Event Horizon: Asymmetric Dependencies for Fast Geo-Distributed Operations

Jonathan Arns
KTH Royal Institute of Technology
jonathan.arns@gmail.com

Harald Ng
KTH Royal Institute of Technology
hng@kth.se

Kyriakos Psarakis*
Ververica GmbH
kyriakos.psarakis@ververica.com

Asterios Katsifodimos
Delft University of Technology
a.katsifodimos@tudelft.nl

Paris Carbone
KTH Royal Institute of Technology
parisc@kth.se

ABSTRACT

Low-latency geo-distributed applications currently face the barrier of cross-site coordination for ensuring state consistency. Existing mixed-consistency models leverage the existence of strongly- and weakly-consistent operations in a given application, to avoid coordination whenever possible. However, existing approaches are rather pessimistic, coordinating more than is necessary.

In this paper, we introduce *Semi-Linearizability* (SL): a consistency model that executes application operations with linearizability guarantees only when strictly necessary, avoiding over-coordination. Specifically, we propose novel operation semantics that can encode ordering relationships between application operations and map them to coordination primitives. Our proposed semantics can be used to reason over latent, asymmetric dependencies between different operations and optimize their execution. We show how SL enables a new class of safe, uncoordinated operations that previous models would otherwise execute under globally strict order, while offering substantial performance gains without violating application invariants. To demonstrate the advantages of SL, we implemented DeMon, a system that achieves four orders of magnitude lower latency on the most frequent operation in the widely used RUBiS benchmark compared to state-of-the-art systems.

1 INTRODUCTION

Geo-replicated applications, such as extended reality (XR) and immersive gaming [1, 2], are rapidly evolving to support increasingly interactive workloads that demand both low latency for responsiveness and strong consistency for critical shared state (e.g., asset ownership) [3]. Achieving low latency for such applications remains a fundamental challenge. At one end, strictly serializable distributed databases such as Spanner [4] pose a fundamental latency barrier because they enforce a total ordering of operations through *coordination*: replicas must first communicate over the network to agree on how the application state should be modified (using protocols such as Paxos [5] and 2PC [6]). At the opposite end, weaker consistency models (e.g., monotonic operations in CALM [7] or CRDTs [8]) avoid coordination but do not support common application invariants, such as uniqueness or stable decision outcomes, in the face of concurrency.

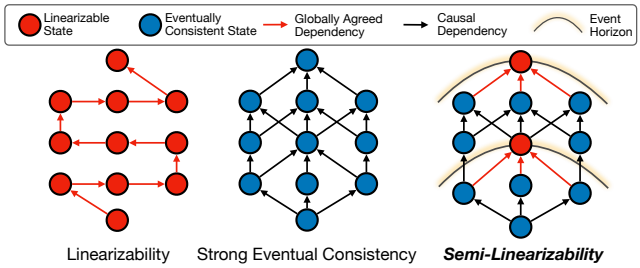

Figure 1: Bridging consistency extremes. Unlike Linearizability (left) and Strong Eventual Consistency (center), Semi-Linearizability (right) offers a hybrid approach that permits concurrent execution until a critical operation forms an *event horizon*, collapsing divergent histories into a unified state only when necessary.

Both strong and weak consistency models (illustrated in Figure 1) are too extreme, failing to recognize that within a single application, strict linearizability is often only required for specific state transitions, while others can safely tolerate temporary divergence. The tension between these extremes has instead driven various efforts to preserve application invariants by selectively applying one of these two models based on the operation. *Invariant confluence* [9] formally identifies which operations require coordination to

*Work done while at Delft University of Technology

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

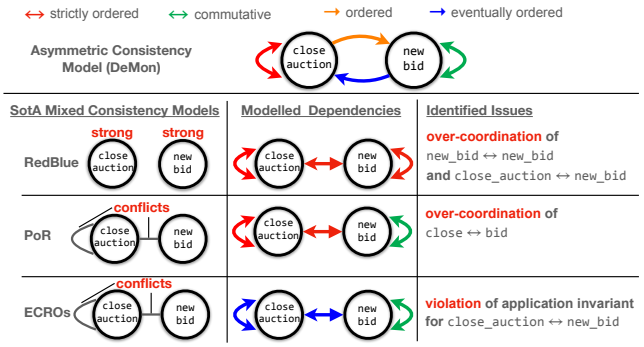

Figure 2: Asymmetric dependency vs. existing hybrid consistency models.

preserve invariants and which can safely avoid it. *Hybrid consistency* models allow the application to define which operations require coordination and handle them differently. For example, RedBlue consistency [10] separates between non-commutative (red) and commutative (blue) operations, such that only the red operations are totally ordered. Similarly, PoR [11] and ECROs [12] employ conflict-based reasoning to determine when coordination is strictly required.

While these hybrid approaches successfully break the dichotomy between strong consistency and coordination avoidance, they are limited by a binary view of "conflict". By categorizing operations as either requiring full coordination or none at all, they fail to capture the nuance of workload semantics that fall between these extremes. Crucially, they assume that conflict relations are symmetric, ignoring the fact that many real-world application semantics are asymmetric, necessitating ordering only in one direction. Consequently, these models can suffer from over-coordination.

To understand the inherent asymmetry, consider the canonical example of distributed auctions [13] in Figure 2. This involves two operations: `new_bid` inserts a bid entry, whereas `close_auction` completes an auction, establishing the winning bid. Essentially, `new_bid` operations commute with each other but must be observed by a subsequent `close_auction`. Existing models like I-Confluence, RedBlue, and PoR treat this dependency symmetrically, requiring either total-order ↔ or non-imposed order ↔ between the operations. Enforcing total order necessitates excessive coordination to serialize all `new_bid` and `close_auction` operations, whereas non-imposed order risks invariant violations where a winning bid is omitted at the time of `close_auction`, awarding the sale to another bidder.

