curves in Figure 7. For the 100μs workload, where system overheads can be masked, all systems achieve similar performance. For the 5μs workload where system overhead can be manifested, we find that DORADD achieves slightly better tail latency and maximum throughput. In DORADD, the dispatcher core issues prefetching and executes atomic operations for each key and each worker only executes the real workload, whereas for non-deterministic systems, each worker core needs to perform additional atomic operations. Better cache locality for atomic operations results in better performance in DORADD. We only show results for the uniform distribution, but the Zipfian distributions behave similarly. Overall, we can claim that DORADD and deterministic execution do not incur any extra overhead when focusing on the achieved throughput under a latency SLA, i.e., 1 ms in this case.

When focusing on the maximum throughput, results look slightly different, especially under high contention. Under higher contention ($\theta > 0.9$), non-deterministic systems exhibit higher throughput (at max 15% compared to DORADD) due to the absence of determinism. Determinism introduces more dependencies among requests compared to just mutual exclusion. Thus it overall limits the available workload parallelism and reduces the maximum throughput. In the same figure, we can also observe the benefits of the asynchronous mutex execution compared to the spinlock-based execution

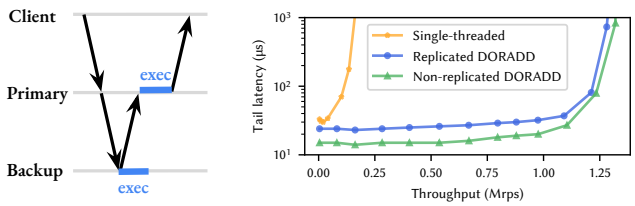

Figure 8. Active primary-backup replication. The primary does not need to wait for execution in the backup due to determinism.

among non-deterministic systems, since the asynchronous mutex makes better use of the underlying CPU resources and allows requests that can make forward progress to proceed. The same property exists in DORADD as well while respecting determinism.

5.3 End-to-End Use Case: Replication

One of DORADD's killer use cases is to increase the throughput of a replicated system since it offers an application-transparent way of doing so. To illustrate this and DORADD's use as a pluggable component, we implement the simplest form of a replicated system, i.e., an active primary-backup, to act as the sequencing layer. For this experiment, we use 3 machines on CloudLab with one client, one primary server, and one backup server. We deploy the 5μs synthetic application of the previous section with a uniform distribution and consider two baselines. A non-replicated, hence not fault-tolerant, version of DORADD serves as an upper limit in terms of throughput and lower limit in terms of latency. The second baseline is a replicated single-threaded system which is the canonical way of deploying such services to guarantee deterministic state transitions. In this experiment, the primary-backup subsystem does not implement durable logging, since this is beyond the scope of this work. Such systems can scale and achieve μs-scale latencies through the use of persistent memory and fast networking [70].

Figure 8 presents the replication process and experiment results. Replicated DORADD achieves both high throughput and fault tolerance, while guaranteeing low latency with only a trivial throughput cost. It has the same latency as the single-threaded execution due to the extra round-trip, while it achieves 1.28Mrps, which is marginally less than the maximum throughput (1.31Mrps) of the non-replicated version.

5.4 DORADD Design Analysis

Dispatcher optimizations: DORADD leverages prefetching and core pipelining to build a high-performance dispatcher. To evaluate the impact of these mechanisms on DORADD's performance, we conduct experiments using

