Distributed Data Persistency

Apostolos Kokolis
University of Illinois,
Urbana-Champaign, USA
kokolis2@illinois.edu

Antonis Psistakis
University of Illinois,
Urbana-Champaign, USA
psistaki@illinois.edu

Benjamin Reidys
University of Illinois,
Urbana-Champaign, USA
breidys2@illinois.edu

Jian Huang
University of Illinois,
Urbana-Champaign, USA
jianh@illinois.edu

Josep Torrellas
University of Illinois,
Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

Distributed applications such as key-value stores and databases avoid frequent writes to secondary storage devices to minimize performance degradation. They provide fault tolerance by replicating variables in the memories of different nodes, and using data consistency protocols to ensure consistency across replicas. Unfortunately, the reduced data durability guarantees provided can cause data loss or slow data recovery. In this environment, non-volatile memory (NVM) offers the ability to attain both high performance and data durability in distributed applications. However, it is unclear how to tie NVM memory persistency models to the existing data consistency frameworks, and what are the durability guarantees that the combination will offer to distributed applications.

In this paper, we propose the concept of *Distributed Data Persistency* (DDP) model, which is the binding of the memory persistency model with the data consistency model in a distributed system. We reason about the interaction between consistency and persistency by using the concepts of Visibility Point and Durability Point. We design low-latency distributed protocols for DDP models that combine five consistency models with five persistency models. For the resulting DDP models, we investigate the trade-offs between performance, durability, and intuition provided to the programmer.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Networks** → *Network protocol design*; • **Hardware → Memory and dense storage**;

KEYWORDS

Distributed architecture, Data consistency, Memory persistency, Non-volatile memory

ACM Reference Format:
Apostolos Kokolis, Antonis Psistakis, Benjamin Reidys, Jian Huang, and Josep Torrellas. 2021. Distributed Data Persistency. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480060

1 INTRODUCTION

Over the past decades, distributed storage systems such as key-value stores and transactional databases have become a core component of the cloud infrastructure [12, 13, 19, 43, 52, 71]. To meet ever-increasing performance requirements, these distributed applications typically avoid frequent accesses to slow storage devices such as solid-state drives (SSDs). This is because an access to such devices can take tens of microseconds. Instead, many applications store data in main memory and provide fault tolerance by making replicas (i.e., copies) of variables in other nodes' memories.

These replicas are managed by the runtime system using a *data consistency model*. There are many different consistency models in use [69], which differ in their strength. Strong consistency models strive to ensure that reading different replicas in different nodes returns similar, largely up-to-date versions of the variable. In contrast, weak models permit reads to different replicas to return inconsistent, sometimes stale versions. Commercial applications support a variety of models—e.g., Apache's ZooKeeper [5, 27] supports the strong Linearizable consistency, while Google's Bigtable [12] provides the weak Eventual consistency.

Unfortunately, due to the reduced support for data durability in these environments, distributed applications can suffer from data loss or slow data recovery. For example, a Facebook key-value store cluster needs hours to recover using remote data replicas, and days to recover using a backend storage [49, 79]. To make matters worse, a failure of the entire system can cause the permanent loss of in-memory state [22, 79].

The recent arrival of non-volatile memory (NVM) [3, 29, 54] offers a promising approach to help distributed applications attain both high performance and data persistence. Indeed, NVM can provide data durability in about 100-400 ns [31, 55, 72]. This is faster than a network round trip in data centers with high-performance interconnects such as InfiniBand [2, 8, 45, 47, 78].

To facilitate the use of NVM, researchers have developed a framework of *data persistency models* for a single machine with hardware-managed cache hierarchies (e.g., [23, 53]). These models vary in how eagerly they persist writes to NVM. For example, Strict persistency requires that a variable be persisted as soon as it is updated, while Epoch persistency only requires that updated variables be persisted at certain program locations.

As we use NVM in distributed applications, we have to carefully manage both the consistency and the persistency of the data. Although distributed data consistency has been well studied (e.g., [4, 7, 16, 33, 38, 51, 64, 69, 71]), it has almost always been used in systems which, at best, use slow storage devices for durability [22, 48, 74]. Hence, it is unclear how to best incorporate the NVM memory persistency models into these data consistency frameworks. In fact, it is unclear how these two classes of models interact with each other, and how their combination impacts data durability, performance, and programmer intuition in applications.

This paper addresses these limitations. We introduce the concept of *Distributed Data Persistency* (DDP) model, which is the binding of the memory persistency model with the data consistency model in a distributed system. To reason about the interaction between data consistency and memory persistency, we use the concepts of *Visibility Point* and *Durability Point* of an update. The former is when the update is visible for consumption, and is specified by the consistency model; the latter is when the update is durable, and is specified by the persistency model.

To understand the tradeoffs, we consider five consistency models (Linearizable, Read-Enforced, Transactional, Causal, and Eventual), and five persistency models (Synchronous, Strict, Read-Enforced, Scope, and Eventual), pair-wise combine them, and design a low-latency distributed protocol for each of the resulting DDP models. Using these protocols, we investigate the trade-offs that DDP models offer in terms of performance, durability, intuition provided to the programmer, programmability, and implementability.

Our analysis shows that different DDP models deliver substantially different performance—e.g., one model delivers a 3.3x higher throughput than another for 100 clients. However, any fair comparison between the models has to consider other dimensions as well, such as durability and intuitiveness. In our analysis, we find that, typically, latency-sensitive applications that can tolerate some data staleness work best with DDP models that combine weak consistency with strong persistency. On the other hand, consistency-sensitive applications benefit from stricter consistency and relaxed persistency. For a broad class of applications, an intermediate model that combines Causal consistency with Synchronous persistency appears to be a good choice.

Overall, this paper makes the following contributions:

- The concept of Distributed Data Persistency (DDP) model, which integrates a memory persistency model with a data consistency model in distributed systems. To reason about DDPs, we use the interaction between the Visibility and Durability points of an update.
- The design of novel, low-latency distributed protocols for many DDP models. These protocols are tailored to contemporary hardware, which provides low-latency, high-bandwidth network and storage through RDMA and NVM.
- A thorough trade-off comparison of different DDP models, in terms of performance, durability, intuitiveness, programmability, and implementability.
- A performance evaluation of different DDP models using distributed applications.

2 BACKGROUND

2.1 Data Consistency Models

To provide fault tolerance and performance, distributed computing applications replicate variables in the volatile memory hierarchy of multiple nodes. The replicas of a variable in different nodes may be read and updated concurrently by different processes. The consistency model of a system defines the requirements and guarantees of what data values can processes read. Many consistency models exist, as described in the distributed-computing literature (e.g. [4, 7, 38, 69]). Typically, there is a performance vs. data staleness trade-off: strict models require writes to update the replicas very soon, while weaker models sacrifice this requirement for higher performance. Next, we describe several models.

Linearizable Consistency or Linearizability. Linearizable consistency is the strongest consistency model for distributed systems. It requires that all writes to all variables be seen by all processes in the same order and, additionally, that all reads and writes be ordered by their timestamps [18, 30, 69]. This model is highly intuitive but may deliver low performance.

Causal Consistency. In this model, accesses are partially ordered according to the happens-before relationship. Specifically, two accesses within the same thread are ordered based on program order. Moreover, a read from a thread that obtains a value written by a write from another thread is ordered after the write. Further, this relation is built transitively. In this model, a thread can observe a write w only after it observes every previous write in w's happens-before history. Note that writes do not need to be applied instantly and, therefore, reads can return stale values. Replicas only need to reflect causally-related writes in order.

Eventual Consistency. In this model, writes are propagated lazily. The model only guarantees that all the replicas will eventually see all the writes. This model provides very weak consistency guarantees, and processes might read unexpected values. However, it offers great performance.

Transactional Consistency. Many contemporary databases organize their operations in transactions (Xactions). While different variations of this model exist, this paper uses a simple one. The writes in a Xaction only need to be propagated to all the replicas by the end of the Xaction. If the Xaction fails, none of the updates are performed. Moreover, the operations within a Xaction can only see the effects of other Xactions that have completed prior to it.

Read-Enforced Consistency. In this paper, we introduce this new model, which is slightly weaker than Linearizable consistency. It is inspired by the Read-Enforced durability of Ganesan et al. [22]. In this model, a write only needs to be visible to all the replicas at the point when a subsequent read tries to read any of the replicas. Compared to linearizability, this model allows faster completion of writes at the potential expense of delaying reads.

Current systems support most of these models, although linearizability is often eschewed for performance reasons. The availability and performance of Causal consistency make it an attractive choice for many applications [7, 38, 44, 69], such as online services. Eventual consistency is one of the most widely deployed [19, 69, 70] because of its performance.

2.2 Memory Persistency Models

The availability of NVM has led to the creation of multiple memory persistency models for single-server platforms. These models differ in how eagerly writes are persisted to NVM [53]. The models range from a strict one, where a write is immediately persisted to NVM, to relaxed ones, where writes are persisted lazily under certain conditions. These models need to be adapted to work in a distributed system, where nodes use asynchronous messages to coordinate.

In this paper, we build on these models and on more traditional durability protocols that distributed systems have used to persist data to SSDs (e.g., [22, 48, 74]). In this section, we describe several persistency models.