The fundamental idea behind this work is to leverage *directional* dependencies to capture invariants. Specifically, we observe that `new_bid` operations must eventually be included in an auction or rejected, but do not immediately impose order on the auction closure: `new_bid` → `close_auction`. This implies that bid operations can execute concurrently without any coordination. However, `close_auction` must impose order on the set of preceding bids (i.e., observe the same set of bids at each replica) to consistently elect the same winner: `close_auction` → `new_bid`. As such, only `close_auction` needs to be coordinated with relation to the bids. After the auction has been closed, bids can be ignored or rejected without violating correctness. In this work, we expand on this observation to show how to capture application intent more precisely with directional dependencies, i.e., `close_auction` determines the outcome based on the causal history of `new_bids`, but not vice versa. This asymmetry enables more fine-grained semantics, new consistency models, and systems with reduced coordination.

In summary, we advocate for establishing richer operation dependency semantics to enable mixed consistency models that can effectively exploit implicit coordination. Our key contributions are:

- We formalize *asymmetric operation dependencies* (§2) to express application invariants more precisely, moving beyond binary conflict views to capture directional ordering requirements.
- We introduce *semi-linearizability* (§3), a new mixed consistency model illustrated in Figure 1. This model allows relaxed dependencies to commute until a stronger operation forms an "event horizon": a boundary that collapses them into a globally agreed order, enforcing synchronization only when strictly necessary.
- We present *DeMon* (§4), a runtime execution framework that realizes this model by committing dependencies in a global log using bags. In our evaluation (§5), we demonstrate that DeMon significantly reduces coordination overhead while maintaining correctness.
- Finally, we discuss the implications (§6) and challenges of further evolving this vision into large-scale systems with implicit coordination (§7).

In astrophysics, an *event horizon* of a black hole marks the frontier beyond which all events, including the motion of matter, radiation, light and information, can no longer diverge from an inward path. No matter how chaotic the trajectories leading up to that boundary, everything that crosses it, is committed to an irreversible progression toward the singularity, where all possible future paths collapse. Similarly, distributed applications generate large volumes of concurrent events whose order is often unconstrained, yet, some operations must eventually cross a frontier of agreement, i.e., an *event horizon*, where their outcome and their causal dependencies collapse into a fixed, durable state. In Semi Linearizability, commutative operations flow freely until a relatively stronger operation "pulls them" into an immutable order.

2 ORDERING SEMANTICS

Over the past decade, database research has increasingly focused on aligning coordination with application-level semantics rather than enforcing uniform, system-wide guarantees. A well-known example in ACID databases is invariant confluence [9], which reasoned about safe execution without coordination by analyzing the conditions under which user-defined application invariants are preserved. The same perspective has influenced a range of systems that leverage read-write sets, commutativity, and operation dependencies to minimize synchronization overhead [10–12, 14–17]. These approaches assume symmetric relationships between operations, either requiring total order or assuming full independence. In practice, many applications involve directional dependencies, where one operation must observe another in a correct order, but not vice versa. For example, an auction has to observe the bids that have been submitted *before* closing the auction (termed *happened-before* relation [18]). At the same time, when a user bids, the application developer may not require a bid to be submitted before closing the auction. This is because the user can simply be notified later if their bid did not make it on time.

2.1 Asymmetric Operation Dependencies

Let us consider a more generalized model for operation dependencies, that is, a directed graph $G = (V, E)$ where each vertex $v \in V$ represents one operation and each edge $(v \rightarrow v') \in E$ represents an ordering invariant between two operations v and v'. To illustrate the different types of such dependencies depicted in Figure 2, we adopt a subset of operations from the RUBiS application [13] that is used to benchmark mixed consistency models [10–13]. Again, we use `new_bid` and `close_auction`. When an auction closes, a common invariant is that the winner of an auction, identified by the maximum bid, cannot change after `close_auction` is executed. Existing works would simply mark `new_bid` and `close_auction` as *strong* operations, which must be strictly ordered to ensure that no auction outcome can change during concurrent execution [10, 11, 14–17, 19, 20]. Instead, we now consider four types of inter-dependencies between these two operations and use the example illustrated in Figure 2 to discuss these dependencies.

1. Strictly Ordered Dependency: $[V \leftrightarrow V]$ A symmetric dependency between operations that require linearizable real-time order. i.e., if $v_1 \leftrightarrow v_2$ then it is expected that in any valid execution schedule, all invocations of v_1 and v_2 must be totally ordered in respect to each other.

In the example of Figure 2, a strictly ordered dependency exists between `close_auction` operations, i.e., they must be totally ordered with each other, as concurrent `close_auction` operations could choose different winners for the same auction. Strictly ordered dependencies have been supported through strong coordination mechanisms, such as atomic registers and consensus [5, 21]. In existing mixed consistency models these dependencies can be captured either by declaring cross-operation conflicts (PoR [11] and ECROs [12]), or strong to strong dependencies (RedBlue [10]).

2. Commutative Dependency: $[V \leftrightarrow V]$ A symmetric dependency requiring two operations to be fully commutative, such that they do not need to be ordered.

In our example, `new_bids` are commutative with each other. Thus, there is no need to order `new_bid` operations with respect to other `new_bid` operations. While purely commutative operations are trivial to support, there has been active research under the topic of (Strong) Eventual Consistency [8, 22–24], CRDTs [25], and CALM [26]. These lines of work exploit weaker properties between operations, such as monotonicity of updates, to provide commutativity and eventual convergence. Existing conflict-centric mixed consistency models can capture commutative dependencies in the absence of a conflict (PoR [11] and ECROs [12] in Figure 2).

