2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

MINOS: Distributed Consistency and Persistency Protocol Implementation & Offloading to SmartNICs

Antonis Psistakis, Fabien Chaix❖, and Josep Torrellas
University of Illinois Urbana-Champaign, USA ❖FORTH, Greece
{psistaki, torrella}@illinois.edu, chaix@ics.forth.gr

***Abstract*—To enable high-performance, programmable, and resilient distributed systems, Distributed Data Persistency (DDP) models provide specific data consistency and persistency guarantees. Since these models target leaderless systems (i.e., systems where any node can initiate requests), they deliver high performance and are scalable. However, they are also more complex.**

In this paper, we develop detailed distributed algorithms for DDP models. They support Linearizable consistency with five different types of persistency. We call these algorithms *MINOS-Baseline* (MINOS-B) and evaluate them on a 5-node distributed machine. Additionally, to improve performance, we also redesign the algorithms to offload them to a new SmartNIC architecture. The resulting system is called *MINOS-Offload* (MINOS-O). The MINOS-O SmartNIC introduces optimizations such as selective data coherence in hardware between host and SmartNIC, message batching, and message broadcasting. Our evaluation shows that offloading is very beneficial. It substantially reduces request latency and increases request throughput for various workloads and number of nodes. For example, compared to MINOS-B, MINOS-O reduces the average end-to-end latency of two microservice functions by 35%.

I. Introduction

A key component of modern cloud infrastructure is distributed storage systems, including key value stores, file systems, and databases [6], [11], [14]. To satisfy user needs, these systems must provide high performance, availability, and resilience to failures [2], [15], [16], [45]. To achieve these goals, they replicate data across multiple nodes. Such data replicas improve performance and allow the system to continue operation even if some of the machines become unavailable. However, supporting replicas requires the system to ensure their consistency when updates occur. For this purpose, different distributed *data consistency* models exist that describe when updates need to become visible to the replica nodes. Still, consistency models by themselves do not guarantee the resilience of the system, as they are not concerned with system recovery on a fault. To support recovery, data need to be persisted to non-volatile media. *Memory persistency* models describe when updates need to be persisted [30], [46].

Over the years, several distributed consistency models have been proposed, including Sequential [4], Linearizable [4], [13], [22], [27], [59], Eventual [60], Causal [5], [37], [38], and Transactional [15], [16], [62]. These models differ in the guarantees of replica consistency that they provide to the user, and offer different trade-offs between performance and programmer intuition. Similarly, several persistency models have been proposed, including Synchronous [30], Strict [46], Read-Enforced [19], Eventual [30], and Scope [30], [34]. Persistency models differ in the data persistency guarantees that they provide to the user, and offer different trade-offs between performance and durability. Finally, Distributed Data Persistency (DDP) models [30] combine consistency and persistency models, by introducing a unified framework for data consistency and persistency.

The operation of different DDP models was described by Kokolis at al. [30] at a relatively high level. An important contribution of the DDP models is that they target *Leaderless* distributed systems [27]. These are systems where *any node in the system* can initiate read or write requests. Compared to leader-based systems, where all write requests need to be initiated by one leader node, leaderless systems deliver higher performance and are scalable. However, they are conceptually more involved. The DDP models have not been fleshed out in detailed algorithms.

In distributed computing systems, network interface cards (NICs) are responsible for the communication between servers and the network [7], [10], [28], [47], [49]. Traditionally, they are implemented as a PCIe card, and come with limited processing capabilities. However, recently, a new class of NICs with substantial processing and storage capabilities called SmartNICs have become available. SmartNICs have been used to offload operations such as packet processing, encryption, and compression from the CPU [35], [41], [48]. They could potentially offload DDP model protocols.

To improve the performance, availability, and durability of distributed systems, this paper develops detailed distributed algorithms for DDP models and, additionally, offloads their operation from the CPUs to SmartNICs. Specifically, we take the DDP models outlined in Kokolis et al. [30] and develop detailed leaderless algorithms. We focus on Linearizable consistency with Synchronous, Strict, Read-Enforced, Eventual, or Scope persistency. We call these algorithms *MINOS-Baseline* (MINOS-B) and evaluate them on a 5-node distributed machine.

Then, we redesign the algorithms to offload them to a new SmartNIC architecture that we introduce. The SmartNIC architecture introduces several optimizations, including selective data coherence in hardware between host and SmartNIC, message batching, and message broadcasting. The resulting DDP algorithms and SmartNIC architecture are called *MINOS-Offload* (MINOS-O).

An evaluation of MINOS-O using simulations shows that offloading is very beneficial. It substantially reduces request latency and increases request throughput for various workloads and number of nodes. For example, compared to MINOS-B, MINOS-O reduces the average end-to-end latency of two microservice functions by 35%. We also evaluate the impact of individual optimizations.

The contributions of this paper are:

• The leaderless MINOS-B algorithms of several DDP models and their evaluation.

• The MINOS-O NIC-offloaded algorithms and SmartNIC.

• A comparative evaluation of the MINOS-B and MINOS-O algorithms.

II. Background

A. Consistency and Persistency Models

Distributed storage systems keep copies of a record (i.e., replicas) in multiple nodes. How these replicas are kept consistent is given by the *consistency* model. Specifically, on a write, strict or strong consistency models update the replicas eagerly, while relaxed or weak models update them lazily—potentially allowing reads to get stale values. Moreover, how updates to these different replicas in volatile memory are made persistent to durable storage is given by the *persistency* model. Specifically, when a replica is updated, strict or strong persistency models persist it eagerly, while relaxed or weak models persist it lazily—risking that a machine failure wipes out the updated value.

A Distributed Data Persistency (DDP) model [30] combines a specific consistency model and a specific persistency model. It is defined by the supported Visibility and Durability points of updates. The Visibility point of an update is the time when the update becomes available for consumption, and is given by the consistency model; the Durability point is the time when the update is made durable and, hence, cannot be wiped out by a failure, and is given by the persistency model. One can have a DDP model that combines strong consistency and weak persistency, or any combination of them.

Kokolis at al. [30] describe various DDP models at a relatively high level, expressing the operations and messages that are needed, but not the detailed algorithms that need to be running on a real machine to support them. The DDP protocols assume an environment where: (i) for simplicity, a record is replicated in all the nodes rather than in a subset of them; and (ii) importantly, read and write requests can be initiated from any node rather than from a single, leader node. The latter assumption makes the protocols *Leaderless*, which are more general, deliver higher performance, and are scalable—although they are more involved.

In this paper, we focus on DDP models that combine one important consistency model (Linearizable [13], [22], [27], [59]) with one of five persistency models (Synchronous [30], Strict [46], Read-Enforced [19], Eventual [30], and Scope [30], [34]). Space constraints prevent analyzing more models.

Messages in DDP Models. Following the Hermes protocol [27], the node that initiates the request is called *Coordinator*, while all the others are called *Followers*. Consider a write request to a record. The Coordinator issues an invalidation (INV) message to all the Followers. The message carries the new data but it initially invalidates the previous version of the record in all the Followers. The Followers respond with an acknowledgment (ACK) message to the Coordinator, confirming that the update is performed in terms of consistency, or persistency, or both. The Coordinator then sends the validation (VAL) message to all the Followers to mark the completion of the transaction. Depending on the consistency and persistency model used, separate ACK or VAL messages may be generated for consistency (ACK_C, VAL_C) and for persistency (ACK_P, VAL_P), respectively.

Brief Model Definitions. We briefly describe the models considered. More detailed descriptions can be found elsewhere [13], [19], [22], [27], [30], [34], [46], [59].

Linearizable Consistency (Lin) enforces a total ordering of writes to volatile state across all nodes and, additionally, requires that all reads and writes be ordered by their timestamps. It is implemented by returning to the client a write response only when all volatile replicas have been updated.

Synchronous Persistency (Synch) mandates that, in a node, a write be persisted when the local volatile replica is updated. When we combine <Lin, Synch>, the response of a write is returned to the client as soon as all the replicas have been updated and persisted. Implementation-wise, this is when all the ACKs are received by the Coordinator.

Read-Enforced Persistency (REnf) mandates that, on a write, all the updated replicas be persisted by the time any of them is read. When we combine <Lin, REnf>, the response of a write is returned to the client as soon as all the replicas have been updated (i.e., the Coordinator has received all ACK_Cs). Once all replicas have been updated and persisted (i.e., the Coordinator has received all ACK_Cs and ACK_Ps), the Coordinator informs all Followers (i.e., it sends VALs). On reception of the VAL, a Follower enables reads to the record.

Eventual Persistency (Event) mandates that the updated replicas be eventually persisted at some point in the future. No read or write is stalled waiting for that time. When we combine <Lin, Event>, the response of a write is returned to the client as soon as all the replicas have been updated.

Scope Persistency (Scope) uses the notion of scopes. A scope is a set of read and write operations. The messages in this model are marked with an additional *sc*, which denotes the scope they belong to—e.g., [INV]sc. At the end of a scope, a client issues the [PERSIST]sc command. The model mandates that, when the response of a [PERSIST]sc is returned to the client, all the updates in the scope have been persisted. When we combine <Lin, Scope>, the response to a write within the scope returns to the client when the write has updated all the replicas. However, the response to the [PERSIST]sc is returned when all the writes in the scope have updated their replicas and persisted them (i.e., the Coordinator received [ACK_P]sc for the [PERSIST]sc from all the Followers).