Synchronous Persistency. In this paper, we introduce this new model as the adaptation of the Strict memory persistency model from single-server systems [53] to distributed systems. In this model, when a replica is updated in volatile memory, it is immediately persisted to NVM. This model is strict, but the time of the persist depends on when the replica is updated, which in turn depends on the data consistency model of the system. For this reason, we call it *Synchronous*. It is the most intuitive model.

Read-Enforced Persistency. This model was introduced by Ganesan et al. [22]. It is more relaxed than Synchronous. Replicas do not need to be persisted when they are updated. Instead, the requirement is that all the updated replicas are persisted before any of them is read. This model guarantees that any value that has been read is also recoverable. However, there is no guarantee for updates that have not yet been read by processes—such updates may be lost in a crash or program failure.

Eventual Persistency. In the Eventual persistency model, persist operations are performed lazily. They happen whenever it is possible, without any concern about the order of persists. No other guarantees are provided. In case of a volatile storage failure, an arbitrary number of updates may be lost.

Scope Persistency. In the context of NVM persistence, there are proposals that persist a set of writes as a group. They include Strand [23, 53] and Epoch persistency [32, 53]. In this paper, we propose a generalized approach where writes belong to *Scopes*—a concept reminiscent of Fence Scoping [37]. Every write is augmented with a *Scope ID*, and the application can invoke a *Persist* on a given Scope ID. Writes can be persisted in the background, but the model guarantees that all the writes in a scope are persisted by the time the Persist call for that scope terminates. The scopes in a program may be totally ordered, partially ordered, or not ordered at all, based on their Scope IDs. In our design, we use total order within a process and no order across processes. In all cases, if there is a volatile storage failure, the state of all the completed scopes is recovered, and that of those partially executed is discarded.

Strict Persistency. This is the strictest model. It dictates that a write should be persisted in the NVM of all the replica nodes by the time the write completes—possibly even before the replicas in the volatile memories of the replica nodes receive the update[61, 62]. On a failure of volatile storage, no update is lost. This model is relatively less interesting because of its high strictness.

Most existing systems use a persistency model close to Eventual persistency. This is because they value performance and do not want to pay the cost of persisting to SSDs or disks in the critical path. Some systems such as Redis [57] give the user a choice of models, ranging from Eventual-like to more strict. Some systems provide Synchronous-like persistency, such as LogCabin [39]. Read-enforced persistency has been recently proposed [22] and, to our knowledge, it is not used yet.

3 MOTIVATION: IMPACT OF CONSISTENCY AND PERSISTENCY MODELS

To motivate the importance of understanding how consistency and persistency interact, we perform a simple experiment. We take the Odyssey system [68] and implement a strict environment where both writes to volatile replicas and persists to NVM happen synchronously (i.e., a client's write does not return to the client until all replicas are updated and persisted). We then repeat the experiments without synchronously persisting to NVM, but still updating the volatile replicas in the critical path before returning to the client. Finally, we repeat the experiments without persisting to NVM or updating the volatile replicas before returning to the client. For all these experiments, we use a 3-node cluster, and every variable is replicated in all nodes. Each node has 24 Xeon E5-2687W cores, and connects to other nodes with Mellanox ConnectX-4 NICs that perform RDMA over Infiniband. The nodes run client threads issuing write requests and worker threads processing requests. These threads execute on separate cores. Table 1 shows the relative throughput of the three environments.

Table 1: Relative throughput of three environments.

Volatile Updates in Critical Path?	NVM Updates in Critical Path?	Normalized Throughput
Yes	Yes	1
Yes	No	1.32
No	No	4.08

As can be seen from Table 1, the throughput (normalized to the first environment) is significantly different. A relaxed environment that completes writes locally without updating or persisting replicas delivers a 4x higher throughput. Given the large number of consistency and persistency models, we need to develop a framework to examine the interactions between consistency and persistency models, and investigate the tradeoffs between the different combinations. These are the goals of the rest of the paper.

Note that writing correct protocols for distributed systems is complex. For example, the Hermes distributed consistency protocol [25, 33] is 16K lines of code (LOC), and ZooKeeper [5, 6, 27] is 294K LOC. Hence, it is important to understand how consistency and persistency interact and how to systematically design protocols to support combinations of them.

4 INTEGRATING PERSISTENCY AND CONSISTENCY IN DISTRIBUTED SYSTEMS

We propose to bind memory persistency models with data consistency models in distributed platforms, creating what we call *Distributed Data Persistency* (DDP) models. To understand our approach, consider a distributed computer (e.g., a datacenter) where each node has a volatile memory hierarchy and some NVM. This is the architecture we will use in this paper. Figure 1 shows two nodes of such a platform.

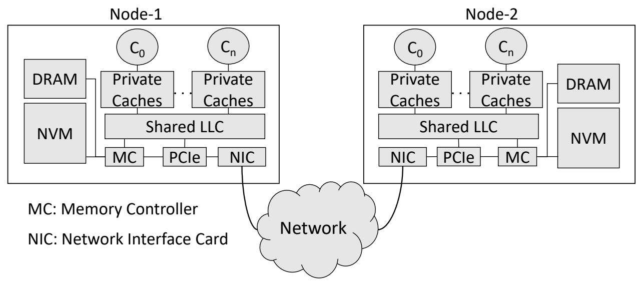


Figure 1: Distributed computer with NVM.

When an application such as a key-value store or a database runs on this platform, the runtime typically makes copies of keys (or records) in the volatile memory hierarchy of multiple nodes—e.g., there are *N replicas* of every key. With today's Data Direct I/O (DDIO) technology [21, 28], the hardware makes the copies in the Last Level Caches (LLC) of the nodes. Such replication is performed for fault tolerance and performance, and may be later followed by the persistence to durable storage—NVM in our case.

In such a system, we decouple data consistency models from memory persistency models by using the concepts of *visibility* and *durability*. The consistency model is concerned with *when to propagate* the update of a key to the key's replicas in the volatile hierarchies of nodes; the persistency model is concerned with *when to persist* the update to the NVMs of nodes. To be specific:

The consistency model defines the *Visibility Point* (VP). The VP of an update with respect to a node is when the update becomes available for consumption at that node. The persistency model defines the *Durability Point* (DP). The DP of an update is when the update is made durable (in the necessary number of nodes, as required by the recovery system) and, hence, cannot be wiped out by a failure.

Broadly speaking, it helps to think as follows. Consistency models are more or less strict depending on how eagerly they propagate the update of a key to the volatile memory hierarchy of the nodes with replicas. Persistency models are more or less strict depending on how eagerly they persist the update to the NVMs of the nodes with replicas. This separation of concerns provides an intuitive way to plug the framework of persistency models into the framework of consistency models, creating what we call DDP models.

Table 2 shows the VP and DP of an update in the different consistency and persistency models, respectively, that we consider in this paper. The models are listed from more to less strict. In the following, for a given variable, we call "replica nodes" all the nodes that contain a copy of the variable.

Consistency Models. In the *Linearizable* model, the VP of an update with respect to all replica nodes is when the update takes place. A client's write in a node is not completed until the volatile memories of all the replica nodes have been updated.

In *Read-Enforced* consistency, the VP of an update with respect to all replica nodes is sometime before the update is read by a node.

Table 2: Visiblity and durability points of an update for different data consistency and memory persistency models, respectively. "Wrt all/a node(s)" stands for "with respect to all/a replica node(s)".

Consistency	Visibility Point (VP) of an Update
Linearizable	Wrt all nodes: when the update takes place
Read-Enforced	Wrt all nodes: before the update is read
Transactional	Wrt all nodes: at the transaction end
Causal	Wrt a node: after the VPs wrt the same node of all the updates in the happens-before history
Eventual	Wrt a node: sometime in the future
Persistency	**Durability Point (DP) of an Update**
Strict	When the update takes place
Synchronous	At the visibility point of the update
Read-Enforced	Before the update is read
Scope	Before or at the scope end
Eventual	Sometime in the future

A client's write completes as soon as the local key is updated; the update propagates to the replica nodes in the background. However, a read to any replica will stall until all the replica nodes have been updated.

In *Transactional* consistency, the code is annotated with transactions, and the VP of an update with respect to all replica nodes is at the transaction end. A write completes as soon as the local key is updated; the update propagates to the replica nodes in the background. The end-transaction operation stalls until all the writes in the transaction have updated all the replica nodes.

In *Causal* consistency, the VP of an update u with respect to a given replica node is sometime after the VPs with respect to the same replica node of all the updates U (to any key) that are in u's happens-before history. We call the *list* of U updates the *Causal History* (or *cauhist*) of u.

Finally, in *Eventual* consistency, the VP of the update with respect to a given replica node is sometime in the future. The update is propagated lazily. The model only guarantees that the update will eventually reach its VPs with respect to the different replica nodes.

Persistency Models. In the *Synchronous* persistency model, the DP of an update is at the VP of the update. In other words, when a volatile replica is updated (according to the consistency model), the replica is also persisted to NVM.

Table 2 also shows the even stronger but unintuitive *Strict* persistency, where the DP is when the update takes place. In this model, briefly mentioned by Talpey [62], when a node writes a key, the update has to be immediately persisted in the replica nodes, even if the volatile replicas in such nodes are not updated. In this paper, we de-emphasize this model.