3. Ordered Dependency: $[V \rightarrow V]$ An asymmetric dependency that arises when ordering invariants need to be strictly preserved with respect to one of the operations. Given $v_1 \rightarrow v_2$, if an operation of v_1 reads one of v_2 on one server, then it must do so on all servers. We say that a reads b, if b writes a value that is later read by a.

An intuitive way to think of an ordered dependency is the relation between `close_auction` and `new_bid`. Closing an auction signifies a winner based on the set of executed bids. The auction winner cannot change and must be the same across all replicas. It is crucial that when `close_auction` is executed on a remote replica, it reads the same set of `new_bids` that were read on the initiating replica. To the best of our knowledge, this type of dependency is not considered by existing consistency models, and systems fall back to treating this as a symmetric, strictly ordered dependency.

4. Eventually Ordered Dependency: $[V \rightarrow V]$ An asymmetric dependency that requires two operations to be eventually ordered. Given $v_1 \rightarrow v_2$, each invocation of v_1 must eventually read the same set of invocations of v_2 on all servers. To achieve this, it is permitted to reorder invocations of v_1.

In our example, in contrast to `close_auction` operations, `new_bid` is allowed to be marked as invalid by the application if the respective auction was closed concurrently on a different replica. This can happen when the `new_bid` was not applied at the `close_auction` operation's origin replica. The `new_bid` is reordered after the `close_auction` on all replicas, invalidating it. Eventually ordered dependencies have been explored in the context of ECROs [12], however,

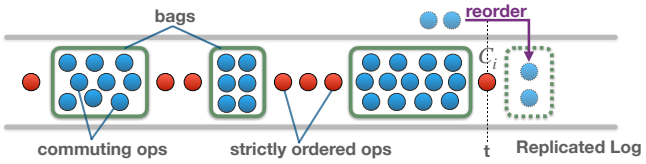

Figure 3: Bags of commutative operations committed at a fixed position between strictly ordered operations. Operations executed concurrently to C_i are re-ordered to the next bag when C_i commits at time t.

that model can express only symmetric eventually ordered dependencies. ECROs allow violating the ordered invariant safety, limiting its applicability when correctness is required by invariants.

3 A MODEL OF SEMI-LINEARIZABILITY

Based on our insights gained from asymmetric operation dependencies, we define a practical consistency model that captures these dependencies accurately. Traditionally, the binary classification of operations into strong or weak only captures symmetric dependencies. We extend it to also express both types of asymmetric dependencies. Specifically, strictly ordered between strong operations ($s \leftrightarrow s'$), commutative between weak operations ($w \leftrightarrow w'$), ordered from strong to weak ($s \rightarrow w$), and eventually ordered from weak to strong ($s \leftarrow w$). This way, the interaction between weak and strong operations can express the asymmetry of ordered and eventually ordered dependencies.

System model. Replicas are state machines that, upon receiving an operation v, can either (a) apply v directly to their state or (b) undo the previous x operations, apply v, and re-apply the x operations. In this case, v now counts as applied before those x operations. This reordering of the x operations is needed to implement eventually ordered dependencies ($v \rightarrow v'$).

Semi-linearizable order. As shown in Figure 3, the semi-linearizable order represents a totally ordered log that consists of strictly ordered strong operations and bags of commutative ($w \leftrightarrow w'$) weak operations between them. We define the semi-linearizable order as a partial order $O = (U, \prec_o)$ over the union of all strong and all weak operations $U = S \cup W$. To represent strictly ordered dependencies ($s \leftrightarrow s'$), $\prec_o$ orders all strong operations with each other according to their linearizable real-time order. To represent ordered dependencies ($s \rightarrow w$), $\prec_o$ orders each weak operation w into some bag after all strong operations that happened before w. So, relative to all strong operations, w is ordered in the position of that bag. Since causality is often required for CRDT-based commutative operations [8], we also define weak operations to be causally ordered with each other.

	$S \leftrightarrow S$	$S \leftrightarrow W$	$W \leftrightarrow W$
RedBlue [10]	LIN	CC	CC
PoR [11]	LIN*	CC	CC
Well-Coordination [16]	LIN*	CC*	CC*
OAC [14]	LIN	LIN	CC
Semi-Linearizability	LIN	SEQ→LIN	CC

Table 1: The consistency of strong (S) and weak (W) operations under different mixed consistency models. LIN: Linearizability [27]; SEQ: Sequential Consistency [28]; CC: Causal Consistency [29]. *PoR and Well-Coordination symmetrically relax these consistency levels for non-conflicting operation pairs.

Semi-Linearizability. A replicated system satisfies Semi-Linearizability given a semi-linearizable order $O = (U, \prec_o)$, if for every replica p, the following two conditions hold:

(1) At any time, all operations that p has applied are applied in the order that they have in O, i.e., p respects the order that is enforced by the dependencies set as application invariants.
(2) Before applying any strong operation s, p has previously applied all operations that are ordered before s, as enforced by O.

Together, these two conditions guarantee that *i*) strong operations are strictly ordered ($s \leftrightarrow s'$); *ii*) strong operations are never reordered and are always ordered exactly after all previous weak operations ($s \rightarrow w$) and; *iii*) weak operations are eventually ordered with respect to strong operations ($w \rightarrow s$). Therefore, coming back to the `new_bid` and `close_auction` operations, we must mark the `close_auction` operation as strong. Contrary to existing mixed consistency models, we can safely mark the `new_bid` operation as weak and allow its execution without coordination.

This model guarantees sequential consistency for strong operations reading weak writes. Upon their initial execution, weak operations also read strong writes with sequential consistency. However, as seen in Figure 3, after applying a strong operation (e.g., through consensus), weak operations may eventually be reordered to read that strong operation with linearizable consistency (SEQ→LIN). Table 1 shows SL as the only consistency model that unifies three distinct levels of consistency based on ordering restrictions. The existing models of RedBlue [10], PoR [11], Well-Coordination [16], and OAC [14] all come down to a binary split between Linearizable and Causal Consistency, leaving no room in-between. While Explicit Consistency, which is defined purely based on invariant preservation, allows a notion of reordering operations to reduce coordination, no system implementing it supports the asymmetric operation dependencies that SL captures [12, 30, 31].