Strict Persistency (Strict) This is the strictest model. It dictates that a write should be persisted in all the replica nodes by the time the write response returns to the client—possibly even before the replicas in the volatile memories of the replica nodes are updated. In contrast to Synch, it decouples consistency and persistency using two types of acknowledgments (ACK_C and ACK_P). When we combine <Lin, Strict>, the response of a write is returned to the client as soon as all the replicas have been updated and persisted.

The combination of <consistency, persistency> places constraints on when a local read to a record can access the result of a prior update (from any node) to the record. Specifically, because of Lin consistency in all the models discussed in this paper, it is required that the update be completed across all nodes consistency-wise. This is known at the Coordinator upon the reception of all consistency ACKs, and known at a Follower upon the reception of the consistency VAL. Further, because of the persistency model, there are additional requirements for two of the models. Specifically, for <Lin, Synch> and <Lin, REnf>, it is also required that the update be completed across all nodes persistency-wise. In <Lin, Synch>, this is known at the Coordinator upon the reception of all ACKs, and known at a Follower upon the reception of the single VAL; this requirement matches the consistency one. In <Lin, REnf>, this is known at the Coordinator upon the reception of all ACK_Cs and all ACK_Ps, and known at a Follower upon the reception of the single VAL; this requirement adds to the consistency one.

B. SmartNICs

SmartNICs are NICs with programmable logic that offloads some of the processing tasks from the CPU. They have recently been introduced to improve performance. They usually contain low-end energy-efficient computing units with caches and an on-board DRAM. Typically, SmartNICs handle network-related tasks or completely offload network processing workloads [47], [56]. They can also offload complex tasks such as replication and persistency [35].

III. MINOS-Baseline Algorithms

In this section, we introduce detailed algorithms for the efficient implementation of the DDP model protocols outlined by Kokolis at al. [30]. We call these algorithms *MINOS-Baseline* (*MINOS-B*). As already mentioned, we focus on models that combine Lin consistency with one of five persistency models. Next, we first introduce some definitions and then describe the protocol algorithms for writes and for reads.

A. Definitions

Our algorithms use *Locks* and *Logical Timestamps*. In this section, we define them and describe their use. In our description, we refer to two types of writes. A write transaction (i.e., operation) initiated by a client is called a "client-write". During the transaction, there are substeps that involve updating the local memory subsystem of a node (e.g., the last-level cache (LLC)); we call such writes "local-writes".

Locks. Our algorithms use two types of locks for the two types of writes: read locks (*RDLock_Owner* or *RDLock* for short) for client-writes and write locks (*WRLock*) for local-writes.

Figure 1(a) shows the metadata associated with a data record. RDLock_Owner tells whether the read lock for the record is taken and, if so, who the read lock owner is; WRLock tells whether the write lock for the record is taken. The other three fields relate to the timestamps.

RDLock_Owner	WRLock	volatileTS	glb_volatileTS	glb_durableTS

(a) Record Metadata

node_id	version

(b) Timestamp Format

Fig. 1 – Record metadata and timestamp format.

The locks of a record are used as follows. In a given node, if one or more threads are attempting to perform a client-write (initiated locally or remotely) on a given record concurrently, one of them holds RDLock_Owner for the record. A taken RDLock_Owner prevents concurrent read transactions from accessing the record.

Assume that there is a single thread that is performing a client-write on the record. Such thread holds RDLock_Owner. During a client-write transaction, there is point when the thread, say *T1*, updates the local record (i.e., it performs a local-write). During this time, *T1* must grab and hold the WRLock to ensure that no other client-write transaction to the same record tries to write concurrently. Once *T1* finishes the update of the local record, it releases the WRLock and continues with the client-write operation. Once *T1* completes the client-write operation, if it still holds RDLock_Owner, it releases it.

It is possible that a second thread, say *T2*, wants to perform a concurrent client-write on the same record while *T1* is executing its client-write. If *T2* has a higher timestamp (i.e., comes later in time), our algorithm allows *T2* to "snatch" RDLock_Owner from *T1*, by updating RDLock_Owner. *T1* continues execution and can even attempt and grab the WR-Lock. However, as we will see later, *T1* will not be able to update the local record to a stale value. The benefit of snatching RDLock_Owner is that it will ensure that *T2*'s completion will not be delayed by *T1*'s completion. *T2* is the only thread that can release RDLock_Owner. When it does, the record can be read by other threads.

Logical Timestamps. In our design, each data record in the volatile local memory of a node keeps three logical timestamps [31]: volatileTS, glb_volatileTS, and glb_durableTS (Figure 1(a)). *volatileTS* describes the record's version in the local volatile memory; *glb_volatileTS* describes the record's global version in the machine-wide volatile memory as determined by the consistency model; and *glb_durableTS* describes the global version in the machine-wide non-volatile memory as determined by the persistency model. As shown in Figure 1(b), each timestamp is a tuple with a node identifier (node_id) and a version number.

When a node initiates a client-write, it generates a new timestamp for the write. The timestamp's node_id is set to the Coordinator node ID, while the version is set by reading the volatileTS version of the record in the Coordinator node and adding one (without updating volatileTS). The resulting timestamp is included in a field called TS_{WR} in all the messages sent for this client-write. Moreover, TS_{WR} is eventually applied to the timestamp metadata of the record in all the nodes (including the Coordinator). Specifically, in any node, as soon as the volatile record is updated, the volatileTS is set to TS_{WR}. Similarly, once the volatile record is updated across all replicas, the glb_volatileTS is set to TS_{WR}, and when the durable record is updated across all replicas, the glb_durableTS is set to TS_{WR}. Note that TS_{WR} is unique for a write transaction.

Writes to the same record are ordered from older to newer based on their timestamp. Given two writes, the newer one is the one that has the higher version field or, if the versions are the same, the one with the higher node_id.

In our implementation, RDLock_Owner has the same format as a timestamp. Hence, it is a tuple of <node_id, version>. When RDLock_Owner is acquired, it is atomically set to the client-write's TS_{WR}. When RDLock_Owner is released, it is atomically set to <-1, -1>.

Outdated Writes. It is possible that, when a Coordinator or a Follower node is about to process a write transaction *WR1* with timestamp τ_1, the volatileTS timestamp of the local record already has a newer timestamp τ_2, where $\tau_2 > \tau_1$. In this case, we would like to cut *WR1* short: return to the client right away and not inform other nodes (if the local node is the Coordinator) or return an ACK to the Coordinator right away (if the local node is a Follower). In either case, we want to skip updating the record's volatile and non-volatile data.

Before we can do this, however, we need to ensure that the new transaction *WR2* that renders *WR1* obsolete has reached a correct state. There are both consistency and persistency concerns. Consider consistency first, recalling that we are only concerned with Lin consistency. Before cutting *WR1* short, we need to make sure that *WR2* has updated the volatile record in all the nodes and, therefore, no read will see the value before *WR2*. To ensure this, the Coordinator node for *WR2* must have received consistency ACKs from all the nodes in the system, and the Follower nodes for *WR2* must have received a consistency VAL. Note that *WR2* has already updated the local volatile record (as reflected by the volatileTS). Consequently, if the local node is the Coordinator for *WR2*, before cutting *WR1* short, the thread processing *WR1* must spin until all the *WR2* consistency ACKs are received from all the nodes. Similarly, if the local node is a Follower for *WR2*, before cutting *WR1* short, the thread processing *WR1* must spin until the *WR2* consistency VAL has been received. In other words, in both cases, the thread must spin until glb_volatileTS in the local record is updated. We define a primitive to perform this operation called *ConsistencySpin*.

A similar analysis is performed for persistency issues, and we define a primitive called *PersistencySpin*. For brevity, we do not discuss the details.

We also define the primitive *Obsolete*, which compares the timestamp of a client-write to the volatileTS timestamp of the local record. If the client-write has an older timestamp, the primitive returns true; otherwise, it returns false.