In *Read-Enforced* persistency, the DP is before the update is read. Specifically, a read to any replica will stall until all the replicas have been persisted to NVM.

In *Scope* persistency, every update is annotated with a Scope ID. The DP of an update is before or at the point when execution reaches the *end-of-scope* annotation for that Scope ID. When the annotation is reached, execution stops until all the writes in the scope are persisted in the replica nodes.

Finally, in *Eventual* persistency, the DP of the update is sometime in the future; the update is persisted lazily in the replica nodes.

Distributed Data Persistency (DDP) Models. We define a DDP model as the binding of a memory persistency model with a data consistency model. We represent it as <consistency model, persistency model>.

5 DDP PROTOCOLS FOR MODERN HARDWARE

Based on the insights from the previous section, we now design new distributed protocols for several DDP models. We target a modern data center architecture, where nodes communicate with low latency with advanced RDMA [2, 17, 45, 50] and use NVM for persistency. In this setting, where a round trip between nodes takes single-digit μs, and data persistency can be obtained in a few-hundred *ns*, we design protocols that emphasize low latency. Specifically, we design protocols that have no single leader—i.e., a client read or write request can be received and processed at *any node.* Moreover, on reception of a client's write, a node broadcasts messages to all the other replica nodes, instead of sending a message that sequentially visits all the other replica nodes.

Our designs are based on the linearizable-consistent (no persistency) protocol used by Hermes [33]. Following their terminology, we call *Coordinator* the node that receives the client's read/write request for a key, and *Followers* all the other nodes with a replica of the key. Finally, for simplicity and like in Hermes, we assume that keys are replicated in all the nodes; reducing the number of replica nodes does not change the protocols conceptually, but may affect performance.

Before we describe the design of the DDP protocols, we outline the protocol operations.

5.1 Overview of the Protocol Operations

Table 3 shows the types of messages in our protocols. The basic protocol operation can be illustrated with a write in a strict model. When the coordinator receives a write from the client, it broadcasts an *INV (+data)* message to all the followers. This is an invalidation message that also includes the update. On reception of the message, a follower invalidates its current value of the key, sets the key's state to transient, and buffers the new key value (*data*). Then, it acknowledges the operation with an *ACK* message back to the coordinator. When the coordinator has received all the ACKs, it broadcasts a validation *VAL* message to all the followers. On reception, each follower knows that all the followers have been notified, and sets the key to the new value. As shown in Table 3, our ACKs and VALs may combine consistency and persistency information or may only apply to consistency (*ACK_c*, *VAL*_c) or persistency (*ACK_p*, *VAL*_p) events.

Causal and Eventual consistency protocols do not use *ACK* or *VAL* because there is no need for global information about *when* an update becomes visible. Hence, the coordinator simply sends update *UPD* messages with the data. In the case of Causal consistency, UPD includes the causal history (*cauhist*) of the write.

Table 3: Types of messages in our protocols.

Message	Explanation
INV (+data)	Invalidates the current value of a key and provides its updated value (*data*)
ACK	Acknowledges an event
ACK_c	Acknowledges a consistency event
ACK_p	Acknowledges a persistency event
VAL	Marks the termination of an event
VAL_c	Marks the termination of a consistency event
VAL_p	Marks the termination of a persistency event
UPD (+cauhist)	Provides an updated value for a key plus the causal history of this update (*cauhist*)
INITX	Informs of the beginning of a transaction
ENDX	Informs of the end of a transaction
[PERSIST]s	Informs of the end of scope *s*
[XXX]s	[INV]s [ACK_c]s [ACK_p]s [VAL_c]s [VAL_p]s for scope *s*

In Transactional consistency, the coordinator also sends begin and end transaction messages (INITX and *ENDX*). Finally, under Scope persistency, all messages are tagged with the scope *s* they belong to (e.g., *[ACK_c]s*). Furthermore, the coordinator also sends an end-of-scope message (*[PERSIST]s*) when execution reaches the end of the scope.

5.2 DDP Models with Synchronous Persistency

Figure 2 shows the timelines of the protocols for the DDP models that bind Synchronous persistency with various consistency models. The top row corresponds to the coordinator and the bottom row to a follower. In each subfigure, the left part shows the requests issued by a client. The thicker line shows the time during which a persist operation to NVM takes place. Furthermore, a down arrow means that the write updates the local node's cache. Finally, a two-headed arrow means that the message is sent to or received from multiple followers.

<Linearizable, Synchronous>. The coordinator is shown in (a) and a follower in (b). When the coordinator receives a write from a client, it updates its local cache and sends an INV (+data) to all the followers. On reception of INV (+data), a follower updates its local LLC and, to satisfy Synchronous persistency, persists the update to NVM before retuning an ACK to the coordinator. On reception of all the ACKs, the coordinator finishes persisting the update (to satisfy Synchronous persistency) before broadcasting a VAL to all the followers to indicate the operation is complete. *Only then* can the coordinator tell the client that the write is complete: all nodes have updated their volatile replica (required by Linearizable consistency) and have persisted it in their NVM (required by Synchronous persistency). Overall, we see that writes have high latency in this DDP model.

Consider now a client read to the same key. The coordinator cannot process any other request while the write is in progress. A follower stalls the read until all replicas have been updated (required by Linearizable consistency) and persisted (required by Synchronous persistency). Such condition is only guaranteed when VAL is received. Overall, reads also have high latency.

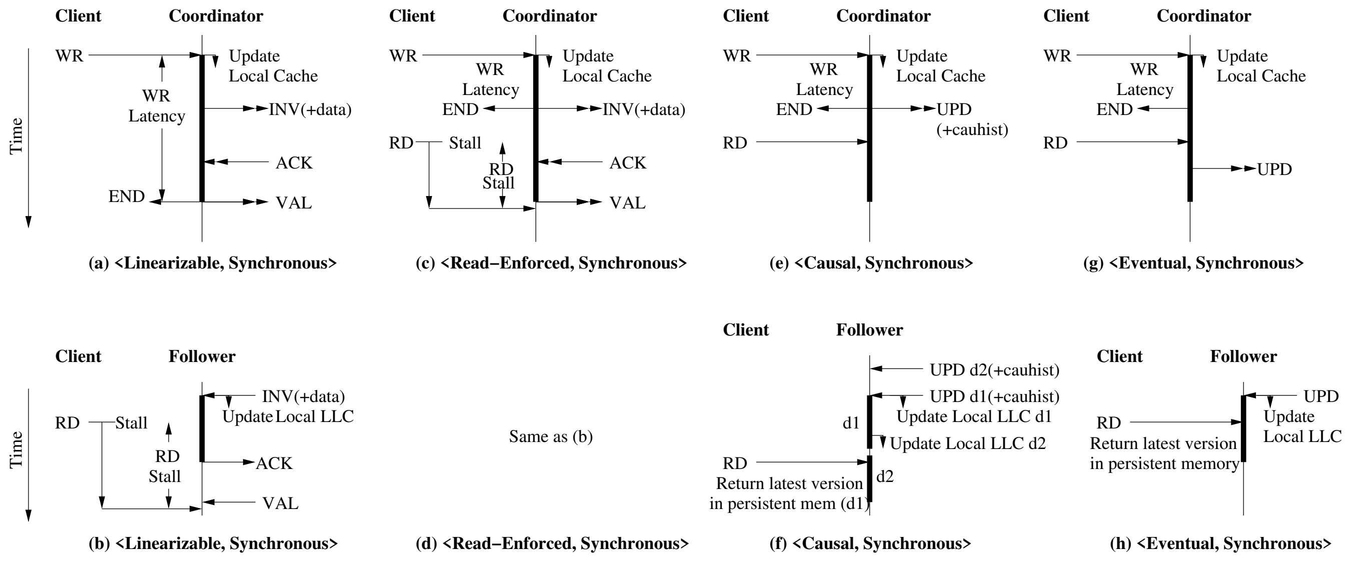

Client
Coordinator
Follower
WR
WR Latency
END
RD
Stall
RD Stall
Update Local Cache
INV(+data)
ACK
VAL
UPD (+cauhist)
UPD
Time
(a) <Linearizable, Synchronous>
(c) <Read–Enforced, Synchronous>
(e) <Causal, Synchronous>
(g) <Eventual, Synchronous>
Update Local LLC
Same as (b)
UPD d2(+cauhist)
UPD d1(+cauhist)
Update Local LLC d1
Update Local LLC d2
d1
d2
Return latest version in persistent mem (d1)
Return latest version in persistent memory
(b) <Linearizable, Synchronous>
(d) <Read–Enforced, Synchronous>
(f) <Causal, Synchronous>
(h) <Eventual, Synchronous>

Figure 2: Timelines of the protocols for the DDP models that bind Synchronous persistency with various consistency models. The top row corresponds to the coordinator and the bottom one to a follower.