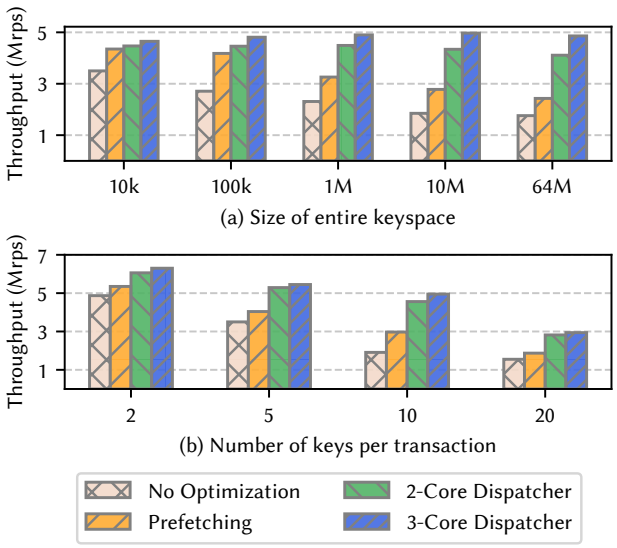

Figure 9. Performance contribution for DORADD's dispatcher optimizations, i.e., prefetching and core-pipelining.

synthetic workloads with four variants: ① single-core dispatcher without prefetching (No Optimization), ② single-core dispatcher with prefetching (Prefetching), ③ two-core dispatcher where one core handles both index lookups and prefetching, and another core is responsible for spawning tasks, ④ three-core dispatcher with one core dedicated to indexing, another to prefetching, and a third for spawning tasks. To explore the impact of optimizations upon memory and cache behaviors, we vary i) the size of the entire keyspace and ii) the number of keys for each transaction.

Figure 9a shows the maximum achieved throughput for various sizes of key space, with 10 keys chosen uniformly for each transaction. We observe that as the size of the key space increases, the importance of the pipeline design becomes more significant. Also, the pipeline design allows throughput to stay around the same levels despite the increased memory pressure.

Figure 9b shows the maximum achieved throughput as a function of the number of keys in a request for a 10M keyspace. We observe a consistent benefit from prefetching and pipelining across configurations. Throughput decreases though, as the number of keys per request increases, given the `Spawner` is the current bottleneck in the pipeline. The `Spawner` needs to do atomic operations, which in the worst case are equal to the number of keys. Although we could further pipeline the execution of the `Spawner`, this is not necessary for DORADD's current performance goals. Our rationale was that for more keys per request, request execution would become more expensive, thus the bottleneck would shift to the worker cores.

Limitations of the pipeline design: While core-pipelining provides evident benefits for scaling up a single logical dispatcher, we set out to investigate its limitations. We study

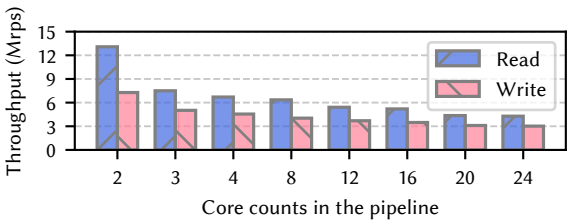

Figure 10. Pipeline throughput with various core counts in the pipeline. Multiple cores in a pipeline reads/writes a shared ring buffer sequentially.

the introduced overheads and how these change as a function of the number of cores in the pipeline. We eliminate the processing in each stage to the bare minimum, i.e., a single read or a write operation on the shared ring entry. We use the same pipeline infrastructure, i.e., shared ring, bounded queues of size 4, and a maximum adaptive batch size of 8. Although different queue and batch sizes could lead to higher throughput, we use the same settings as the rest of the evaluation.

Figure 10 shows the maximum achieved throughput as a function of the number of cores in the pipeline for read and write accesses. As expected, adding cores in the pipeline reduces the overall throughput due to their inter-core communication overheads, while throughput is lower when all cores need to update the shared cache line. Note that in the current DORADD design, only the `Indexer` needs to modify shared memory to be read by the `Prefetcher` and `Spawner`. Based on this analysis, one can conclude that the current DORADD design could support more stages by breaking down the `Spawner` as identified in the previous experiment. Furthermore, this analysis offers insights for future systems that require a single dispatcher that could be pipelined, beyond the scope of DPS.