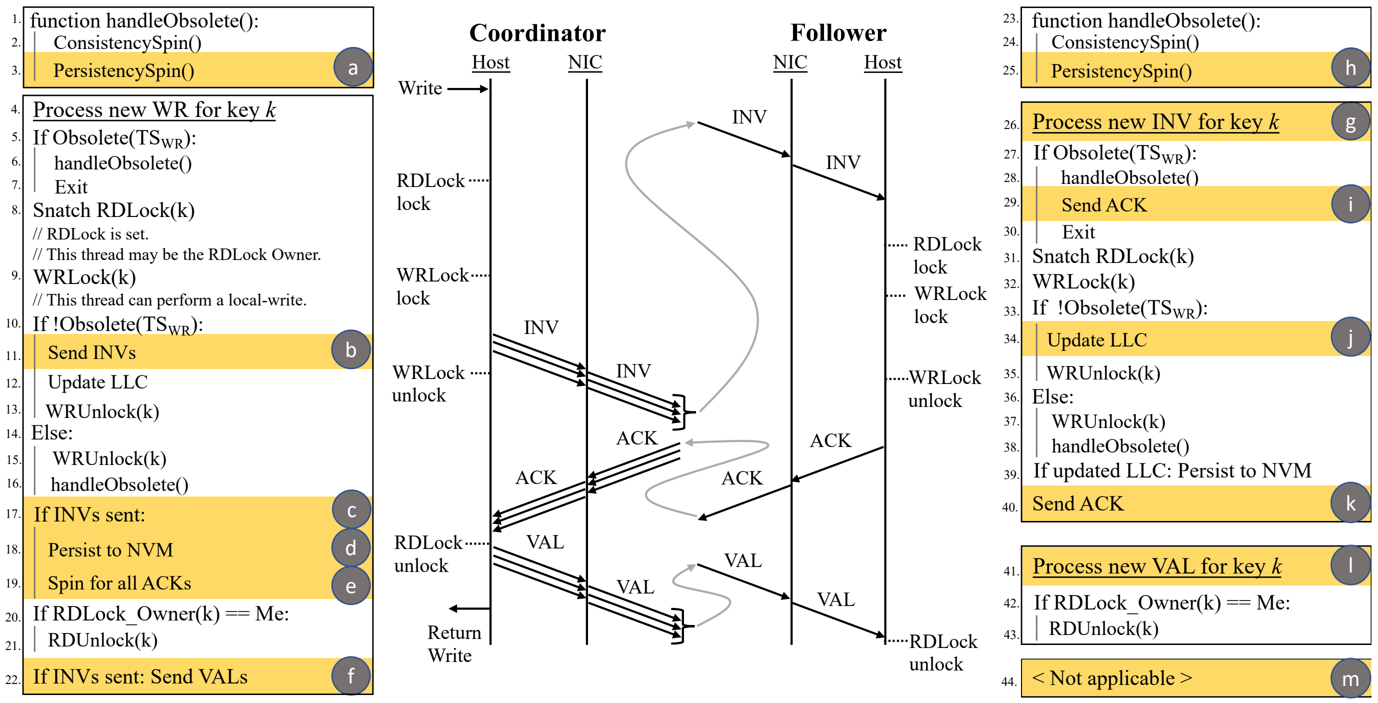
B. Protocol Algorithm for Writes

The write algorithm is similar in different persistency models. Consequently, we describe the algorithm for <Lin, Synch> in detail and then the differences for the rest of the persistency models. Figure 2 shows the algorithmic steps (Coordinator on the left side and Follower on the right side) as well as the messages exchanged in the protocol for <Lin, Synch>. We use a helper function called "handleObsolete" (Lines 1-3, 23-25) to run the ConsistencySpin and the PersistencySpin. We start by describing the Coordinator steps, and then the Follower steps.

Coordinator. When the Coordinator receives a write request WR1 for key (i.e., record) *k* (Line 4), a timestamp TS_{WR} is generated as explained before. The algorithm first checks if WR1 is *Obsolete* (Line 5). If the check returns true, it means that there is another outstanding write WR2 to the same record that is more recent. Hence, WR1 should be skipped and execution should return to the client. Before doing so, however, the algorithm calls *handleObsolete* (Line 6), which performs the *ConsistencySpin* (Line 2) and *PersistencySpin* (Line 3) operations.

If WR1 is not obsolete, it proceeds to perform a "Snatch RDLock" operation on key *k* (Line 8), which consists of the following: (i) if RDLock is free, WR1 grabs the lock; (ii) if RDLock is taken by an older write, WR1 snatches the RDLock from it; and (iii) if RDLock is taken by a younger thread, WR1 simply continues without grabbing the RDLock. This third case is fine because, in the presence of a younger write, we will see that WR1 cannot create an inconsistent state. What is important is that the RDLock be held by the youngest concurrent WR transaction to the record at a time, and that the owner of the RDLock is the one that later releases it. With such arrangement, read transactions to the record are prevented from proceeding concurrently with LLC updates, and LLC updates always produce a consistent state. Note that if WR1 successfully grabs the RDLock, a younger client-write WR2 may later snatch the RDLock from WR1; as we will see, correctness is guaranteed in all cases.

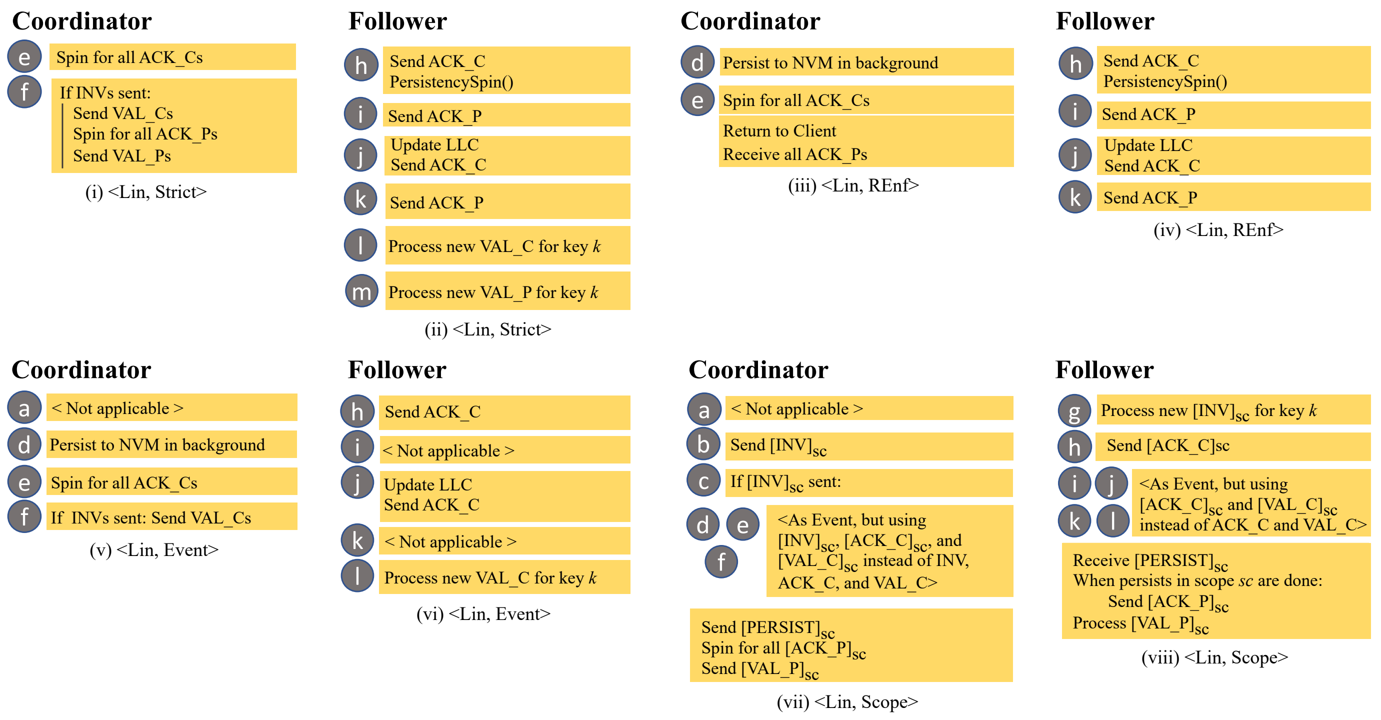
At this point, WR1 spins until it is able to grab the WRLock of key *k* (Line 9). Obtaining WRLock is essential because WR1 is about to perform a local-write to the record. Before performing the local-write, WR1 performs a final timestamp check (Line 10) to see if, since Line 5, WR1 has become obsolete. If WR1 remains the latest write, the algorithm sends the INV messages (Line 11) to all Followers and updates the local, volatile state of the record in the LLC (Line 12). When the local volatile state is updated, the local *volatileTS* (not shown)



```
1.  function handleObsolete():
2.  │ ConsistencySpin()
3.  │ PersistencySpin()                                  a

4.  Process new WR for key k
5.  If Obsolete(TS_WR):
6.  │ handleObsolete()
7.  │ Exit
8.  Snatch RDLock(k)
    // RDLock is set.
    // This thread may be the RDLock Owner.
9.  WRLock(k)
    // This thread can perform a local-write.
10. If !Obsolete(TS_WR):
11. │ Send INVs                                          b
12. │ Update LLC
13. │ WRUnlock(k)
14. Else:
15. │ WRUnlock(k)
16. │ handleObsolete()
17. If INVs sent:                                        c
18. │ Persist to NVM                                     d
19. │ Spin for all ACKs                                  e
20. If RDLock_Owner(k) == Me:
21. │ RDUnlock(k)
22. If INVs sent: Send VALs                              f
```

```
23. function handleObsolete():
24. │ ConsistencySpin()
25. │ PersistencySpin()                                  h

26. Process new INV for key k                            g
27. If Obsolete(TS_WR):
28. │ handleObsolete()
29. │ Send ACK                                           i
30. │ Exit
31. Snatch RDLock(k)
32. WRLock(k)
33. If !Obsolete(TS_WR):
34. │ Update LLC                                         j
35. │ WRUnlock(k)
36. Else:
37. │ WRUnlock(k)
38. │ handleObsolete()
39. If updated LLC: Persist to NVM
40. Send ACK                                             k

41. Process new VAL for key k                            l
42. If RDLock_Owner(k) == Me:
43. │ RDUnlock(k)
44. < Not applicable >                                   m
```

Fig. 2 – Detailed algorithm for the <Lin, Synch> model, with the Coordinator on the left side and the Follower on the right side. The steps in the algorithm that may change for different persistency models are highlighted.