<Read-Enforced, Synchronous>. The coordinator is shown in (c) and a follower in (d). When the coordinator receives a write, it updates its local cache and sends an INV (+data) to all the followers. Read-Enforced consistency does not require the volatile replicas to be updated before completing the write—only when one of the replicas is read. Hence, the coordinator immediately tells the client that the write is complete, while the local persist and the remote updates and persists are in progress. In this DDP model, writes have low latency. In the follower, the operation is the same as in (b): once the LLC is updated and the update is persisted (required by Synchronous persistency), an ACK is sent. The coordinator collects all ACKs and finishes its persist. Then, it sends a VAL that indicates that *all replicas* have been updated and persisted.

A client read for the same key can only be serviced by the coordinator after sending VAL, and by a follower after receiving VAL. The reason is that Read-Enforced consistency requires a read to stall until all volatile replicas are updated. Moreover, Synchronous persistency requires that, at the same time as the replicas are updated, they are also persisted. Overall, reads have high latency.

<Causal, Synchronous>. The coordinator is shown in (e) and a follower in (f). On a write, the coordinator updates the local cache, sends the UPD with the cauhist of the write to all the followers, and returns to the client. Causal consistency only requires that a replica be updated after it has been updated with the updates in the causal history of the write. Subfigure (f) shows the follower receiving two updates (UPD d2 and UPD d1) in an order opposite to their cauhist. In this case, the first one (UPD d2) is buffered. When the second one (UPD d1) is received, it updates the LLC and, because of Synchronous persistency, it is persisted right away. Then, the first update is performed on the LLC and is persisted.

At any time, any read for the key that arrives at the coordinator or a follower proceeds with no stall. This is because Causal consistency places few constraints on when the replicas are updated, and Synchronous persistency only requires that, when the replica *is* updated, it is persisted. However, Synchronous persistency requires that the read get the latest *persisted* version, so that the version is recoverable on a failure. In our example in the follower, it is version d1. Overall, in this DDP model, both writes and reads have low latency.

<Eventual, Synchronous>. The coordinator is shown in (g) and a follower in (h). Using Eventual consistency makes this DDP model even more relaxed. Indeed, Eventual consistency adds no cauhist to the UPD messages. Updates are performed on the LLC of the follower in the order they arrive—but, because of Synchronous persistency, they are immediately persisted when they do. We denote the relaxed nature of Eventual consistency by delaying the sending of UPD. Overall, both writes and reads have low latency.

5.3 DDP Models with Read-Enforced Persistency

For brevity, we only discuss protocols for two DDP models that bind Read-Enforced persistency with consistency models: <Linearizable, Read-Enforced> and <Causal, Read-Enforced> (Figure 3).

<Linearizable, Read-Enforced>. The coordinator is shown in (a) and a follower in (b). This protocol decouples ACKs for consistency (ACK_c) from those for persistency (ACK_p). When the coordinator receives a write, it updates its local cache and sends INV (+data) to all followers. On reception of INV (+data), a follower updates its local LLC and immediately sends an ACK_c. When the coordinator receives all ACK_c, it knows that all the replicas have been updated. This is the condition that Linearizable consistency requires to tell the client that the write terminated. Read-Enforced persistency does not pose any condition on write termination. Overall, writes still have substantial latency, but less than in <Linearizable, Synchronous>.

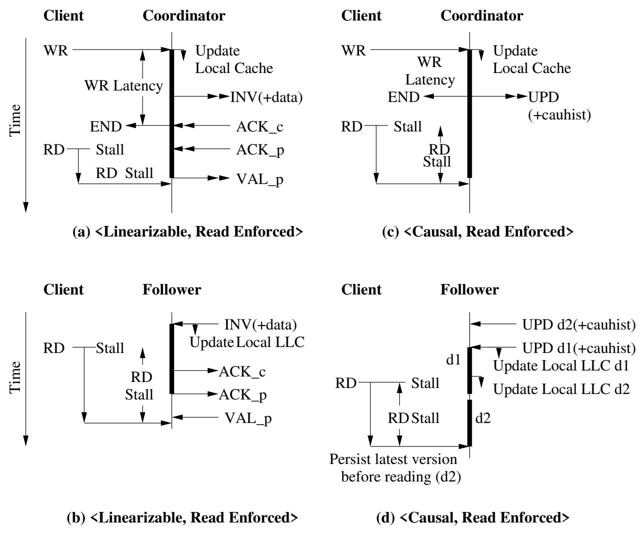


Figure 3: Timelines of the protocols for the DDP models that bind Read-Enforced persistency with Linearizable ((a) and (b)) or Causal ((c) and (d)) consistency.

A read in either the coordinator or follower, however, has to stall until all the replicas have persisted to NVM—as required by Read-Enforced persistency. Therefore, after a follower persists the update, it sends an ACK_p to the coordinator. When the coordinator receives all ACK_p messages and persists its version, it sends VAL_p. Because of Read-Enforced persistency, a read stalls in the coordinator until VAL_p is sent. Further, a read stalls in a follower until VAL_p is received. Hence, reads have high latency in this DDP model.

<Causal, Read-Enforced>. The coordinator is shown in (c) and a follower in (d). The operation of a write is the same as in <Causal, Synchronous> because the change in persistency model does not impact the actions on a write. However, reads now may have to stall longer. The reason is that, while Synchronous persistency only required that a read obtained a persisted version of the key, Read-Enforced persistency prevents a read to proceed unless the latest visible version of the key is persisted. Hence, as shown in (c), a read in the coordinator stalls until the update is persisted (thick line). Further, as shown in (d), a read in the follower stalls until the follower's latest visible version persists—i.e., the read waits until d2 persists and then reads it. Overall, in this DDP model, writes have low latency but reads do not.

5.4 <Transactional, Synchronous> DDP Model

To understand the protocol of a DDP model that includes Transactional consistency, we consider <Transactional, Synchronous> in Figure 4 (coordinator in (a) and follower in (b)). The client performs a transaction by issuing an *Init Xaction* request followed by multiple writes and reads, and then an *End Xaction* request. When the coordinator receives *Init Xaction*, it sends an INITX to all the followers, which persist the event (because of Synchronous persistency) and send an ACK to the coordinator. The Init Xaction request only completes when the coordinator has received the ACK from all the followers and persisted the event locally. At this point, all replica nodes are aware of the transaction. From then on, a write causes the coordinator to update the local cache, send the INV (+data) and immediately acknowledge to the client without waiting for the ACKs from the followers. Hence, a write is fast. The followers send their ACK after updating their LLCs, without waiting for the update to be persisted to NVM.

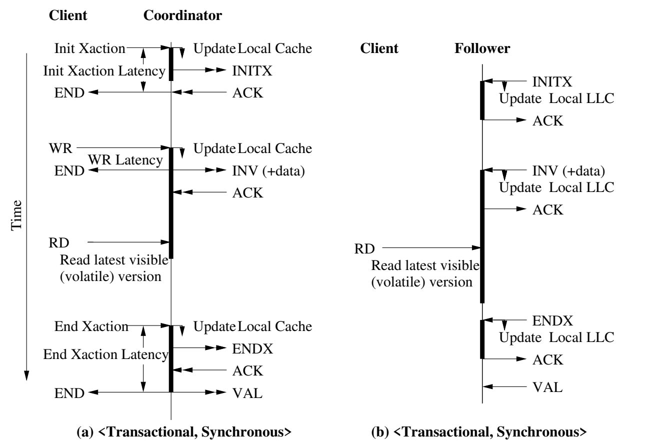


Figure 4: Protocol timeline for the DDP model that binds Synchronous persistency with Transactional consistency. The figure shows the coordinator (a) and a follower (b).

On reception of the *End Xaction* request, the coordinator broadcasts an ENDX to the followers. A follower, before returning an ACK for ENDX, must complete all the updates in the transaction to both the volatile LLC (as required by Transactional consistency) and to the NVM (as required by Synchronous persistency). When the coordinator has received all the ACKs and completed all its updates in the transaction to both the volatile caches and the NVM, it acknowledges the *End Xaction*.

During the transaction, a read in the coordinator or followers is fast. It does not wait; it simply reads the latest visible version—i.e., the latest one in the volatile memory hierarchy.

On top of this protocol, there is additional software infrastructure that detects and handles transactional conflicts. Specifically, at every read and write at the coordinator or followers, the address to be accessed is compared to those of all the reads and writes in the currently-active transactions. If a conflict is detected, different actions can be taken, such as transaction squashes or stalls, depending on the flavor of transactional model supported.

5.5 <Linearizable, Scope> DDP Model

Finally, to understand the protocol of a DDP model that binds Scope persistency with a consistency model, we consider <Linearizable, Scope> in Figure 5 (coordinator in (a) and follower in (b)). Recall that, in Scope persistency, writes, persist operations, and messages are tagged with a scope ID *s*.

When the coordinator receives a write, it updates its local cache and broadcasts an INV (+data) to all followers. Each follower must update its volatile replica before returning an ACK_c. When the coordinator has received all the ACK_c messages, it broadcasts a

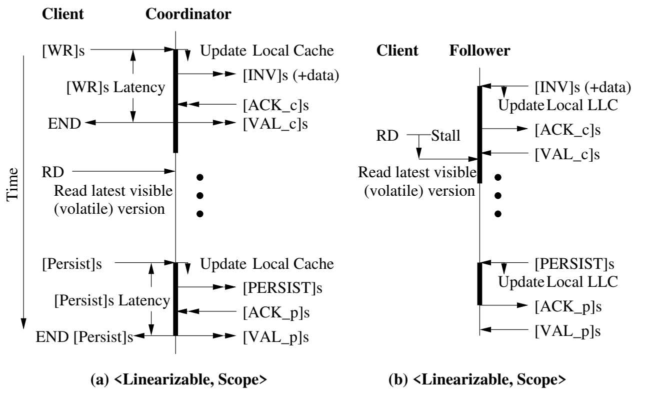


Figure 5: Timeline of the protocol for the DDP model that binds Scope persistency with Linearizable consistency. The figure shows the coordinator (a) and a follower (b).