6 Discussion

Handling other sources of non-determinism: In the current DORADD's implementation and evaluation, we only focus on internal sources of non-determinism, i.e., accesses to shared memory resources, which are necessary for transactional datastores. However, DORADD's programming model can be extended to cover more general-purpose RPCs supporting external sources of determinism, such as timers and random number generators. For example, a random number generator is a *resource* that incoming requests can require as any other and produces numbers in a pseudo-random and deterministic way.

Failures and checkpointing: Many DPS implement checkpointing mechanisms to bootstrap the system execution from a snapshot and reduce the size of the operation log maintained by the sequencing layer that needs replay. Moving away from epoch-based approaches could complicate such mechanisms. A straightforward way to implement checkpointing in DORADD is to periodically stop the dispatcher

from processing new requests, wait for the worker queues to drain completely, and then take a memory snapshot. DORADD's superior throughput performance can increase the duration of the checkpointing epochs, and thus reduce the frequency of pipeline stalls. DORADD currently does not implement checkpointing.

Non-preemptive scheduling: DORADD implements non-preemptive scheduling for *procedures*, with each procedure executed to completion. DORADD can be extended with cooperative or even preemptive scheduling without violating deterministic execution. Long-running procedures can either cooperatively yield execution or be preempted and parked in the runnable procedure set, where they will be scheduled for execution at a later time. Worker cores schedule dependent procedures only after the corresponding procedure has completed, rather than when it is preempted or yields, ensuring a deterministic execution outcome.

7 Related Work

Deterministic parallel execution has been approached by different research communities.

Databases: Calvin [69] is an early deterministic shared-memory epoch-based database, which uses a centralized single-threaded lock manager to serialize lock orders. Unlike Calvin, Bohm [25] and PWV [26] take a partitioned approach in lock acquisition, which introduces several issues, i.e., load imbalance, poor scaling with skewed or hard-to-partition workloads, and increased programming complexity. QueCC [62] adopts a priority queue-oriented approach to dispatch transactions to different partitions to avoid aborts. Aria [48] optimistically overcomes the limitation of predefining read-write sets with deterministic reordering by transforming read-after-write dependencies to write-after-write ones, yielding a deterministic result that differs from that of the agreed-upon order. All of these works are primarily built atop an epoch-based design, leading to high latency and low efficiency. DecentSched [14] uses per-object queuing and a decentralized scheduling scheme, yet relying on epoch-based execution. Recent work also studies distributed multi-partition deterministic transactions [55, 74].

Operating systems and architecture: Several prior works focus on thread or process level determinism [10, 22, 46]. Kendo [57] introduces the notion of a deterministic logical clock to construct a lock acquisition schedule based on instruction counters. Other works re-architect the memory consistency model [23, 47] or rely on extensive kernel modifications [49, 50]. DetTrace [54] proposes reproducible containers yet in single-threaded execution, mainly targeting debugging and testing. There are also research efforts on deterministic GPUs [16, 33].

State machine replication: CRANE [19], similar to Rex [29] and Eve [36], follows a co-design approach, builds on top of Parrot [20], and runs consensus for each system call, to implement a DPS for SMR. Alchieri et al. [6] suggest running requests touching disjoint memory shards on separate dedicated threads, while cross-shard ones need to synchronize using barriers, which suffers from P2. Kuafu [30] and CBASE [39] construct dependency graphs, but fail to provide an efficient and high-performance implementation without the focus on μs-scale datacenter services. Sparkle [43], Harmony [40], and Spectrum [15] process transactions speculatively and abort in case of conflicts. Meerkat [67] proposes zero-coordination principle to avoid inter-core and cross-replica synchronization for scalable transactions. DORADD conforms to this principle and shows high efficiency for deterministic systems.

Programming languages: Deterministic parallel Java [34] provides deterministic-by-default semantics using compile-time checking. Behaviour-oriented Concurrency (BoC) [13] is a new concurrency paradigm that forgoes concurrent mutation and ensures exclusive access to the underlying resources. Despite not targeting DPS, BoC implements a similar programming model to DORADD, in which developers are expected to bundle the expected memory accesses with the units of execution.