Coordinator

- e: Spin for all ACK_Cs
- f: If INVs sent:
  - Send VAL_Cs
  - Spin for all ACK_Ps
  - Send VAL_Ps

(i) <Lin, Strict>

Follower

- h: Send ACK_C
  PersistencySpin()
- i: Send ACK_P
- j: Update LLC
  Send ACK_C
- k: Send ACK_P
- l: Process new VAL_C for key *k*
- m: Process new VAL_P for key *k*

(ii) <Lin, Strict>

Coordinator

- d: Persist to NVM in background
- e: Spin for all ACK_Cs
  Return to Client
  Receive all ACK_Ps

(iii) <Lin, REnf>

Follower

- h: Send ACK_C
  PersistencySpin()
- i: Send ACK_P
- j: Update LLC
  Send ACK_C
- k: Send ACK_P

(iv) <Lin, REnf>

Coordinator

- a: < Not applicable >
- d: Persist to NVM in background
- e: Spin for all ACK_Cs
- f: If INVs sent: Send VAL_Cs

(v) <Lin, Event>

Follower

- h: Send ACK_C
- i: < Not applicable >
- j: Update LLC
  Send ACK_C
- k: < Not applicable >
- l: Process new VAL_C for key *k*

(vi) <Lin, Event>

Coordinator

- a: < Not applicable >
- b: Send $[INV]_{sc}$
- c: If $[INV]_{sc}$ sent:
- d, e, f: <As Event, but using $[INV]_{sc}$, $[ACK\_C]_{sc}$, and $[VAL\_C]_{sc}$ instead of INV, ACK_C, and VAL_C>

Send $[PERSIST]_{sc}$
Spin for all $[ACK\_P]_{sc}$
Send $[VAL\_P]_{sc}$

(vii) <Lin, Scope>

Follower

- g: Process new $[INV]_{sc}$ for key *k*
- h: Send [ACK_C]sc
- i, j, k, l: <As Event, but using $[ACK\_C]_{sc}$ and $[VAL\_C]_{sc}$ instead of ACK_C and VAL_C>

Receive $[PERSIST]_{sc}$
When persists in scope *sc* are done:
  Send $[ACK\_P]_{sc}$
Process $[VAL\_P]_{sc}$

(viii) <Lin, Scope>

Fig. 3 – Differences of Strict, REnf, Event, and Scope persistency over Synch persistency (Figure 2). If a letter is not present, it means that the step does not change from Figure 2.

is also updated. After these steps, the WRLock is released (Line 13). Instead, if WR1 has been made obsolete by another client-write, WR1 first releases the WRLock (Line 15) and then calls *handleObsolete* (Line 16). Releasing WRLock as soon as possible improves performance, as a concurrent write can grab it while WR1 performs additional operations. Holding the

WRLock is not needed when handleObsolete() is invoked.

If INVs were sent (Line 17), the update is persisted to non-volatile memory (NVM) (Line 18). Moreover, the thread spins until all ACKs are received (Line 19). Note that, while the volatile state is always updated in increasing order of write TS_{WR}, the NVM can be updated by writes out of order. This is acceptable because we use a log structure for the persists.

When all ACKs are received, we know that the consistency and persistency requirements are satisfied across all nodes, hence the glb_volatileTS and glb_durableTS are also updated. Therefore, if WR1 remains the RDLock_Owner (which means that there is no other more recent write to the same record in the node), the read lock can be released (Lines 20-21). Finally, if INVs were sent earlier (in Line 11), VALs are now sent to all Followers (Line 22), which mark the completion of the write transaction. After this step, execution returns to the client.

Follower. When a Follower receives the INV for a write request WR1 to key k and timestamp TS_{WR} (Line 26), the algorithm performs the following steps. First, it checks if WR1 is obsolete (Line 27). If so, it calls *handleObsolete* (Line 28 and Lines 23-25), and then responds to the Coordinator with an ACK (Line 29) as if the write was done. This completes the transaction for an obsolete WR1 (Line 30). A VAL will be received later on but will be discarded.

If WR1 was not obsolete, the algorithm executes similar steps as in the Coordinator (Lines 31-38), namely, WR1: (i) performs a "Snatch RDLock" operation on key k, (ii) grabs the WRLock, (iii) checks again if WR1 is obsolete and it either updates the local state in the LLC (and the *volatileTS*) and releases the WRLock, or releases the WRLock and invokes *handleObsolete*. Then, if the LLC was updated (Line 34), WR1 persists the update to NVM (Line 39). Next, WR1 responds to the Coordinator with an ACK, signifying that the write is performed consistency- and persistency-wise (Line 40).

Read transactions in the Follower node cannot yet see WR1's update. Only after the Follower receives the VAL message from the Coordinator (Line 41) can the owner of the RDLock release it (Lines 42-43). Upon the release of the RDLock, any read transaction can read the key. Furthermore, at this point, the glb_volatileTS and glb_durableTS are also updated to reflect that the write is performed in all the replica nodes consistency- and persistency-wise.

C. Algorithm for Writes for Other Persistency Models

In Figure 2, we highlight with a shade and a letter some steps of the <Lin, Synch> algorithm. These are the steps that may change as we keep Lin consistency but move to other persistency models. All other steps remain unchanged. As explained in §II, some models have different INV, ACK, and VAL messages for consistency and persistency enforcement.

Figure 3 shows the changes to the algorithm for the combinations of Lin consistency and other persistency models. The text next to a letter in Figure 3 replaces the text next to the same letter in Figure 2.

For <Lin, Strict> Coordinator (Figure 3(i)), instead of using ACKs and VALs, the Coordinator spins for ACK_Cs in Step *e* and then, in Step *f*, sends VAL_Cs, spins for ACK_Ps, and sends VAL_Ps. Similarly, at the Follower side (Figure 3(ii)), instead of ACKs, the Follower sends ACK_Cs and ACK_Ps. First, if the received client-write is obsolete (Line 27), the algorithm performs ConsistencySpin(), sends ACK_C, and then performs the PersistencySpin() and sends ACK_P (Line 29). If the client-write is not obsolete, after the LLC is updated, ACK_C is sent, and after the update is persisted to NVM, ACK_P is sent. Later, after VAL_C is received, the *RDLock* is released. Finally, after VAL_P is received in Step *m*, the write operation completes.

For the rest of the models, persisting the update to NVM is performed outside of the critical path. Consequently, Step *d* in the Coordinator (Line 18) is changed to persist NVM in the background, but line Line 39 in the Follower needs no change because the operation is not in the critical path: ACK_C has already been sent.

In <Lin, REnf> (Figure 3(iii)), Step *e* of the Coordinator involves waiting for all ACK_Cs and then returning to the client. In the meantime, ACK_Ps will arrive. When all ACK_Ps are received, the RDLock is released and the VALs are sent. At the Follower side (Figure 3(iv)), the only difference compared to *Strict* is that, since there is only one type of VAL, there is no change for Step *l* and there is no Step *m*.

In the rest of the models, the algorithm does not need to perform PersistencySpin() in Step *a* or Step *h*. This is because, in these weak models, accesses do not need to stall for the persist of prior writes that are still outstanding. However, ConsistencySpin() remains because of the Linearizable consistency.

In <Lin, Event> (Figure 3(v)), Step *e* of the Coordinator involves waiting for all ACK_Cs. After that, the RDLock is released and then, in Step *f*, VAL_Cs are sent before returning to the client. In this model, persistency will happen *eventually* and, therefore, there is no message exchange to track persistency. At the Follower side (Figure 3(vi)), in all cases, the Follower sends ACK_Cs. Later, it receives a VAL_C. No persistency-related messages are exchanged.

In <Lin, Scope>, the protocol steps are similar to <Lin, Event>, although the message names are different. In addition, there is the new [PERSIST]sc transaction, shown in boxes without letters in Figures 3(vii) and (viii). The [PERSIST]sc transaction for scope *sc* consists of the following messages. The Coordinator sends the [PERSIST]sc to its Followers, spins until it receives all the [ACK_P]sc, and finally sends the [VAL_P]sc to its Followers, marking the end of the [PERSIST]sc transaction. A Follower, when it receives the [PERSIST]sc, it completes persisting all the WR operations inside scope *sc* and the [PERSIST]sc request itself, and then responds with an [ACK_P]sc to the Coordinator. Later, it receives [VAL_P]sc, which terminates the [PERSIST]sc transaction.

D. Protocol Algorithm for Reads

Like write operations, read operations can be initiated from any node. Since, in our environment, all records are replicated in all nodes, all read operations are satisfied locally. A read operation to a record is only stalled when the record's RDLock is taken by a write. Once the RDLock is free, the read operation can proceed. The handling of read operations is the same across all models.

E. Failure Detection and Recovery

While the DDP designs [30] do not discuss failure detection or recovery, MINOS-B has protocol extensions for failure detection and recovery that are similar for all the models considered. We assume that nodes can fail due to a crash or network disconnection and that, eventually, such nodes are re-inserted back into the cluster.

Failure detection is attained with timeout mechanisms [27], [45] that identify the non-responding node(s) (F) and alert all the other nodes. Later, when F is re-inserted back into the cluster, we need to bring F's logs up-to-date. This is done by having a designated node send to F a message with the log of all the updates that have been committed since the time when F stopped responding. F then applies the updates to its local persistent and volatile state. More details of the recovery are left for future work.

IV. MINOS-Baseline Performance

To understand the bottlenecks in the MINOS-B algorithms, we implement and run them on a cluster and measure their performance. We run them on a 5-node cluster where each node has a Xeon E5-2450 processor running at 2.1 GHz, and keep 5 cores busy per node. We use the eRPC [25] library for communication. More details are discussed in §VII.