VAL_c. At this point, since all the volatile replicas have been updated (as required by Linearizable consistency), the write is acknowledged. Writes are relatively slow because of Linearizable consistency.

When the coordinator receives a persist request for this scope, it broadcasts a PERSIST to all the followers. The latter persist to NVM all the updates in the scope and respond with ACK_p. When the coordinator has collected all the ACK_p and locally persisted all the updates in the scope, it broadcasts VAL_p. Since now the scope is fully persisted (as required by Scope persistency), the client is acknowledged.

A read in the coordinator or the followers is typically fast. It reads the latest visible version in the volatile hierarchy. However, sometimes it has to stall. Consider the read in Figure 5(b). There is a new version in the follower but, because of Linearizable consistency, the read cannot read it until VAL_c is received—i.e., when all the followers have this version as well.

6 TRADEOFFS BETWEEN DDP MODELS

The different DDP models provide different tradeoffs between durability, performance, intuition provided to the programmer, programmability, and implementability. Durability refers to how capable the system is to retain a consistent state after a failure that causes the loss of some or all the volatile state. Performance depends on three main factors: the speed of reads, the speed of writes, and the volume of traffic generated.

Programmer intuition is determined by what values a read can return. In particular, we consider whether the system supports monotonic reads and/or non-stale reads [22]. A system supports monotonic reads if, given two system-wide reads to the same variable, the later read always provides the same or a later version of the variable that the earlier read provided. A system fails to provide non-stale reads if a read that follows a write system-wide fails to provide the value of the write. The most obvious case is when a failure between the write and the read causes the loss of the written version. Intuitive systems support both monotonic and non-stale reads.

Programmability refers to the developer's ease of writing the application. For example, if the developer has to include annotations for transactions or scopes, programmability is hurt. Finally, implementability refers to the simplicity of the algorithms in the model. For example, keeping track of the happens-before histories of writes in the Causal consistency model complicates the implementation.

6.1 Specific DDP Model Analysis

Table 4 compares ten representative DDP models: five that bind Synchronous persistency, two that bind Read-Enforced persistency, one that binds Eventual persistency, and two that bind Scope persistency to consistency models. We consider durability, performance, programmer intuition, programmability, and implementability. In the table, upward, flat, and downward arrows mean high, medium and low; crosses mean no and tick marks yes.

6.1.1 Combinations with Synchronous Persistency. Row 1 shows the very strict <Linearizable, Synchronous>. Durability is high because a write does not return until it is persisted in all replica nodes. In terms of performance, writes are not optimized because a write in the coordinator only returns when all ACKs are received and the VALs are sent out; reads are not optimized either because a read in a follower is blocked until the VAL from the coordinator is received. For these reasons, even though we can say that the traffic is medium, the overall performance is low. In terms of intuitiveness, this model is highly intuitive because it provides both monotonic reads and non-stale reads. Finally, both programmability and implementability are high.

Row 2 shows <Read-Enforced, Synchronous>, which relaxes consistency by allowing writes to return as soon as the coordinator sends the INVs. Reads, however, are not optimized and still need to wait until a prior write to the same address is propagated to all the replicas and persisted. In this model, durability is medium because, if a failure occurs between the write and the subsequent read, the written version may fail to be persisted and be lost. Since writes are optimized but reads are not, and the traffic is medium, overall performance is medium. Monotonic reads are guaranteed but not non-stale reads, due to the failure just described: as the system recovers from the failure, a read will not return the value produced by the lost write. Hence, intuitiveness is medium. Programmability and implementability are high.

Row 3 shows <Transactional, Synchronous>, which is similar to <Linearizable, Synchronous> except that it operates at the transaction level. It has high durability—completed transactions are never lost. It optimizes writes through overlapping them inside a transaction, and reads by not stalling them. As a result, although traffic is high due to transaction begin/end messages, its performance is high. It provides both monotonic reads and non-stale reads and, hence, intuitiveness is high. However, programmability is low due to the need to annotate code with transactions, and implementability is low due to the need to implement transactions and their conflict detection and resolution.

Row 4 shows <Causal, Synchronous>, which optimizes both writes and reads. Neither of them stalls: writes return as soon as the coordinator sends the updates, and reads return the latest version in persistent memory. Since the write optimization may result in a write to be lost in a failure, durability is medium. Both reads and writes are fast but the traffic is high because each write carries its cauhist. Still, performance is high. Monotonic reads are guaranteed

Table 4: Comparing different DDP models. "Opt" means optimized.

Consistency Model	Persistency Model	Durability	Performance: Wr Opt?	Performance: Rd Opt?	Performance: Traffic	Performance: Overall	Programmer Intuition: Monot. Rds?	Programmer Intuition: Non Stale Rds?	Programmer Intuition: Overall	Other: Programmability?	Other: Implementability?
1. Linearizable	Synchronous	⇑	✘	✘	⇔	⇓	✔	✔	⇑	⇑	⇑
2. Read-Enfor.	Synchronous	⇔	✔	✘	⇔	⇔	✔	✘	⇔	⇑	⇑
3. Transactional	Synchronous	⇑	✔	✔	⇑	⇑	✔	✔	⇑	⇓	⇓
4. Causal	Synchronous	⇔	✔	✔	⇑	⇑	✔	✘	⇔	⇑	⇓
5. Eventual	Synchronous	⇓	✔	✔	⇓	⇑	✘	✘	⇓	⇑	⇑
6. Linearizable	Read-Enfor.	⇔	✔	✘	⇑	⇔	✔	✘	⇔	⇑	⇑
7. Causal	Read-Enfor.	⇔	✔	✘	⇑	⇑	✔	✘	⇔	⇑	⇓
8. Linearizable	Eventual	⇓	✔	✔	⇓	⇑	✘	✘	⇓	⇑	⇑
9. Linearizable	Scope	⇑	✔	✔	⇑	⇑	✘	✘	⇑	⇓	⇓
10. Transactional	Scope	⇑	✔	✔	⇑	⇑	✘	✘	⇑	⇓⇓	⇓⇓

because, even if updates arrive at a follower out of order, the system buffers them and performs them in order based on their cauhist. However, non-stale reads are unsupported because writes can be lost to failures. Hence, intuition is medium. Programmability is high but implementability is low because of the need to buffer and enforce the cauhists.

Row 5 shows <Eventual, Synchronous>. As it provides practically no guarantees on when writes update replicas and persist, its durability is low. It has optimized reads and writes, and low traffic. Hence, performance is high. However, since neither monotonic reads nor non-stale reads are supported, intuitiveness is low. Programmability and implementability are high.

6.1.2 Relaxing Persistency. As we relax persistency and go from <Linearizable, Synchronous> to <Linearizable, Read-Enforced> in Row 6, we optimize writes by returning before they are persisted, but not reads. Writes can be lost in a failure. Consequently, durability decreases to medium. Despite the higher traffic due to double ACKs (Figures 3(a)-(b)), performance increases to medium. Further, since non-stale reads are not guaranteed in a failure, intuitiveness decreases to medium.

Row 7 shows <Causal, Read-Enforced>, which mostly optimizes writes over <Linearizable, Synchronous> (Figure 3). Because of this change, and despite the high traffic, its performance is high. However, since writes can be lost, durability is medium and non-stale reads are not supported. As a result, intuitiveness is medium. Implementability is low because of the need to keep cauhists.

Further relaxing persistency to <Linearizable, Eventual> in Row 8 creates a system with both read and write optimization but neither monotonic nor non-stale reads. The result is low durability, high performance, and low intuitiveness.

Finally, we consider Scope persistency. In <Linearizable, Scope> (Row 9) and <Transactional, Scope> (Row 10), we have systems with high durability: in a volatile storage failure, the state of all the completed scopes is recovered, and that of those partially executed is discarded. Within a scope, writes are optimized because they do not serialize their persists, and so are reads, which can read before the scope persists. As a result, despite the higher traffic caused by scope-persist messages, performance is high. Neither monotonic reads nor non-stale reads are guaranteed: on a failure, a group of writes may be discarded after being read because the scope did not persist. However, intuitiveness is still high because either the whole scope survives or no part of it does. Finally, both programmability and implementability are low due to the need to mark and support scopes. Further, both properties are worse if scopes are combined with transactions (Row 10).

7 EVALUATION METHODOLOGY

Modeled Architecture. We model the architecture of a distributed system with 5 servers. Each server is a 20-core multicore with 80 GBs of main memory composed of 64 GBs of NVM and 16 GBs of DRAM. The architecture parameters are shown in Table 5. Each core is an out-of-order core with private L1 and L2 caches, and a shared LLC. A 10% portion of the LLC can be used for direct cache access with DDIO [21, 28]. The servers' Network Interface Card (NIC) supports Remote Direct Memory Access (RDMA), which enables a server to access the remote memory of other servers.