Non-deterministic μs-scale RPC scheduling: Going beyond deterministic parallel execution, the single-dispatcher multi-worker design has been widely adopted in μs-scale RPC scheduling. Several systems [21, 31, 35, 59] use an asymmetric single-dispatcher system, similar to DORADD, yet they do so to implement better RPC scheduling policies and improve tail latency. Unlike DORADD though, these works use a single-threaded dispatcher that could become a throughput bottleneck. ZygOS [60], a symmetric, non-preemptive system uses work-stealing similar to DORADD, to approximate a single queue.

8 Conclusion

DORADD is a high-performance deterministic parallel runtime for modern datacenter services. It employs an efficient scheduling scheme which decouples dispatching, i.e., deterministically constructing dependency graphs, from work-conserving and synchronization-free execution. It leverages core pipelining to scale the single-dispatcher throughput. DORADD achieves much better latency and higher throughput than state-of-the-art DPS, and induces no performance overhead compared to state-of-the-art non-deterministic RPC schedulers.

Acknowledgments

We thank the anonymous reviewers, the members of the LSDS group, Adrien Ghosn, Ashvin Goel, Diyu Zhou, and James Larus for their detailed and valuable feedback. We also thank Joshua Fried for helping us setup and run Caladan. We appreciate CloudLab [24] for providing the experiment platform. This work is supported by a gift from Mysten Labs.

References

[1] 2023. The Rise of REST API. https://blog.restcase.com/the-rise-of-rest-api/. Accessed: May 14, 2024.

[2] 2023. TPC-C Benchmark. https://www.tpc.org/tpcc/. Accessed: December 2, 2023.

[3] 2024. Volt Active Data. https://www.voltactivedata.com/. Accessed: August 10, 2024.

[4] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *OSDI*. USENIX Association, 599–616.

[5] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *ASPLOS (2)*. ACM, 862–877.

[6] Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. 2018. Early scheduling in parallel state machine replication. In *Proceedings of the ACM Symposium on Cloud Computing*. 82–94.

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

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 49–65.

[10] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. 2010. Deterministic Process Groups in {dOS}. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[11] Nils Boeschen and Carsten Binnig. 2022. GaccO-A GPU-accelerated OLTP DBMS. In *Proceedings of the 2022 International Conference on Management of Data*. 1003–1016.

[12] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association, 173–186.

[13] Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters: Behaviour-Oriented Concurrency. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 276 (oct 2023), 30 pages. https://doi.org/10.1145/3622852

[14] Chen Chen, Xingbo Wu, Wenshao Zhong, and Jakob Eriksson. 2024. Fast Abort-Freedom for Deterministic Transactions. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 692–704.

[15] 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.

[16] Yuan Hsi Chou, Christopher Ng, Shaylin Cattell, Jeremy Intan, Matthew D Sinclair, Joseph Devietti, Timothy G Rogers, and Tor M Aamodt. 2020. Deterministic atomic buffering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 981–995.

[17] 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.

[18] James Cowling and Barbara Liskov. 2012. Granola:{Low-Overhead} distributed transaction coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 223–235.

[19] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos made transparent. In *SOSP*. ACM, 105–120.

[20] 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.

[21] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 621–637.

[22] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 85–96.

[23] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: a relaxed consistency deterministic computer. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 67–78.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[25] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. , 1190–1201 pages.

[26] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[28] 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.

[29] 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.

[30] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1186–1195.

[31] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 466–481.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*. USENIX Association, 35–49.

[33] Hadi Jooybar, Wilson WL Fung, Mike O'Connor, Joseph Devietti, and Tor M Aamodt. 2013. GPUDet: a deterministic GPU architecture. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 1–12.

[34] 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.

[35] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[36] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about eve:{Execute-Verify} replication for {Multi-Core} servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 237–250.

[37] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[38] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *EuroSys*. ACM, 25:1–25:17.

[39] Ramakrishna Kotla and Michael Dahlin. 2004. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004.* IEEE, 575–584.