We measure the average latency of a write transaction (Figure 2). This is because, in our environment, write operations are much costlier than reads. We divide the latency into *communication* and *computation* times. Communication is the time taken by all the messages between hosts in a write transaction. For a given message, communication time starts after the sender host has deposited the message in its host send queue in memory, and finishes when the message has been deposited in the host receive queue in the memory of the receiver node. The communication time then includes the transfer of the message from the host send queue through the PCIe bus to the sender NIC, the NIC actions to send the message, the transfer of the message through the network, and the equivalent operations at the receiving NIC. The rest of the transaction is computation time, and can include LLC and NVM accesses.

Roughly speaking, the communication time in a write transaction is seen in Figure 2 as the time from when the first *INV* is sent (Line 11) until when the last *ACK* is received (Line 19), subtracting the average time it takes for a Follower to handle an *INV* message (Lines 26-40). For other models, the communication time may be accounted for slightly differently.

Figure 4 shows the average latency of a write transaction broken down into communication and computation times for the different <consistency, persistency> models. We see that the models with the more conservative persistency enforcement have higher write latencies. This is mostly because of their higher computation times—which are greatly affected by the overhead of persisting a record in the critical path of a write operation in the conservative persistency models. In addition, as multiple writes are sometimes trying to access the same record concurrently, locks add overhead.

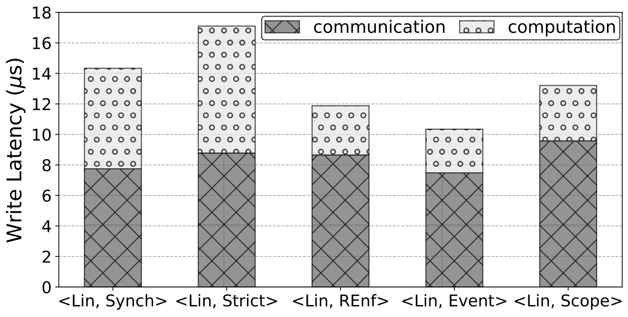


Fig. 4 – Average latency of a write transaction for different <consistency, persistency> models.

The communication time varies less across models. However, it is the highest contributor to the write latency, contributing 51–73% to each model's total write time. A major reason why communication time is so high is that the multiple INV messages in a transaction are sent one at a time. Indeed, they are taken one at a time from the send queue, transferred along the slow PCIe bus, and then sent out to the network. The same happens for the VAL messages. Current NICs lack the support to process batched messages or to use (true) broadcasting [51] of messages in the network. As we scale the protocols to many nodes, this bottleneck is likely to get worse.

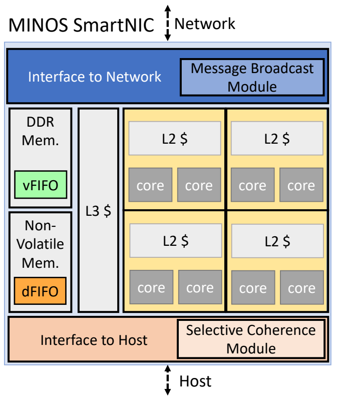
V. MINOS-Offload Design

To reduce the latency of write transactions in MINOS-B, we propose to offload supporting the consistency and persistency model protocols from the host CPU to a SmartNIC. Based on the characterization of the latency of write transactions from Section IV, we propose modified algorithms to enforce consistency and persistency models and a SmartNIC architecture to support them. We call the algorithms and the SmartNIC architecture MINOS-Offload (*MINOS-O*). MINOS-O substantially reduces both the communication and the computation time of write transactions. In this section, we outline the SmartNIC architecture, present mechanisms to minimize communication and computation time, and describe the overall MINOS-O algorithms to support <consistency, persistency> models. The same SmartNIC design supports all the models discussed in this paper.

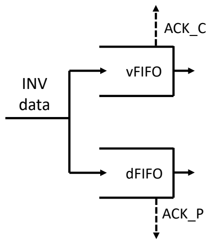
A. MINOS-O SmartNIC Architecture

The MINOS-O SmartNIC architecture is broadly based on the Mellanox Bluefield Data Processing Unit (DPU) [40]. Figure 5(a) shows the top level diagram. Like other SmartNICs, MINOS-O includes multiple cores, three levels of caches, a DRAM module, and interfaces to the network and the host.

The new components that MINOS-O adds are: (i) the host interface is augmented with a module that provides some selective cache coherence between the L3 cache of MINOS-O and the L3 cache of the host; (ii) the network interface includes a module that supports message broadcast; (iii) the NIC has an NVM module that includes a durable FIFO queue (dFIFO) for the persistency protocol; and (iv) the DRAM module includes a volatile FIFO queue (vFIFO) for the consistency models.



(a) Top level diagram



(b) Parallel insertion of INV data to FIFOs

Fig. 5 – MINOS-O SmartNIC architecture.

B. Mechanisms to Improve Performance

We modify the distributed consistency and persistency algorithms of §III to leverage the SmartNIC architecture. In this section, we describe the new mechanisms proposed to improve performance.

1. Offloading Operations to the SmartNIC. We offload the execution of most of the Coordinator and Follower write algorithms of Figure 2 from the host to the SmartNIC (SNIC). Figure 6 shows the high-level operation. The algorithm changes little from that in Figure 2. On the Coordinator side, the host starts the write transaction (Lines 1-8 in Figure 2). After snatching the RDLock, it sends the multiple INVs to the SmartNIC. The SmartNIC takes over starting at Line 9 and executes the rest of the algorithm. Every time an ACK is received, it is passed to the host. Once the SmartNIC has received all of the ACKs, it proceeds to release the RDLock and send the VALs.

In the Follower, all the operations (Lines 23-44) are performed in the SmartNIC. The host is not invoked. Note that a SmartNIC can reject a request from its local host or from the network if it runs out of resources.

With this support, MINOS-O reduces the data and control transfers between SmartNIC and host in the Follower and, to a lesser extent, in the Coordinator. The result is that both the communication and the computation overheads of write transactions in Figure 4 decrease.

2. Coherence between Host and SmartNIC. When we offload some operations of the algorithm to the SmartNIC, we need to ensure that data which can be accessed concurrently

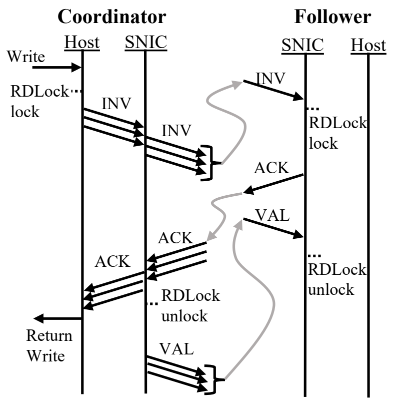


Fig. 6 – Offloading operations to the SmartNIC.

by host and SmartNIC is kept coherent. MINOS-O enables fast sharing of some data structures by keeping them coherent between host and SmartNIC in hardware.

There are only four types of data structures that need to be kept coherent. They are four of the metadata fields of data records shown in Figure 1(a): RDLock_Owner, volatileTS, glb_volatileTS, and glb_durableTS. The WRLock is not included because, as we will see, MINOS-O does not use it.

Recall that RDLock_Owner (also called RDLock for short) for a record in a node is set when there is at least one ongoing write transaction to the record locally. As shown in Figure 6, there are four places in a write transaction where either the host or the SmartNIC can update RDLock. In addition, when a host receives a read transaction, it needs to check RDLock. MINOS-O enables fast access by providing coherence in hardware.

The volatileTS, glb_volatileTS, and glb_durableTS timestamps of a record are set in the Coordinator and Follower algorithms when the record in local volatile storage is updated, when the consistency of the record is enforced across all replicas, and when the persistency of the record is enforced across all replicas, respectively (§III-B). In MINOS-O, the code that updates these timestamps is executed by the SmartNIC.

These timestamps are read when the algorithm checks for the obsoleteness of a write (i.e., *Obsolete*(TS_{WR})) and when performing the consistency and persistency spinning (i.e., *handleObsolete()*) (§III-B). In MINOS-O, the code that reads these timestamps is executed by the host and by the SmartNIC.

To provide hardware coherence for these four data structures, MINOS-O places them in a special range of addresses mapped to a small on-chip memory in both host and SmartNIC. A dedicated bus supports MSI snoopy coherence between these two memories. This hardware is logically placed in the Selective Coherence Module of Figure 5(a).

3. Message Batching and Broadcasting. A source of overhead in distributed protocols that replicate data in multiple nodes is that, on a write operation, messages have to be sent to update all the replicas. MINOS-O minimizes this overhead by supporting two mechanisms that have been proposed in other contexts. The first one focuses on minimizing the many messages that the Coordinator's host and SmartNIC exchange through the slow PCIe bus. Indeed, as shown in Figure 6, the host sends as many INV messages as Followers, and the SNIC responds with as many ACKs as Followers. To reduce this overhead, MINOS-O performs *batching* [24], [47], [52], whereby the host sends a single INV message with information about which nodes should receive it, and the SmartNIC responds with a single ACK message when it has received all the ACKs. Later, the SmartNIC sends the VALs without host involvement.