We use RDMA because it supports low-latency data transfers across nodes without involving the remote processor. Unfortunately, current RDMA support is limited, in that an RDMA transaction provides no guarantees that the data have been successfully persisted in remote NVM. However, recent work has proposed RDMA extensions that facilitate operations with NVM [26, 61, 65]. In particular, in our evaluation, we follow SNIA's proposals [61] and model RDMA update commands that guarantee that, on acknowledgment, the remote volatile memory or the remote NVM (depending on the type of command) has been successfully updated. We also model an RDMA command that flushes data from a remote volatile memory to the remote NVM.

We model a high-end NIC with a bandwidth of 200Gb/s [45], and up to 400 Queue Pairs [73] for scheduling messages. Further, we model a $1\mu s$ round-trip latency for a message between two NICs [2, 45, 78].

Table 5: Architectural parameters used for evaluation.

Server Architecture Parameters	
Servers; Clients	5 servers; 20 clients per server
Multicore chip	20 out-of-order cores, 6-issue, 2GHz
Ld-St queue; ROB	92 entries; 192 entries
L1 cache	64KB, 8-way, 2 cycles round trip latency (RT)
L2 cache	512KB, 8-way, 12 cycles RT
LLC cache	2MB/core, 16-way, 38 cycles RT, 10% for DDIO
Network Parameters	
Network latency	1μs RT NIC-to-NIC
Network Bandwidth	200Gb/s
Queue Pairs	Up to 400
Per-Server Main-Memory Parameters	
Capacity	DRAM: 16GB; NVM: 64GB
Channels, Banks	DRAM: 4, 8; NVM: 2, 8
Latency	DRAM: 100ns read/write RT
	NVM: 140ns read, 400ns write RT
Freq; Bus width	1GHz DDR; 64 bits per channel

Modeling Approach. Since we model non-existing, future RDMA primitives and high-end NICs, and want to do sensitivity analyses of even faster NICs and networks, we model performance using simulations. We use the SST simulator [58], Pin [42], and the DRAMSim2 memory simulator [59]. To model NVM, we modified the DRAMSim2 timing parameters and disabled refreshes. With Pin, we collect instruction traces for N cores processing read and write client requests to our key-value stores locally. Traces have no timing information. Then, we take these traces and simulate N cores in our distributed architecture. Timing is dynamically determined by the simulator. The simulation inserts all the protocol messages for correct operation of individual reads and writes.

With our simulation-based approach, we build an infrastructure that can be easily parameterized with new technology advancements and be used to perform sensistivity analyses.

Configurations and applications. We model the protocols for the DDP models that combine all 5x5 <consistency, persistency> pairs shown in Table 2. To minimize the effort of annotating codes for Transactional consistency and Scope persistency, we artificially select transactions to be 5 client requests and scopes to be 10 client requests. For our experiments, we use popular applications. Specifically, we use the widely used memcached [11] application and some simpler in-memory key-value stores such as HashTable, Map, B-Tree [24] and BPlusTree [9]. We evaluate all of them with Yahoo! Cloud Serving Benchmark (YCSB) [14] using different workloads with varying read and write ratios. In our experiments, we warm up the architectural state by running 1 billion instructions before simulating a total of 10 billion instructions. For brevity, the results show the average across all our applications.

8 EVALUATION

8.1 Performance Analysis

Figure 6 compares the performance of our 25 DDP models from Table 2 by showing throughput (measured in client requests/second) (*a*), mean read latency (*b*), mean write latency (*c*), mean access latency (*d*), 95th percentile read latency (*e*), and 95th percentile write latency (*f*). We find it easier to organize the discussion based on consistency models. Hence, in a plot, each group of bars is labeled with a consistency model, and each bar in the group corresponds to a persistency model, as shown in the legend. In a plot, all bars are normalized to <Linearizable, Synchronous>. We run YCSB workload-A, which has 50% read and 50% write requests.

8.1.1 General Observations. Focusing on throughput (Plot *a*), we see that models with Linearizable consistency have the lowest throughput, while those with Causal and Eventual consistency have the highest (often 2-3x higher). Those with Transactional consistency fail to deliver high throughput, mostly due to transaction conflicts—since ≈30% of the transactions conflict. In settings with minimal conflicts, we expect models with Transactional consistency to perform well.

Typically, throughput is inversely correlated with mean read (Plot *b*) and write (Plot *c*) latencies. Models with Causal and Eventual consistency have low read and write latencies. The exception is the combinations with Strict persistency. The latter stalls writes until the updates are persisted everywhere. Models with Transactional consistency have high write latencies, both because of conflicts—a request will not be satisfied until the transaction restarts and completes—and because writes bunch-up at Xaction end. This is especially the case for strong persistency models such as Strict and Synchronous. On the other hand, some models with Read-Enforced consistency have high read latency. This is because, by enabling more write overlapping than Linearizable consistency, they induce more NVM pressure, causing reads to stall longer for writes to persist. As a result, the throughputs of Read-Enforced consistency in Plot *a* are only modestly higher than those of Linearizable consistency

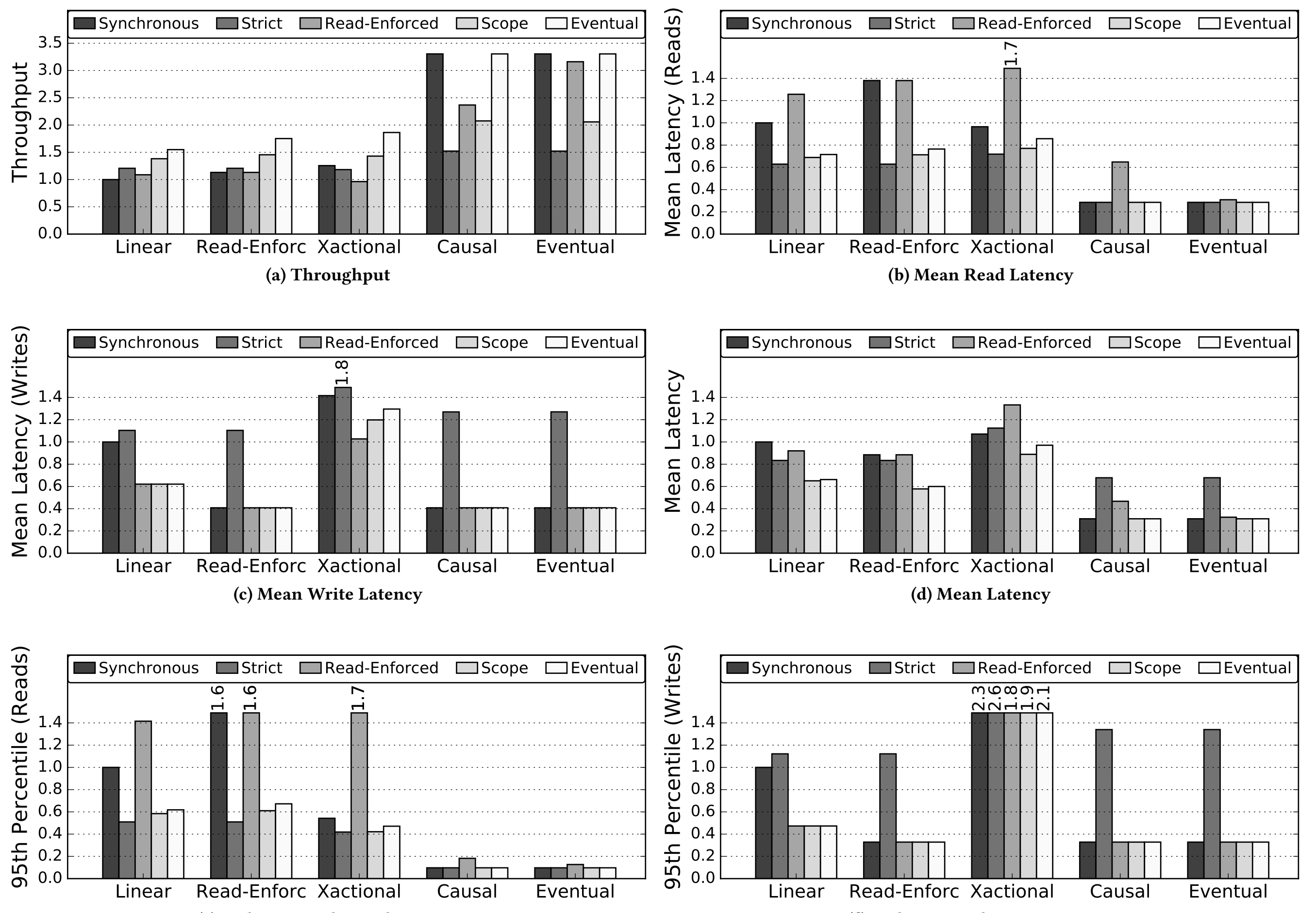
NVM pressure causes unexpected results. Specifically, under Linearizable consistency, Synchronous persistency has a lower read latency than Read-Enforced persistency. Read-Enforced persistency allows more outstanding writes to NVM, which increases NVM pressure and causes subsequent reads that conflict with those writes to take longer.