[40] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When private blockchain meets deterministic database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–28.

[41] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[42] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*. USENIX Association, 467–483.

[43] Zhongmiao Li, Paolo Romano, and Peter Van Roy. 2019. Sparkle: Speculative deterministic concurrency control for partially replicated transactional stores. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 164–175.

[44] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. 2019. MgCrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment* 12, 5 (2019), 597–610.

[45] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1156–1168.

[46] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading. In *SOSP*. ACM, 327–336.

[47] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. 2014. Efficient deterministic multithreading without global barriers. *ACM SIGPLAN Notices* 49, 8 (2014), 287–300.

[48] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database.

[49] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–13.

[50] Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. 2019. Lazy determinism for faster deterministic multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 879–891.

[51] Microsoft. 2023. verona-rt: The runtime for the Verona project. https://github.com/microsoft/verona-rt. Accessed: December 2, 2023.

[52] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 479–494.

[53] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 517–532.

[54] Omar S Navarro Leija, Kelly Shiptoski, Ryan G Scott, Baojun Wang, Nicholas Renner, Ryan R Newton, and Joseph Devietti. 2020. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 167–182.

[55] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[56] National Academies of Sciences, Policy, Global Affairs, Board on Research Data, Information, Division on Engineering, Physical Sciences, Committee on Applied, Theoretical Statistics, Board on Mathematical Sciences, et al. 2019. *Reproducibility and replicability in science*. National Academies Press.

[57] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 97–108.

[58] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX Association, 305–319.

[59] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[60] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 325–341.

[61] Alex Pshenichkin. 2024. So you think you want to write a deterministic hypervisor? https://antithesis.com/blog/deterministic_hypervisor/. Accessed: May 14, 2024.

[62] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[63] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[64] Mike Qin, Zhiqi He, Tudor Brindus, and Mohammad Nasirifar. 2021. felis GitHub Repository. https://github.com/uoft-felis/felis. Accessed: December 2, 2023.

[65] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*. 1583–1598.

[66] M Aater Suleman, Moinuddin K Qureshi, Khubaib, and Yale N Patt. 2010. Feedback-directed pipeline parallelism. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 147–156.

[67] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. 2020. Meerkat: Multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–14.

[68] The MystenLabs Team. 2023. The sui smart contracts platform. https://docs.sui.io/paper/sui.pdf. Accessed: May 14, 2024.

[69] 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.

[70] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. 2023. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. In *OSDI*. USENIX Association, 441–459.

[71] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: lightweight parallel logging for in-memory database management systems. (2020).

[72] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain.

[73] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515.

[74] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2939–2952.

A Artifact

DORADD artifact is available at https://zenodo.org/records/14607089 and https://github.com/doradd-rt/ppopp-artifact. It includes the source code, detailed instructions, and scripts necessary for reproducing the experiments presented in the evaluation section. The code has been tested primarily on Ubuntu 22.04. To fully reproduce the results, two testbeds are required.

A.1 Testbeds

Single-node experiments (Figures 6, 9, and 10):

- These experiments should be run on a local testbed equipped with an Intel Xeon Gold 5318N CPU (24 cores) and 128GB DRAM.

Multi-node experiments (Figures 7 and 8):

- These experiments are designed to run on CloudLab using three `d6515` nodes.
- We provide a CloudLab experiment profile to facilitate the setup of the machines.
- You can follow this guide to set up CloudLab.
- Please reserve these nodes in advance, as they may not always be available.

A.2 Experiments Summary

Estimated time to run all experiments: 5h – 6h. Approximate human time required: 1h. We suggest using tmux to track the experiment progress.

Instructions	Testbed	Human/Machine time
Figure-5.md	Local	5 min/2h
Figure-6.md	CloudLab	10 min/2h
Figure-7.md	CloudLab	10 min/20 min
Figure-8.md	Local	5 min/30 min
Figure-9.md	Local	5 min/5 min

Received 16 August 2024; accepted 11 November 2024