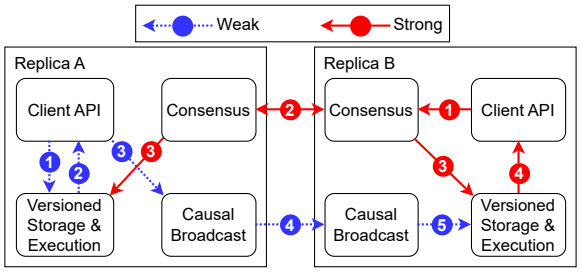

Figure 4: The separate paths of strong and weak operations through DeMon.

4 DEMON

To demonstrate the potential of Semi-Linearizability, we implemented DeMon, a geo-replicated in-memory storage system. The goal of DeMon is to minimize coordination by leveraging the asymmetric dependency between weak and relatively stronger operations. To achieve this, DeMon uses a consensus protocol to replicate strong operations and reliable causal broadcast for weak operations. This enables DeMon to achieve the strict ordering needed for strong operations while enabling weak operations to be executed locally without coordination, similar to CRDTs. The semi-linearizable order further requires ordering between strong and weak operations: any weak operation w must be ordered after all strong operations that happened before w (§3). DeMon guarantees this through watermarks that act as synchronization barriers, ensuring weak operations preserve the correct causal dependency to strong operations.

4.1 Flow of Operations

DeMon uses a consensus protocol (OmniPaxos [32]) to replicate a log that defines the strict order for strong operations. Weak operations are disseminated with reliable causal broadcast to preserve the causal ordering required for CRDT-based operations. Furthermore, every replica maintains its own counter of weak operations. These are used to form vector clocks, which serve as watermarks. We explain the watermark and the different paths of strong and weak operations in DeMon using Figure 4.

Weak operations are handled as follows. 1 The replica receiving the weak operation from the client immediately executes it. 2 Once executed, a response is sent to the client, similar to op-based CRDTs [8]. 3 Then, the operation is asynchronously replicated via causal broadcast. 4 The causal broadcast delivers messages in causal order. 5 Upon delivery at a remote replica, the weak operation is executed. To track replication, replicas also periodically broadcast the ids of the latest weak operations they have received. Based on this, each replica keeps a record of which weak operations have been replicated to a majority quorum of replicas.

Strong operations. 1 Upon receiving a strong operation, the replica attaches a vector clock summarizing its knowledge of every replica's latest weak operation that has been seen by a quorum. This vector clock represents the low watermark of weak operations that must be ordered before the strong operation. Together with the watermark, the operation is then proposed in the consensus protocol. 2 The operation is decided as an entry in the consensus log, requiring synchronous communication. 3 Upon receiving a decided strong operation and its watermark in the consensus log, each replica calculates the latest watermark by merging the new watermark with the previous. The latest watermark defines the stable state that all new operations are applied on top of. If a replica observes that the latest watermark does not include some weak operations it had already executed, these need to be reordered and executed on top of the stable state. We describe an example of such a scenario next. 4 Finally, a response is sent to the client after execution.

Example. Consider three replicas A, B, and C managing a distributed auction. A first accepts `new_bid(100)`, while B concurrently accepts `new_bid(200)`; both execute locally and broadcast the operations to the other replicas asynchronously. Eventually, these updates propagate to all replicas, which converge on the highest bid of 200. Now, assume C issues `new_bid(300)`. However, before this operation reaches any other replica, B issues `close_auction`. With the operation, B attaches the watermark $[a_1, b_1, c_0]$ to the consensus entry that excludes the unseen bid from C. When `close_auction` is decided via consensus, it enforces the state specified by that watermark, finalizing the auction with `new_bid(200)` as the winner. We consider the two possible cases for `new_bid(300)` at a replica: If a replica receives `new_bid(300)` before the `close_auction`, it has been applied to the local state and needs to be rolled back (see §4.2). Conversely, if `new_bid(300)` arrives after the closure, it is simply applied on top of the finalized state. In both scenarios, `new_bid(300)` is effectively ordered after `close_auction`, rendering it invalid.

4.2 Reordering Operations

Semantically, reordering a weak operation w that is already applied to the local state when a strong operation s is executed is equivalent to rolling back w, applying s to the state, and then applying w again. To avoid the need for an inverse implementation of each operation, DeMon keeps two versions of the state. The stable state has all previous strong operations and the weak operations between them; the unstable state has additionally applied all recent weak operations that the replica has received. Weak operations are initially applied only to the unstable state. However, when executing a strong operation s, first, any newly stable weak operations

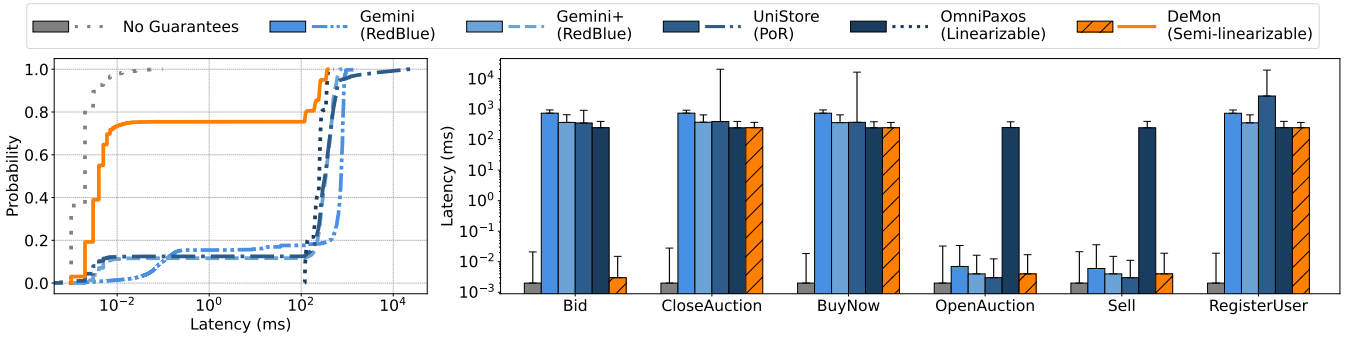

Figure 5: Cumulative latency distribution (left) and latency per transaction (right) for RUBiS in 5 regions on the client side.