The 95th percentile plots mostly magnify the effects described. Models with Transactional consistency have a high write tail; some of those with Read-Enforced consistency have high a read tail.

8.1.2 Interesting Combinations. The plots also show that, for a fixed consistency model, which persistency model is used can make a big difference, changing the throughput by up to 2x. In aggregate, models with Strict persistency are the slowest ones, while those with Eventual persistency perform the best.

However, we note that binding a strict persistency model with a relaxed consistency model often delivers high throughput. In particular, the combinations <Causal, Synchronous> and <Causal, Read-Enforced> are attractive because they deliver high throughput (Plot *a*) while retaining medium durability and medium intuitiveness (Table 4). Of course, models that include an Eventual consistency or persistency model, such as <Causal, Eventual> or <Eventual, Synchronous> can perform great. However, as shown in Table 4 for the latter, both durability and intuitiveness are low.

Models with Causal consistency have good performance, but may require substantial buffering of writes with the more strict persistency models [44]. This is because writes need to be buffered until



(a) Throughput

(b) Mean Read Latency

(c) Mean Write Latency

(d) Mean Latency

(e) 95th Percentile Read Latency

(f) 95th Percentile Write Latency

Figure 6: Performance of the different DDP models. In a plot, each group of bars is labeled with a consistency model, and each bar in the group corresponds to a persistency model as described in the legend. In a plot, all bars are normalized to <Linear, Synchronous>.

all their happens-before updates are persisted. In our experiments, we measure that Causal with Synchronous persistency needs about 1-2 orders of magnitude more buffered writes than with Eventual persistency.

Across consistency models, using Read-Enforced persistency delivers a throughput that is only slightly higher or typically lower than using Synchronous persistency. As indicated above, this is because Read-Enforced persistency forces many reads to stall. This poor performance is in contrast to the results of Ganesan et al. [22]. The reason is that our experiments use a higher number of clients (100 instead of 10), and we implement low-latency protocols with no designated leader. As a result, we find that over 30% of the read requests conflict with a yet-to-persist write in <Read-Enforced, Read-Enforced>, instead of 5.1% in Ganesan's work.

Overall, we conclude that different DDP models deliver quite different throughput values. In the extreme case, <Eventual, Eventual> delivers a 3.3x higher throughput than <Linearizable, Synchronous>. As shown in Table 4, however, performance is only one of our considerations. Consequently, different applications may prefer different DDP models. We discuss this issue in Section 9.

8.2 Sensitivity Analysis

To get a better understanding of what determines the performance, we perform three sensitivity analyses. Due to space reasons, we only show data for Linearizable and Causal consistency with all the persistency models.

First, Figure 7 shows the throughput as we vary the number of clients from 10 to 100 (the default) and to 150. The bars are normalized to <Linearizable, Synchronous> with 100 clients. The number of clients affects the traffic and the probability of conflicts between reads and writes. The figure shows that, in most of the DDP models, the throughput improves substantially as the number of clients decreases. For example, <Linearizable, Synchronous>

increase the throughput by 2.2x when going from 100 to 10 clients. Conversely, the throughput decreases as the clients increase.

The exceptions are <Causal, Synchronous> and <Causal, Eventual>, which are largely unaffected by the number of clients. The reason is that, in these models, reads and writes do not stall. However, in Causal with Strict persistency, writes stall until they are persisted; in Causal with Read-Enforced persistency, conflicting reads stall; and in Causal with Scope persistency, reads and writes stall until the scope persists.

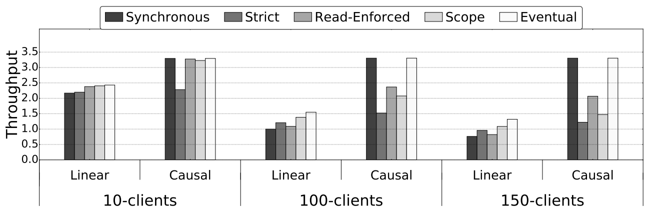


Figure 7: Sensitivity analysis for different clients.

Although not shown in the figure, we also run Transactional consistency. The experimental results show that, as the number of clients reduces from 100 to 10, the number of transaction conflicts decreases by roughly 50%, and Transactional consistency becomes more competitive.

Figure 8 shows the throughput as we vary the NIC-to-NIC round-trip latency from 500ns to 1μs (the default), and to 2μs. All bars are normalized to <Linearizable, Synchronous> with 1μs. The figure shows that the network latency affects mostly the models with Linearizable consistency, while those with Causal consistency are barely affected. The former are affected because network transfer is in the critical path. For example, the throughput of <Linearizable, Synchronous> decreases by 12% when going from 1μs to 2μs. Models with Causal consistency are not affected because updates are generally communicated to other servers in the background.

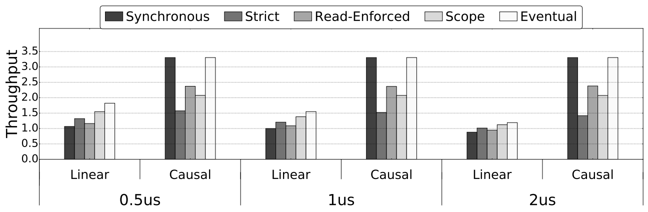


Figure 8: Sensitivity analysis for different NIC-to-NIC round-trip latencies.

Figure 9 shows the throughput as we vary the relative fractions of reads and writes in the workload. We consider workload-B (95% reads and 5% writes), workload-A (the default, which has 50% reads and 50% writes), and our defined workload-W (95% writes and 5% reads). All bars are normalized to <Linearizable, Synchronous> for workload-A. From the figure, we see that the more read-intensive a workload is, the less affected by the consistency and persistency models it is. This is because such models dictate when writes are propagated and persisted. Reads are affected indirectly.

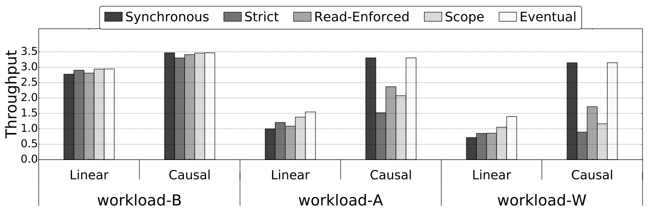


Figure 9: Sensitivity analysis with different relative fractions of reads and writes.

9 IMPLICATIONS FOR APPLICATIONS

Application developers choose the consistency model according to their needs. Our evaluation in Section 8.1 has suggested which persistency models should go with which consistency models and, therefore, which DDP models to use. We now summarize the main insights.

Latency-sensitive applications that can tolerate a certain level of data staleness such as web browsing and social networking often use Eventual consistency [1, 66]. In this case, using Synchronous persistency is a good choice in terms of performance (Figure 6(a)), programmability, and implementability (Table 4).

For consistency-sensitive applications that require bounded staleness but can accept modest latencies, such as certain web search services [66, 69], stricter consistency models such as Read-Enforced consistency are good choices. In this case, Figure 6(a) suggests to combine them with Scope or Eventual persistency, which results in high throughput and low tail latency. Some of these applications aggregate data from thousands of anonymous users and, therefore, the loss of a certain amount of recent data is acceptable.

Applications that want to attain both reasonable consistency guarantees and high performance such as photo sharing and news readers [38, 44] often use Causal consistency. In this case, Figure 6(a) suggests to use Synchronous persistency. In fact, the figure shows that Causal consistency delivers some of the best performance of all cases in combination with multiple persistency models. Therefore, developers can select the appropriate persistency model based on the data durability requirements of their application.

Applications that require transactional guarantees [15, 19, 46, 75] such as Google's globally distributed Spanner database [15] use a form of Transactional consistency. This consistency model can deliver high throughput, but its performance suffers if transaction conflicts are frequent. In this case, Figure 6 shows that using Read-Enforced persistency is not a good choice, since reads end-up suffering long stalls. Other persistency models such as Scope or Eventual should be used.

Many systems use hybrid consistency models [1, 36, 40, 56, 66]—e.g., Linearizable or Read-Enforced consistency in a local cluster, and Eventual consistency across the entire distributed system in a data center [40]. In this case, our results suggest using Scope or Eventual persistency for the local cluster, and Synchronous persistency across the system.

Generally, we find that Causal consistency combined with either Synchronous or Eventual persistency is highly competitive, and robust to increases in number of clients, network latencies, and write traffic. In these DDP models, reads and writes do not stall.

Beyond this, models combining strong consistency with weak persistency, or weak consistency with strong persistency are typically best. Finally, as RDMA advances improve remote communication, and NVM usage speeds-up durability, companies will increasingly favor stronger consistency models and stronger persistency models, respectively.

Irrespective of the DDP model, a recovery algorithm is invoked on a crash. The complexity of the recovery is higher in the weaker models than in the stricter ones. For example, strict models like <Linearizable, Synchronous> have a simple recovery process because all nodes have the same persistent view of the data. On the other hand, weaker DDP models such as those with Eventual consistency or persistency may need an advanced recovery algorithm, such as a voting-based one [10].