The second mechanism addresses the SmartNIC overhead of preparing and sending the same INV and VAL messages to multiple Followers. Rather that considering these messages as completely different messages, MINOS-O provides special hardware and an RDMA verb [51] that broadcasts a message. Specifically, the SmartNIC deposits an INV or VAL message into the Send Buffer only once, and fills a Destination Map register. Then, an FSM broadcasts the message to all the destinations. This hardware is logically placed in the Message Broadcast Module of Figure 5(a). In contrast, the baseline approach requires depositing the same message on the Send Buffer multiple times.

4. Eliminating Write Locks. In the Coordinator and Follower algorithms of Figure 2, a thread must grab the WRLock for a record before it can update the local version of the record in local volatile memory. Grabbing the WRLock is needed for two reasons. First, it prevents record corruption when more than one thread are trying to update the same record concurrently. Second, locking is also needed to safely identify an obsolete write to the local volatile memory (thanks to the *Obsolete(TS_{WR})* check), and avoid it.

Note that the update to the local non-volatile log is outside the WRLock critical section. The reason is that updates are deposited into the log in an atomic fashion. It is possible that entries are inserted into the log in an out-of-order manner, therefore creating obsolete entries. However, correctness is maintained because, before the log entries are applied to the non-volatile database, they are checked for obsoleteness.

Spinning on, grabbing, and releasing the WRLock adds overhead. To eliminate this overhead, MINOS-O adds special hardware that serializes the updates to the same record in the local volatile memory and skips obsolete updates to the local volatile memory. This hardware is the vFIFO queue. There is also the dFIFO queue that persists the updates locally in the SmartNIC, avoiding the need to push them to the host in the critical path. Both vFIFO and dFIFO queues are shown in Figure 5(b).

When a thread executing the Coordinator algorithm wants to write to a record, rather that grabbing the WRLock, it directly checks whether the write is obsolete. If it is not, the SmartNIC sends the INVs and writes to the vFIFO queue atomically and to the dFIFO queue atomically. When the hardware dequeues an entry from the vFIFO queue in the background, it checks for obsoleteness before updating the LLC. If the entry is not obsolete, a DMA operation pushes the update to the host's LLC. Dequeueing can be done in parallel for updates to different records. When the hardware dequeues an entry from the dFIFO, it pushes it to the host NVM log. A thread cannot proceed to unlocking the RDLock until (1) the update has drained from the vFIFO to the LLC, and (2) all the ACKs have been received. There is no need to wait for the update to drain from the dFIFO because the update is already durable.

When a thread executing the Follower algorithm in the SmartNIC wants to write to a record, the process is similar. First, it checks whether the update is obsolete. If it is not, it writes to the vFIFO queue atomically and to the dFIFO queue atomically. Finally, it sends the ACK. When the hardware dequeues the entry from the vFIFO queue, it checks for obsoleteness before updating the LLC; when the hardware dequeues an entry from the dFIFO, it pushes it to the host NVM log. After the update has drained from the vFIFO and the VAL has been received, the thread can proceed to unlocking the RDLock.

C. Overall MINOS-O Algorithms

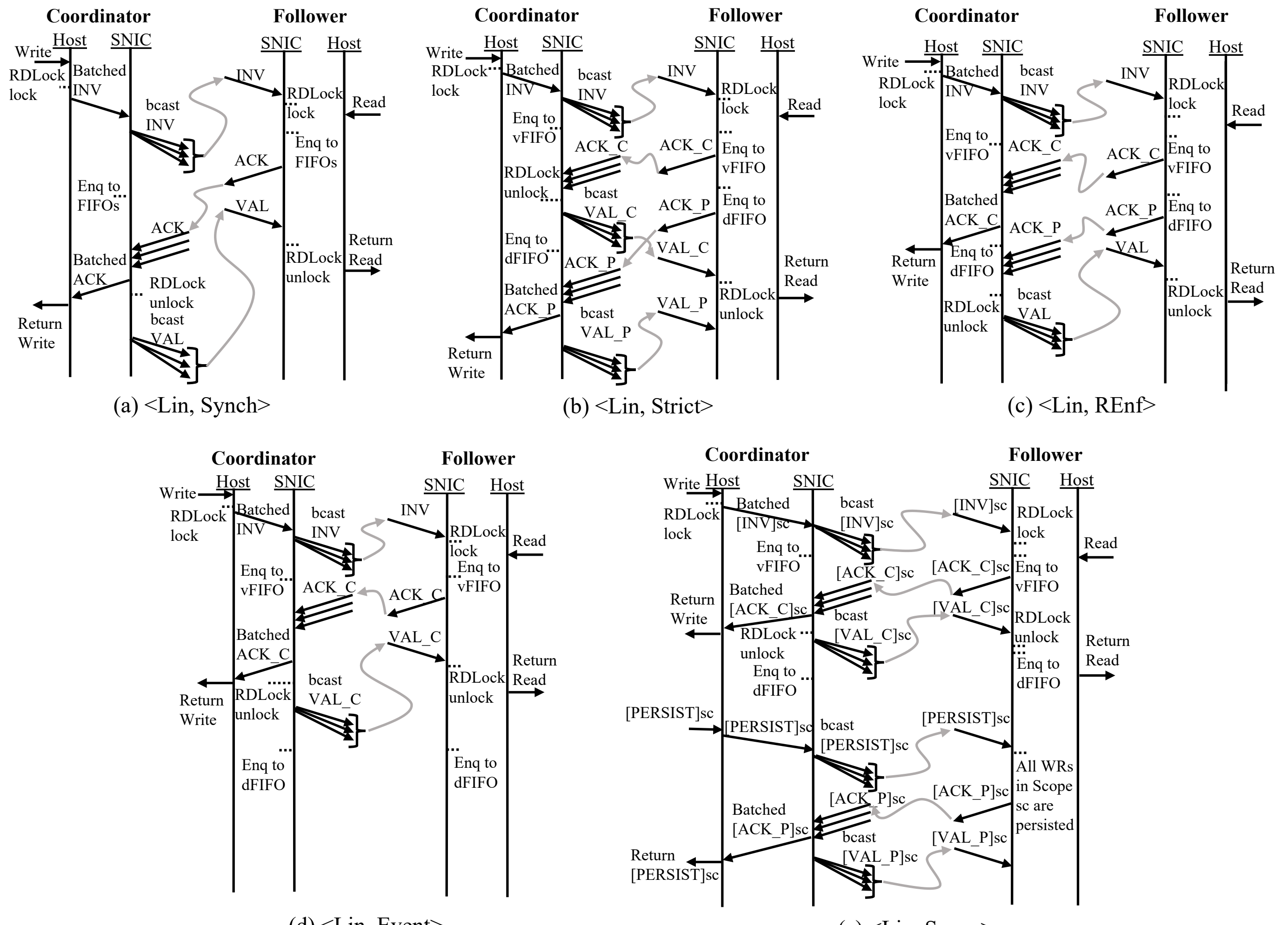
When all the mechanisms are applied together, we obtain the MINOS-O algorithm. Figure 7(a) shows the message exchanges of the MINOS-O algorithm for a write and a read using the <Lin, Synch> model. As shown in the figure, for a client-write, the Coordinator host grabs the RDLock and sends a batched INV message to its SNIC. The SNIC broadcasts the INV message to all the Followers. Afterwards, the SNIC enqueues the update in vFIFO and dFIFO. Later, when all the corresponding ACKs are received, the Coordinator's SNIC sends a batched ACK to the host, which marks the end of the client-write operation. After the corresponding data entry is drained from vFIFO, the SNIC releases the RDlock and broadcasts the VALs to all the Followers.

Figure 7(a) also shows that, in a Follower, when the SNIC receives an INV, a thread grabs the RDLock, enqueues the update in vFIFO and dFIFO, and returns an ACK. Later, when the Follower receives a VAL and the corresponding data entry is drained from vFIFO, the Follower releases the RDLock. In both Coordinator and Follower, a client-read can only proceed when the RDLock is free.

Figure 8 shows the detailed MINOS-O algorithm for the Coordinator and Follower for <Lin, Synch>. The algorithm is organized as Figure 2 with the four MINOS-O optimizations applied.

D. Supporting Other Persistency Models

Figures 7(b)–(e) show the algorithms for the other persistency models. They are similar to <Lin, Synch> except for the following differences. First, ACKs and VALs can be for consistency (ACK_C, VAL_C) or persistency (ACK_P, VAL_P). Second, different models have slightly different logic, as discussed in Section III-C and shown in Figure 3. For example, different algorithms use different conditions to release the RDLock in the Coordinator and in the Follower, and to return to the client.



(a) <Lin, Synch> (b) <Lin, Strict> (c) <Lin, REnf> (d) <Lin, Event> (e) <Lin, Scope>

Fig. 7 – Timeline of the MINOS-O algorithms for consistency and persistency models using SmartNICs (SNICs).

Note that, in some cases, we have separated the enqueuing to the vFIFO from the enqueuing to the dFIFO. We do this when only the enqueuing to the vFIFO is in the critical path and, therefore, waiting for the slower enqueuing to the dFIFO in the critical path in unnecessary. For the [PERSIST]sc command of the <Lin, Scope> model, we do not include the bookkeeping operations in the volatile and non-volatile memory in order to simplify the picture.