in s's watermark are applied to the stable state. The replica may need to wait if it has not received these new weak operations. Then, s is applied to the stable state. Finally, i) the unstable state is updated by generating a delta of the changes done by s and applying that delta to the unstable state and; ii) all recent weak operations that were overwritten by the delta, are applied to the unstable state again.

5 EVALUATION

We evaluate DeMon in a geo-replicated setting, using a non-negative counter microbenchmark and the RUBiS [13] benchmark. While the primary endpoint is low latency, we also evaluate throughput and specifically compare coordination overhead between protocols.

Hardware. We use a setup that spans five regions: US-East, Finland, Brazil, US-West, and Singapore. We measured the mean round-trip latencies between them, which range from 73.7ms (US-East ↔ US-West) to 387.6ms (Brazil ↔ Singapore). For protocols that use a primary replica, the primary region is set to US-East. In each region, one DeMon replica is deployed on a Google Compute Engine instance of type n2-standard-4 with 4 vCPUs and 16GB of Memory. The clients run as lightweight tasks on the same instances to remove network delays between clients and replicas.

Baselines. For the performance evaluation, we compare DeMon to five reference protocols, which were all implemented in the same Rust codebase as DeMon (code at https://github.com/JonathanArns/demon).

- **Gemini** is an implementation of the original Gemini protocol for RedBlue consistency [10]. Note that Gemini does not guarantee fault-tolerant writes.
- **Gemini+** is a fault-tolerant implementation of RedBlue consistency, which uses OmniPaxos [32] for establishing total order among strong operations and a similar deterministic watermarking strategy as DeMon to establish partial order with weak operations.
- **UniStore** implements Partial Order-Restrictions consistency (PoR) via an optimistic commit protocol similar to UniStore [20].
- **OmniPaxos** [32] is a consensus protocol and included as a baseline to evaluate the performance required to achieve strong consistency. It follows the typical Paxos pattern with a primary (leader) that needs to coordinate with a majority quorum to commit an operation.
- **No Guarantees** represents the ideal performance of no coordination. It is similar to op-based CRDTs where operations are disseminated to other replicas using causal broadcast and completed once executed locally. As we describe next, it is not safe for workloads with non-commutative operation dependencies.

RUBiS. We use the standard RUBiS benchmark, which consists of five operations: `BuyNow` and `RegisterUser` are strong operations with strictly-ordered self-dependency, while the other (`Bid`, `OpenAuction`, and `Sell`) are weak operations. As is common in related work, we extended RUBiS with the `CloseAuction` operation [11, 12, 20] that has an asymmetric dependency to `Bid` (Figure 2). The benchmark generates an operation distribution based on the update portion of the bidding mix. Crucially, the `Bid` operation, which can only be weak under Semi-Linearizability, makes up 60% of the update operations. We focus on update operations, as read-only operations can be local and wait-free in all protocols.

Figure 5 shows the cumulative latency distribution of the RUBiS workload for each protocol, as well as a breakdown of the latency by operation type. Especially in the latency distribution, DeMon's advantage of the frequent `Bid` operation being weak is visible, offering sub-millisecond latency for over 75% of the workload. For the other RUBiS operations, DeMon does not offer such drastic gains but still matches OmniPaxos in latency for the strong `CloseAuction`, `BuyNow`, and `RegisterUser` operations at 245ms median latency. This latency stems from forwarding strong operations to the primary and deciding them via consensus for a total of two

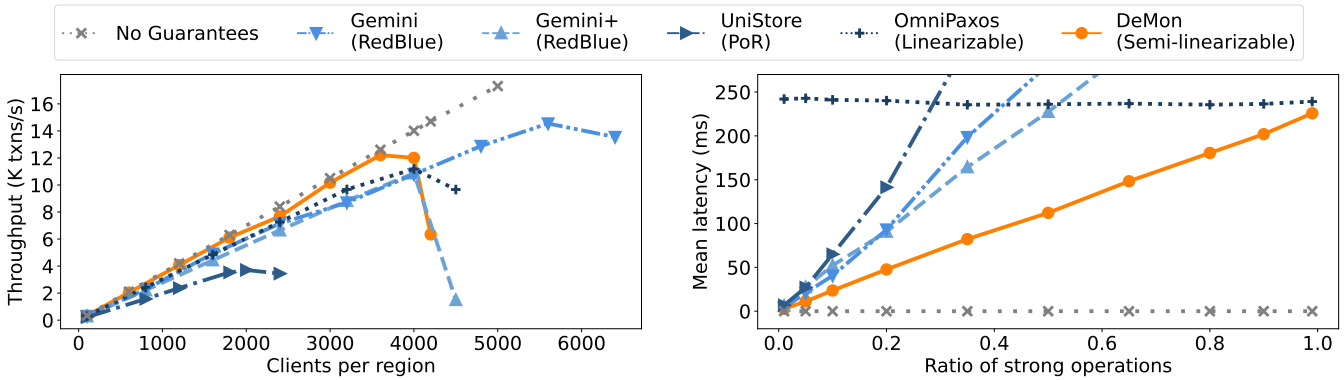

Figure 6: RUBiS throughput/client count (left); Latency/strong op. ratio in a non-negative counter (right).

message round-trips. In contrast, the other hybrid consistency protocols with a primary writer incur at least one additional round trip of latency for these operations, since they must additionally wait for new weak operations to become quorum-replicated before forwarding strong operations to the primary [20], which manifests in 371ms and 391ms median latency for CloseAuction in Gemini+ and UniStore respectively. For the commutative `OpenAuction` and `Sell` operations, all protocols except OmniPaxos offer sub-millisecond latency even at the 99th percentile.