10 RELATED WORK

Distributed Consistency and Persistency Models. Many distributed data consistency models have been studied over the past decades (e.g., [4, 7, 16, 35, 38, 64, 69]). However, few of them have decoupled the discussion of data consistency from data durability. With the advent of NVM, memory persistency models have been proposed [34, 53]. They mainly target a single machine with a hardware-controlled cache hierarchy. Recently, Katsarakis et al. [33] developed Hermes, a broadcast-based replication protocol for in-memory datastores. Hermes uses Linearizable consistency and does not persist data to durable storage. It relies on remote replicas for data recovery, which may cause long recovery delay and even data loss upon full datacenter crashes [22, 79]. Ganesan et al. [22] proposed read-enforced persistency to attain a strong model with low performance overhead. In this work, we decouple consistency from persistency in distributed systems and show how they interact.

Distributed NVM Systems. Recently, many distributed systems have been built based on persistent memory (e.g., [20, 41, 60, 67, 76, 77, 79, 80]). For instance, FileMR [76, 77] developed a distributed NVM file system through RDMA. FaRM [20] implemented a distributed transactional system with battery backed DRAM and RDMA, which supports strict serializability and data durability. Most of these distributed systems follow one of the conventional distributed data consistency models, and develop optimization techniques to reduce remote persistency overhead. We believe our paper will help the future development of such systems by offering insights into DDP models.

Network Support for NVM. To improve the performance of remote data persistency, some architectural techniques have been proposed [26, 63, 65, 76]. Hu et al. [26] present persistence parallelism techniques to improve the network bandwidth utilization using RDMA. Industry plans to extend RDMA to support atomic write and flush operations for NVM [63]. Our work is orthogonal, and can benefit from such hardware. The DDP models we propose can take advantage of network hardware optimizations.

11 CONCLUSION

This paper proposed the concept of Distributed Data Persistency (DDP) model, which is the binding of the memory persistency model with the data consistency model in a distributed system. We reasoned about the interaction between consistency and persistency using the Visibility and Durability points. We designed low-latency distributed protocols for DDP models that combine five consistency models with five persistency models. For the resulting DDP models, we studied the trade-offs between performance, durability, intuitiveness, programmability, and implementability.

We found that, in general, models combining strong consistency with weak persistency, or weak consistency with strong persistency are typically highly competitive. In addition, Causal consistency combined with different memory persistency models is often a good choice.

ACKNOWLEDGMENTS

We thank the reviewers for their comments. This work was partially supported by NSF with grants CNS-1763658, CCF-1919044, and CCF-2107470.

REFERENCES

[1] Marcos K. Aguilera and Douglas B. Terry. 2016. The Many Faces of Consistency. *IEEE Computer Society* (2016).

[2] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue. 2018. The Tofu Interconnect D. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 646–654. https://doi.org/10.1109/CLUSTER.2018.00090

[3] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

[4] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 390–408. https://www.usenix.org/conference/osdi18/presentation/alagappan

[5] Apache. 2020. ZooKeeper. https://zookeeper.apache.org/.

[6] Apache. 2021. Apache Zookeeper. https://github.com/apache/zookeeper.

[7] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

[8] Paolo Bianco. 2017. New Fabric Interconnects: A comparison between Omni-Path and EDR Infiniband Architectures. https://agenda.infn.it/event/13040/contributions/17299/attachments/12506/14064/INFN_CCR_2017_-_DELLEMC_v2.pdf

[9] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, retrieved Oct. 7, 2020.

[10] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/paxos-replicated-state-machines-basis-high-performance-data-store

[11] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* (Aug 2004).

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[13] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In

10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). Hollywood, CA.

[16] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company, USA.

[17] Alexandros Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. SABRes: Atomic Object Reads for in-Memory Rack-Scale Computing. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Article 6, 13 pages.

[18] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. 1985. Consistency in a Partitioned Network: A Survey. *ACM Comput. Surv.* 17, 3 (Sept. 1985), 341–370. https://doi.org/10.1145/5505.5508

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[20] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[21] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 673–689. https://www.usenix.org/conference/atc20/presentation/farshin

[22] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. Strong and Efficient Consistency with Consistency-Aware Durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 323–337. https://www.usenix.org/conference/fast20/presentation/ganesan

[23] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 652–665. https://doi.org/10.1109/ISCA45697.2020.00060

[24] Google Code, cpp-btree. 2007. https://code.google.com/archive/p/cpp-btree/.

[25] Hermes Reliable Replication Protocol. 2021. https://github.com/ease-lab/Hermes.

[26] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie. 2018. Persistence Parallelism Optimization: A Holistic Approach from Memory Bus to RDMA Network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 494–506.

[27] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX-ATC'10)*. USENIX Association, USA, 11.

[28] Intel. 2012. Intel Data Direct I/O Technology Overview. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html

[29] Intel. 2018. 3D XPoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[30] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

[31] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[32] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. 2015. Efficient persist barriers for multicores. In *48th Annual IEEE/ACM International Symposium on Microarchitecture*. 660–671. https://doi.org/10.1145/2830772.2830805

[33] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. https://doi.org/10.1145/3373376.3378496

[34] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Toronto, ON, Canada.

[35] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 71–86. https://doi.org/10.1145/2815400.2815416

[36] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA.

[37] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2014. Fence Scoping. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, 105–116. https://doi.org/10.1109/SC.2014.14

[38] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[39] LogCabin. 2020. https://github.com/logcabin/logcabin.

[40] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. Monterey, California.

[41] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-Enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, USA, 773–785.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200.

[43] Yoshinori Matsunobu. 2016. MyRocks: A space- and write-optimized MySQL database. https://engineering.fb.com/2016/08/31/core-data/myrocks-a-space-and-write-optimized-mysql-database/.

[44] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 453–468. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

[45] Mellanox Technologies. 2019. *White paper: Introducing 200G HDR InfiniBand Solutions.* Technical Report 060058WP. 350 Oakmead Parkway, Suite 100, Sunnyvale, CA 94085. 4 pages. https://www.mellanox.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf

[46] 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)*. Broomfield, CO.

[47] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[48] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2008. Rethink the Sync. *ACM Trans. Comput. Syst.* 26, 3, Article 6 (Sept. 2008), 26 pages. https://doi.org/10.1145/1394441.1394442

[49] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[50] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2541940.2541965

[51] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[52] Oracle. 2018. Oracle NoSQL Database: Fast, Reliable, Predictable. An Oracle white paper. https://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf.

[53] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[54] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. *Phase Change Memory: From Devices to Systems* (1st ed.). Morgan & Claypool Publishers.

[55] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *the 36th Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/1555754.1555760

[56] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. 2017. Characterizing and Adapting the Consistency-Latency Tradeoff in Distributed Key-Value Stores. *ACM Trans. Auton. Adapt. Syst.* 11, 4 (Jan. 2017).

[57] Redis. 2020. https://redis.io/.

[58] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011).

[59] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (Jan 2011).

[60] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[61] SNIA. 2019. NVM PM Remote Access for High Availability (Technical White Paper). https://www.snia.org/sites/default/files/technical_work/Whitepapers/NVM-PM-Remote-Access-for-High-Availability.pdf.

[62] Tom Talpey. 2019. RDMA Persistent Memory Extensions. https://www.openfabrics.org/wp-content/uploads/209_TTalpey.pdf.

[63] T. Talpey, T. Hurson, G. Agarwal, and T. Reu. 2020. RDMA Extensions for Enhanced Memory Placement. https://tools.ietf.org/id/draft-talpey-rdma-commit-01.html.

[64] Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., USA.

[65] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. 2018. Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions. *CoRR* abs/1810.09360 (2018). arXiv:1810.09360 http://arxiv.org/abs/1810.09360

[66] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731

[67] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.

[68] Gavrielatos Vasilis, Antonios Katsarakis, and Vijay Nagarajan. 2021. Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/3447786.3456240

[69] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (June 2016), 34 pages. https://doi.org/10.1145/2926965

[70] Werner Vogels. 2008. Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs - Between Consistency and Availability. *Queue* 6, 6 (Oct. 2008), 14–19. https://doi.org/10.1145/1466443.1466448

[71] Werner Vogels. 2010. All Things Distributed. https://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html.

[72] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[73] Zhi Wang, Xiaoliang Wang, Zhuzhong Qian, Baoliu Ye, and Sanglu Lu. 2018. RDMAvisor: Toward Deploying Scalable and Simple RDMA as a Service in Datacenters. *CoRR* abs/1802.01870 (2018). arXiv:1802.01870 http://arxiv.org/abs/1802.01870

[74] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-Enabled IO Stack for Flash Storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 211–226. https://www.usenix.org/conference/fast18/presentation/won

[75] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. Monterey, California.

[76] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, USA, 221–234.

[77] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 111–125. https://www.usenix.org/conference/nsdi20/presentation/yang

[78] Rohit Zambre, Megan Grodowitz, Aparna Chandramowlishwaran, and Pavel Shamis. 2019. Breaking Band: A Breakdown of High-Performance Communication. In *International Conference on Parallel Processing (ICPP)*. Association for Computing Machinery, New York, NY, USA, Article 47, 10 pages. https://doi.org/10.1145/3337821.3337910

[79] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[80] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2694344.2694370