VI. Protocol Verification

To verify the correctness of the MINOS-B and MINOS-O protocols, we use the TLA+ formal specification and verification language, and model-check the protocols in TLC [32]. Our TLA+ work draws upon the Hermes [27] TLA+ design. With TLA+, we specify all the states and possible actions from all the states, and then check that certain correctness conditions hold. We model-check all <consistency, persistency> models analyzed.

Table I shows the correctness conditions checked for all the <consistency, persistency> models analyzed. We first check two concurrency conditions: no deadlock and no livelock. Next, for data consistency correctness, we check three invariants related to the values of the volatileTS and glb_volatileTS metadata of records. Next, for data persistency correctness, we check two invariants involving the value of glb_durableTS. Finally, we perform a set of type checks that ensure: a) only legal messages are sent, b) record metadata and locks take only legal values, and c) bookkeeping data in write operations take only legal values.

VII. Methodology

We evaluate MINOS-B using a distributed machine. Since the proposed MINOS-O hardware does not exist, we evaluate MINOS-O and also MINOS-B (for comparison) using a simulated distributed machine.

Distributed Machine. We implement MINOS-B in a 5-node cluster provided by CloudLab [17] with the parameters shown in Table II. Since we do not have access to a modern persistent memory device, we use information from prior works [30], [58], [61] and assume 1295ns to persist 1KB of data. One way to exchange messages between nodes is to use one-sided Remote Direct Memory Access (RDMA), which is the state-of-the-art paradigm for server-to-server communications in datacenters [1], [20], [57], [65]. However, in

```
Coordinator

function handleObsolete():
    ConsistencySpin()
    PersistencySpin()

Host: Process new WR for key k
If Obsolete(TS_WR):
    handleObsolete()
    Exit
Snatch RDLock(k)
// RDLock is set.
// This thread may be the RDLock Owner.
If !Obsolete(TS_WR):
    Send Batched INV (to SNIC)
Else:
    handleObsolete()
If Batched INV sent:
    Spin for Batched ACK

SNIC: Process Batched INV
Broadcast INV
Enqueue to vFIFO and dFIFO

SNIC: Process ACK
If all ACKs are received
    Send Batched ACK to Host
    Wait for data to drain from vFIFO
    If RDLock_Owner(k) == Me:
        RDUnlock(k)
    Send VALs
```

```
Follower

function handleObsolete():
    ConsistencySpin()
    PersistencySpin()

SNIC: Process new INV for key k
If Obsolete(TS_WR):
    handleObsolete()
    Send ACK
    Exit
Snatch RDLock(k)
If !Obsolete(TS_WR):
    Enqueue to vFIFO and dFIFO
Else:
    handleObsolete()
Send ACK

SNIC: Process new VAL for key k
Wait for data to drain from vFIFO
If RDLock_Owner(k) == Me:
    RDUnlock(k)
```

Fig. 8 – MINOS-O algorithm for <Lin, Synch>.

1. Concurrency Checks
Absence of deadlocks and livelocks.
2. Consistency Checks
a) When a record is read-unlocked in all nodes, the volatileTS and glb_volatileTS of the record are the same across all nodes.
b) When all ACKs for consistency have been received for a write to a record, the volatileTS of the record is the same across all nodes.
c) When not all ACKs for consistency have been received for a write to a record, the glb_volatileTS of the record is the same across all nodes.
3. Persistency Checks
a) When a record is read-unlocked in all nodes, the glb_durableTS of the record is the same across all nodes.
b) When not all ACKs for persistency have been received for a write to a record, the glb_durableTS of the record is the same across all nodes.
4. Type Checks
a) Each message ∈ {INV, ACK, ACK_C, ACK_P, VAL, VAL_C, VAL_P, [INV]sc, [ACK_C]sc, [ACK_P]sc, [VAL_C]sc, [VAL_P]sc, [PERSIST]sc}

b) Record metadata:
i) For volatileTS, glb_volatileTS, and glb_durableTS:
version ∈ {0...(MAX_VERS-1)}, node_id ∈ {0...(MAX_NODES-1)}
ii) For RDLock_Owner:
version ∈ {-1...(MAX_VERS-1)}, node_id ∈ {-1...(MAX_NODES-1)}
iii) WRLock ∈ {0, 1}

c) Bookkeeping: ∀ node_id, {RcvedACK_SenderID[node_id], RcvedACK_C_SenderID[node_id], and RcvedACK_P_SenderID[node_id]} ∈ {0...(MAX_NODES-1) \ node_id}

TABLE I – Conditions checked using TLA+ for all the <consistency, persistency> models analyzed.

our protocol, a message received by a node performs multiple operations before triggering a response to the requester node. Using RDMA would require multiple messages to perform these multiple operations, resulting in performance overhead. Hence, we instead use eRPC [25], [26] for communication, which provides performance similar to RDMA.

This distributed machine was used in the experiments of Section IV and to calibrate the parameters of the simulated distributed machine.

Number of nodes	5
CPU per node	Xeon E5-2450 (5 cores, 2.1 GHz)
Main memory per node	16GB of DRAM (DDR3-1600)
NIC per node	Mellanox MX354A FDR CX3
Emulated NVM per node	1295 ns to persist 1KB of data

TABLE II – Distributed machine running MINOS-B.

Simulated Distributed Machine. The simulator we use is SimGrid [8], [9], which is an accurate and scalable simulator for distributed systems. We model a distributed architecture similar to the CloudLab one with 2, 4, 5 (default), 6, 8, 10, or 16 nodes. The various access latencies of the memory hierarchy of the host are set based on measurements of the CloudLab system. The various overheads and latencies in the SmartNIC are set based on measurements of the BlueField-2 HDR100 100Gb/s SmartNIC [44] of the Thor cluster at the HPC-AI Advisory Council Cluster Center [21]. Other parameters of the SmartNIC and of the communication between SmartNIC and host or between SmartNICs are obtained from the literature [35], [39], [43] or from measurements on the CloudLab system.

Some parameters used in the simulator are shown in Table III. The synchronization latency is the average latency to perform a compare-and-swap. The table also shows the latencies to send an INV and an ACK message. Further, it shows the time between consecutive messages when sending the same INV to a set of Followers without broadcasting.

Number of Nodes	2, 4, 5 (default), 6, 8, 10, or 16	
Node	**Host**	**SmartNIC**
Number of Cores	5	8
Core frequency	2.1 GHz	2 GHz
Synchronization latency	42 ns	105 ns
Communication Link	**Latency**	**BW**
PCIe between Host and SmartNIC	500 ns [35]	6.25 GB/s [43]
Network link between SmartNICs	150 ns	7 GB/s [39]
MINOS-O Parameters		
vFIFO & dFIFO latency (wr 1KB)	465 ns and 1295 ns	
vFIFO & dFIFO size	5 and 5 entries	
Send one INV and send one ACK	200 ns and 100 ns	
Time between consecutive msgs	100 ns (with no broadcast support)	

TABLE III – Parameters of the simulated system.

In addition to running the MINOS-O algorithms on the simulated distributed architecture, we also run the MINOS-B algorithms on a simulated model of the CloudLab distributed machine. With our parameters, MINOS-B performs similarly in both the real and the simulated machine.

Workloads Used. To support our proposed metadata format (Figure 1), we implement our own key-value store, named MINOS-KV. Requests are generated using a C++ version of Yahoo! Cloud Serving Benchmark (YCSB) [12], [50]. The back-end in-memory application used is a Hashtable [11].

We use various workloads with different write and read ratios. The database of each node has 100,000 records. The default workload uses a zipfian distribution for keys, has 50% write and 50% read operations, and issues 100,000 requests per node. We use a record size of 1KB, which is the default in YCSB. In some experiments, we use the DeathStar [18] benchmark suite for microservices.

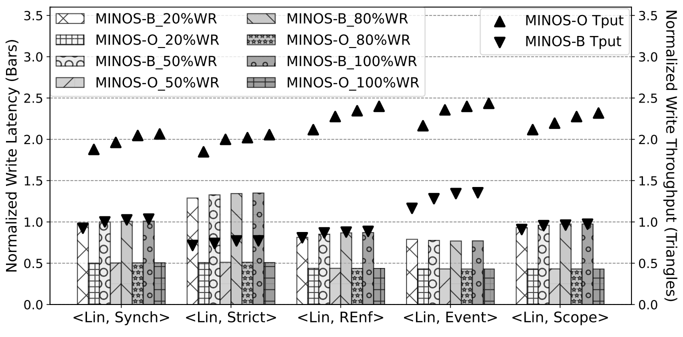
All nodes contain replicas of all records. Therefore, all write operations initiated in a node need to be propagated to all other nodes. Read operations are always local.

VIII. Evaluation

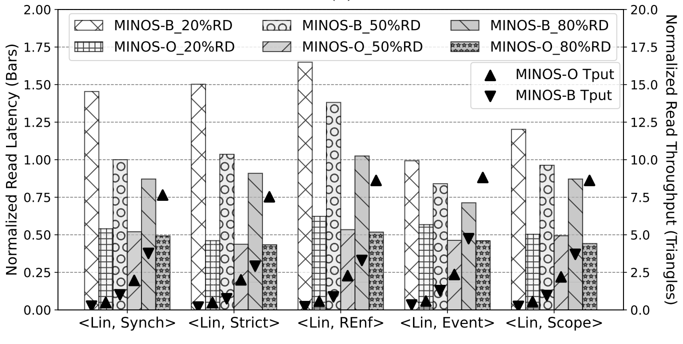
In this section, we compare MINOS-B and MINOS-O under a variety of conditions, evaluate the impact of the MINOS-O optimizations, and perform a sensitivity analysis of some parameters.