Figure 6 (left) shows that DeMon reaches a marginally higher maximum throughput than all other fault-tolerant protocols. Notably, the protocols using consensus (Gemini+, UniStore, Demon) for replication are limited by this in their throughput, as they are limited to a single primary writer. When the primary gets overloaded, throughput drops off sharply. Since DeMon is able to safely use causal broadcast for more of its workload (`Bid` operations), it achieves higher throughput, even with the additional computational overhead of reordering operations. Gemini achieves a higher maximum throughput than DeMon by omitting coordination primitives for fault tolerance; however, this results in 3 times higher median latency (737ms vs 245ms) for strong operations. Gemini's high latency stems from replicas waiting for mutually exclusive write access to invoke strong operations instead of sending strong operations directly to a primary that can always write.

Coordination Overhead. To evaluate the impact on performance for coordinating strong operations further, we use a microbenchmark for replicating non-negative counters. The datatype has two operations, `add` and `subtract`, with the invariant that the counters may never be less than zero. `Subtract` is unsafe and has a strictly ordered dependency on itself. All hybrid consistency models in our comparison guarantee safety by coordinating `subtract` as a strong operation while treating `add` as a weak operation that can be executed locally. This enables us to control the amount of coordination by tuning the ratio of `subtract`.

As shown in Figure 6 (right), Demon exhibits a 'pay-as-you-go' performance characteristic: its mean latency scales proportionally with the ratio of strong operations. This contrasts with the baselines: regardless of the operation mix, OmniPaxos incurs a fixed coordination cost that involves forwarding requests to the leader followed by a quorum round-trip, resulting in a constant latency of 245ms. On the other extreme, No Guarantees acts as a lower bound with sub-millisecond latency. In contrast, the other hybrid protocols surpass the latency of OmniPaxos before or at 50% strong operations. This is because RedBlue and PoR consistency require strong operations to wait until all causally related operations are durable, resulting in an additional round trip of synchronous communication. For DeMon, the only coordination overhead for strong operations is in its underlying consensus protocol.

6 DISCUSSION

Analyzing asymmetric operation dependencies poses an opportunity for new consistency models, such as Semi Linearizability that is proposed in this paper, to offer large reductions in coordination without sacrificing application invariants. The resulting latency reductions can enable interactive applications at a geo-distributed scale. We demonstrate this possibility and present further research directions.

Dependency Types: Our classification of operation dependencies is powerful yet open-ended. Further extensions can be made based on different workloads that might exhibit different distinct types of dependencies. Identifying these dependencies can be crucial to achieving further optimizations in mixed consistency models.

Reordering Semantics. Under SL, reordering weak operations may lead to unexpected behaviors, such as bids in an auction being revoked. Only once an operation is committed behind the event horizon by a strong operation, can the system guarantee stability. Depending on the application, the difference between the fast committed state and the stable committed state of weak operations should be

exposed via the user interface to prevent surprises. Akin to semantic trade-offs between CRDT algorithms (e.g. OR-set vs 2P-set [25]), there may also be multiple semantics of reordering with different trade-offs to choose between. Imagine a replicated set that can be `reset` via a strong operation. Whether concurrent `insert` operations should take effect after the `reset` depends on the application.

Application Analysis. At the individual application level, identifying dependencies between operations is a complex task that can be challenging for developers. It is crucial not only to understand the possible interactions between all operations, but also the application's set of invariants. In the auction example, it is clear that an auction's winner should never change once determined, but rejecting bids after the fact is an inconvenience that might be acceptable. Had we deemed it unacceptable, there would not have been an asymmetric dependency, and no potential for coordination avoidance. Static analysis tools can automate difficult parts of this process (e.g., binary conflict identification [19]) and help in identifying these tradeoffs, but human input might ultimately still be needed to decide them.

Decentralized Systems. Beyond interactive applications, SL might be a key step towards efficient byzantine fault-tolerant (BFT) systems by minimizing the use of slow BFT consensus [33] in favor of fast BFT CRDTs [34].

7 A FUTURE OUTLOOK

The formalization of Semi-Linearizability and asymmetric dependencies provides more than just a consistency model; it offers the basis towards an intermediate representation (IR) for composing mixed-consistency distributed applications. Having a detailed asymmetric dependency graph representation of an application makes distributed code generation a nearly deterministic process. However, a natural question that arises is: "how do we get there?" As illustrated in Figure 7, we envision several ways to bridge the gap between user intent (implicit or explicit), modelling and code generation. Choosing the right avenue (i.e., natural language-, formal-, DSL- or source code-based) depends on the level of domain expertise of the user or creator of the application. Implementing each avenue is subject to different associated research challenges, that we detail below.

Natural Language Specs for Non-Experts. We currently witness a movement towards empowering non-experts with the ability to prompt using natural language applications and systems [35, 36]. Informal requirement gathering from free narratives often lacks the precision required to specify the right level of distributed safety. Furthermore, while LLMs can use natural language to generate code, they often fail to guarantee that the generated logic adheres to strict safety invariants without hallucinations [37]. The primary

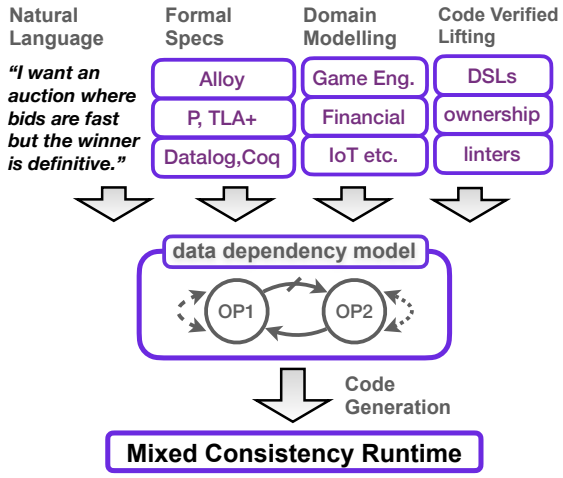

Figure 7: The case of asymmetric operation dependencies as an intermediate representation for automating or assisting the creation of distributed applications.

research challenge here is to verify whether a set of asymmetric dependencies extracted from "vibe coding" or LLM-User interactions is complete enough to build adequately safe systems without resorting to strict linearizability. This necessitates an iterative process, where the AI identifies ambiguity, such as specifically asking about edge cases in an auction closing and a relative taxonomy of operations, to guide the user towards a complete symbolic specification that is both necessary and sufficient.

Model Translation for Domain-Experts. Beyond the realm of databases, a plethora of domain models are used to capture dependencies. For example, game engines like Unity and Unreal Engine which become prevalent for composing XR applications already feature models that capture nuanced user-defined interactions with shared virtual objects. Game engines generate netcode based on these models which synchronizes actions and state across game clients and servers [38, 39]. Many other model semantics could be lifted into asymmetric dependencies. For instance, BPMN 2.0 models business processes [40] where sequence flows dictate strict causal ordering while parallel gateways imply commutativity. Similarly, in infrastructure management, Terraform graphs [41] in infrastructure specifications, construct directed acyclic graphs (DAGs) via explicit `depends_on` attributes. Future research could target these rich semantic sources, formalizing the mapping between their static structures, such as GraphQL [42] parent-child nested fields, and dynamic constraints required for safe, concurrent execution.

Specification Languages for Experts. Distributed Systems experts currently rely on complex formal verification languages such as TLA+ [43] (and model checkers like TLC [44])

that require a comprehensive "god-mode" view of the entire system state to verify global correctness. These tools offer powerful checks but impose a high cognitive burden and often result in opaque specifications that are difficult to modify. We envision deducing a simpler and more approachable specification language that focuses solely on visible orderings and coordination costs. A key research challenge here is to replace both manual protocol verification and code synthesis. By specifying a cost model and a set of local constraints for coordination, a min-cost optimizer could synthesize a protocol that satisfies given constraints, effectively bridging the gap between abstract specifications, formal proofs and practical, optimized implementations.

Verified Lifting for Developers. Developers currently implement distributed logic by manually injecting coordination primitives from existing frameworks and systems. This manual approach is often pessimistic, leading to over-coordination, or even unexpected anomalies if dependencies are missed. To materialize a better solution, we must advance static code analysis techniques to support *verified lifting* as showcased in other solutions such as Katara [45]. In this case, the goal is to infer the dependency graph automatically by analyzing read/write patterns and commutative operations within a DSL. This would allow the runtime to automatically inject the necessary coordination by mapping detected strong dependencies to consensus and weak dependencies to causal broadcast, shifting the burden of correctness from the programmer to the compiler. New programming languages and frameworks such as Hydro [46], Aqua [47], Lasp [48], Styx [49] and Portals [50], among others build towards this trajectory.

8 CONCLUSION

The rigid dichotomy between strong and weak consistency has long forced developers into a compromise: incur the high latency of coordination for strict ordering or constrain application logic to operations that are commutative or monotonic. In this paper, we showed that operation dependencies are more nuanced: operations often exhibit asymmetric dependencies that can be leveraged to reduce coordination. To this end, we introduced Semi-Linearizability (SL), a consistency model where commuting operations can diverge until a relatively stronger operation enforces an "event horizon", that is, a boundary of coordination. This minimizes coordination overhead while strictly preserving the correctness invariants enforced by the application.

The insight of asymmetry provides a foundation for building systems with more precise consistency guarantees. By leveraging techniques such as static analysis and AI-assisted development to identify these directional relationships, we can move beyond coarse-grained models to construct systems that enforce the exact constraints of the application. This paves the way for a new generation of distributed architectures where the gap between user intent and runtime execution is bridged by automated synthesis, ensuring systems are both correct by construction and efficient by default.

REFERENCES

[1] M. Xu, W. C. Ng, W. Y. B. Lim, J. Kang, Z. Xiong, D. Niyato, Q. Yang, X. Shen, and C. Miao, "A Full Dive Into Realizing the Edge-Enabled Metaverse: Visions, Enabling Technologies, and Challenges," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 656–700, 2023.

[2] M. Ali, F. Naeem, G. Kaddoum, and E. Hossain, "Metaverse Communications, Networking, Security, and Applications: Research Issues, State-of-the-Art, and Future Directions," *IEEE Communications Surveys & Tutorials*, vol. 26, no. 2, pp. 1238–1278, 2024.

[3] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed virtual actors for programmability and scalability," *MSRTR2014*, vol. 41, 2014.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Transactions on Computer Systems*, vol. 31, pp. 1–22, Aug. 2013.

[5] L. Lamport, "Paxos Made Simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001.

[6] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, (Berlin, Heidelberg), p. 393–481, Springer-Verlag, 1978.

[7] J. M. Hellerstein and P. Alvaro, "Keeping CALM: when distributed consistency is easy," *Communications of the ACM*, vol. 63, pp. 72–81, Aug. 2020.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*, Springer, 2011.

[9] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, pp. 185–196, Nov. 2014.

[10] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making {Geo-Replicated} Systems Fast as Possible, Consistent when Necessary," pp. 265–278, 2012.

[11] C. Li, N. M. Preguiça, and R. Rodrigues, "Fine-grained consistency for geo-replicated systems," in *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018.

[12] K. De Porre, C. Ferreira, N. Preguiça, and E. Gonzalez Boix, "ECROs: building global scale systems from sequential code," *Proceedings of the ACM on Programming Languages*, vol. 5, pp. 107:1–107:30, Oct. 2021.

[13] E. Cecchet, "RUBiS archive," 2009. Publication Title: OW2 Projects.

[14] X. Zhao and P. Haller, "Observable atomic consistency for CvRDTs," in *Proceedings of the 8th ACM International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ACM, 2018.

[15] M. Whittaker and J. M. Hellerstein, "Interactive checks for coordination avoidance," *Proceedings of the VLDB Endowment*, 2018.

[16] F. Houshmand and M. Lesani, "Hamsaz: replication coordination analysis and synthesis," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019.

[17] X. Zhao and P. Haller, "Replicated data types that unify eventual consistency and observable atomic consistency," *Journal of Logical and Algebraic Methods in Programming*, vol. 114, p. 100561, Aug. 2020.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[19] J. Wang, C. Li, K. Ma, J. Huo, F. Yan, X. Feng, and Y. Xu, "AUTOGR: automated geo-replication with fast system performance and preserved application semantics," *Proceedings of the VLDB Endowment*, vol. 14, pp. 1517–1530, May 2021.

[20] M. Bravo, A. Gotsman, B. de Régil, and H. Wei, "UNISTORE: A fault-tolerant marriage of causal and strong consistency," *Proceedings of the 2021 USENIX Annual Technical Conference*, 2021.

[21] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," pp. 305–319, 2014.

[22] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pp. 140–149, Sept. 1994.

[23] W. Vogels, "Eventually Consistent: Building reliable distributed systems at a worldwide scale demands trade-offs?between consistency and availability.," *Queue*, vol. 6, pp. 14–19, Oct. 2008.

[24] P. Bailis and A. Ghodsi, "Eventual consistency today: limitations, extensions, and beyond," *Communications of the ACM*, May 2013.

[25] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," 2011.

[26] S. Laddad, C. Power, M. Milano, A. Cheung, N. Crooks, and J. M. Hellerstein, "Keep CALM and CRDT On," *Proceedings of the VLDB Endowment*, vol. 16, pp. 856–863, Dec. 2022.

[27] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.

[28] Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, pp. 690–691, Sept. 1979.

[29] P. Hutto and M. Ahamad, "Slow memory: weakening consistency to enhance concurrency in distributed shared memories," in *10th International Conference on Distributed Computing Systems*, 1990.

[30] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *EuroSys*, 2015.

[31] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitao, N. Preguia, R. Rodrigues, M. Shapiro, and V. Vafeiadis, "Geo-Replication: Fast If Possible, Consistent If Necessary," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2016.

[32] H. Ng, S. Haridi, and P. Carbone, "Omni-Paxos: Breaking the Barriers of Partial Connectivity," in *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, 2023.

[33] G. Zhang, F. Pan, Y. Mao, S. Tijanic, M. Dang'Ana, S. Motepalli, S. Zhang, and H.-A. Jacobsen, "Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–41, 2024.

[34] M. Kleppmann, "Making crdts byzantine fault tolerant," in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 8–15, 2022.

[35] A. Beheshti, "Natural language-oriented programming (nlop): Towards democratizing software creation," in *2024 IEEE International Conference on Software Services Engineering (SSE)*, pp. 258–267, 2024.

[36] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, Dec. 2024.

[37] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proc. ACM Softw. Eng.*, vol. 2, June 2025.

[38] "Replicating UObjects in Unreal Engine | Unreal Engine 5.7 Documentation | Epic Developer Community — dev.epicgames.com." https://dev.epicgames.com/documentation/en-us/unreal-engine/replicating-uobjects-in-unreal-engine. [Accessed 05-12-2025].

[39] T. Köylüoglu and J. Larsson, "Zero self-view latency: An implementation of conflict-free replicated data types in unity: A benchmark of operation-based crdts," 2025.

[40] T. Allweyer, *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2016.

[41] "Terraform | HashiCorp Developer — developer.hashicorp.com." https://developer.hashicorp.com/terraform. [Accessed 05-12-2025].

[42] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, "Graphql: A systematic mapping study," *ACM computing surveys*, vol. 55, no. 10, pp. 1–35, 2023.

[43] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications," in *Advanced research working conference on correct hardware design and verification methods*, pp. 54–66, Springer, 1999.

[44] L. Lamport and Y. Yu, "Tlc–the tla+ model checker," 2001.

[45] S. Laddad, C. Power, M. Milano, A. Cheung, and J. M. Hellerstein, "Katara: Synthesizing crdts with verified lifting," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1349–1377, 2022.

[46] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano, "New Directions in Cloud Programming," Jan. 2021.

[47] K. Segeljakt, S. Haridi, and P. Carbone, "Aqualang: A dataflow programming language," in *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*, DEBS '24, (New York, NY, USA), p. 42–53, Association for Computing Machinery, 2024.

[48] C. Meiklejohn and P. Van Roy, "Lasp: A language for distributed, coordination-free programming," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pp. 184–195, 2015.

[49] K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos, "Styx: Transactional stateful functions on streaming dataflows," *Proc. ACM Manag. Data*, vol. 3, June 2025.

[50] J. Spenger, P. Carbone, and P. Haller, "Portals: An extension of dataflow streaming for stateful serverless," in *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, (New York, NY, USA), p. 153–171, Association for Computing Machinery, 2022.