A. Comparing MINOS-B and MINOS-O

Figure 9 compares the latency and throughput of client writes (a) and client reads (b) under various conditions in MINOS-B and MINOS-O. In each chart, the latency is represented with bars and is measured in the left Y-axis, while the throughput is represented with triangles and is measured in the right Y-axis. The bars/triangles are organized in groups corresponding to the different <consistency, persistency> models. For each model, the different bars/triangles correspond to workloads with 20%, 50%, 80%, and 100% of writes or reads. In each figure, the bars and triangles are normalized to MINOS-B with <Lin, Synch> and 50% writes or reads.



(a)



(b)

Fig. 9 – Normalized latency (bars and left Y-axis) and throughput (triangles and right Y-axis) of writes (a) and reads (b) for MINOS-B and MINOS-O for different workloads.

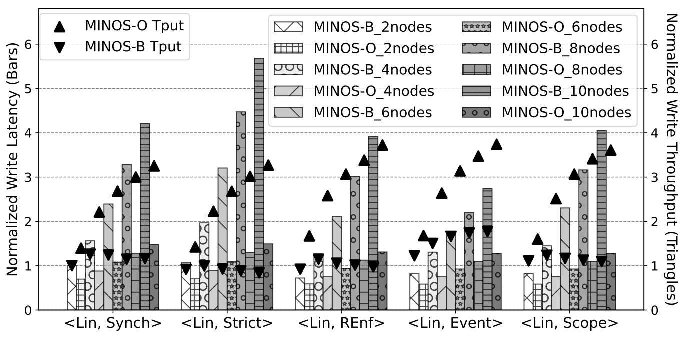
Consider the write transactions first (Figure 9(a)). For all the workloads and <consistency, persistency> models, MINOS-O typically reduces the average write latency by 2-3x over MINOS-B, and increases the average write throughput by 2-3x over MINOS-B. These are major improvements. Also, MINOS-O is much less sensitive to the persistency model than MINOS-B. In addition, as the fraction of writes increases, MINOS-O's throughput increases, but its latency barely changes. In contrast, MINOS-B's throughput typically improves little with higher fraction of writes. Overall, MINOS-O is a high performance scheme, robust to changes in consistency and load.

Consider now read transactions (Figure 9(b)). The trends are similar. MINOS-O often reduces the average latency by 2x or more over MINOS-B, and increases the average throughput by 2x or more over MINOS-B. For both schemes, as the fraction of reads increases, the read latency decreases and the throughput increases. This is due to the high cost of writes.

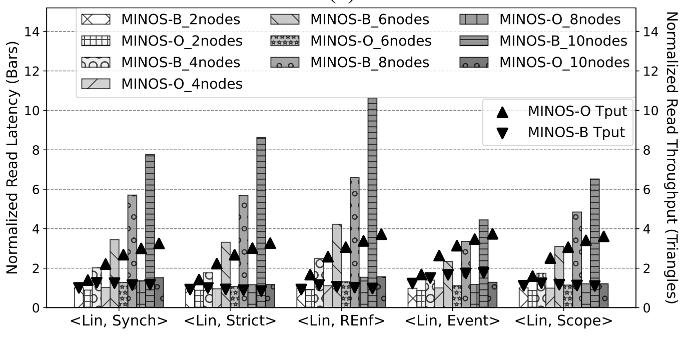
On average across models and workloads, MINOS-O's write and read latency are 2.1x and 2.2x lower than MINOS-B's, respectively; MINOS-O's throughput is 2.3x higher than MINOS-B's for both writes and reads.

B. Comparison for Different Node Counts

Figure 10 compares the latency and throughput of client writes (a) and client reads (b) in MINOS-B and MINOS-O for different node counts (2, 4, 6, 8, and 10). The figure is organized as Figure 9. In each figure, the bars and triangles are normalized to MINOS-B with <Lin, Synch> and two nodes.



(a)



(b)

Fig. 10 – Normalized latency (bars and left Y-axis) and throughput (triangles and right Y-axis) of writes (a) and reads (b) for MINOS-B and MINOS-O for different node counts.

We again see the effectiveness of MINOS-O. As the number of nodes increases, MINOS-O rapidly increases the throughput, while keeping latency increases modest (for writes) or non-existing (for reads). The write latency increases are due to higher contention. In contrast, as the number of nodes increases, MINOS-B increases the latency quickly and is typically unable to improve the throughput.

On average across models and node counts, MINOS-O's write and read latency are 2.3x and 3.1x lower than MINOS-B's; MINOS-O's throughput is 2.4x higher than MINOS-B's for both writes and reads.

C. Comparison for Real Applications

We compare the end-to-end latency of running DeathStar [18] functions on MINOS-B and MINOS-O. We evaluate the *Login* function of the *UserService* microservice in the *Social Network* and *Media Microservices* applications. In each SET and GET operation, we invoke our client-write and client-read algorithm, respectively. We assume a node-to-node round-trip latency of 500μs, which has been measured in datacenters [3]. We model a cluster with 16 nodes.

Figure 11 shows the end-to-end latencies for MINOS-B and MINOS-O. The bars are grouped by <consistency, persistency> model. For each model, there are bars for the two functions and for MINOS-B and MINOS-O. The bars are normalized to <Lin, Synch> MINOS-B and *Social*. From the figure, we see that MINOS-O reduces the end-to-end latency across the board. On average, it reduces the end-to-end latency by 35%.

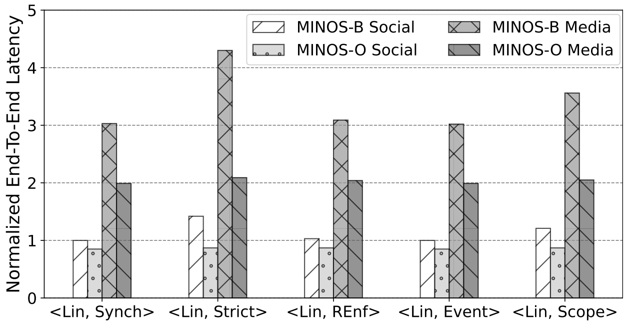


Fig. 11 – End-to-end latency of real applications.

D. Evaluating the MINOS-O Optimizations

To evaluate the impact of the MINOS-O optimizations of §V-B, we group them into three groups: (i) offloading operations to the SmartNIC plus supporting coherence between host and SmartNIC, and eliminating write locks; (ii) message batching; and (iii) message broadcasting. The first group, called *Combined*, combines three optimizations because applying them separately is sub-optimal. Also note that batching can only be beneficial if Combined is applied first.

Figure 12 compares the average write latency of a workload that issues only client write operations for different architectures: MINOS-B, MINOS-B plus broadcast, MINOS-B plus batching, MINOS-B plus Combined (represented as Offl+Coh+WRLock), MINOS-B plus Combined and broadcast, MINOS-B plus Combined and batching, and MINOS-O. The bars are normalized to MINOS-B and the evaluated model is <Lin, Synch>.

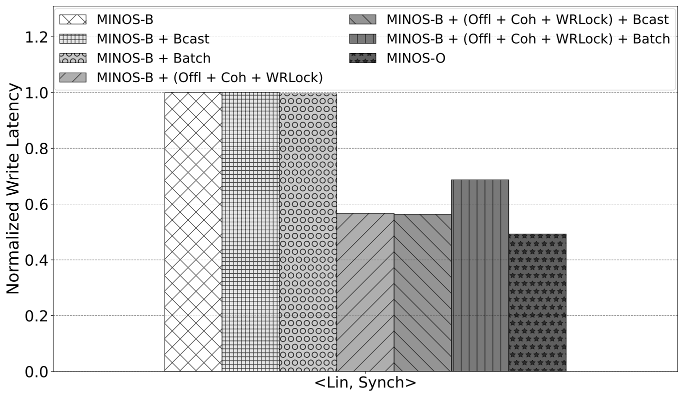


Fig. 12 – Impact of the MINOS-O optimizations.

We see that augmenting MINOS-B with broadcast or batching has no noticeable effect. However, augmenting MINOS-B with the Combined optimization is very effective: the write latency goes down by 43.3%. The reasons are the high efficiency of operation execution in the SmartNIC and the reduced number of host-SNIC communications. Taking this design and adding broadcast barely affects the write latency, as the system is already quite efficient. On the other hand, taking MINOS-B with the Combined optimization and adding batching slows the execution. The reason is that, as a batched message arrives at the SmartNIC, the latter has to unpack it, which adds overhead. However, if we add all the optimizations (creating MINOS-O), batching is beneficial because, with broadcast, the message does not need to be unpacked. Overall, MINOS-O reduces the average write latency by 50.7% over MINOS-B.

E. Sensitivity Analysis

In this section, we perform a sensitivity analysis of different MINOS-O parameters. We use a workload with 50% writes and 50% reads running with <Lin, Synch>, and measure the average write latency. Figure 13 shows the normalized write latency of MINOS-O as we vary the FIFO size. The figure shows bars for 1, 2, 3, 4, 5, and 100 entries in each of vFIFO and dFIFO. The bars are normalized to the MINOS-O write latency with an unlimited number of FIFO entries. We see that, with 3-5 entries, one attains the same average latency as with an unlimited number of them.

Figure 14 compares the average execution time of a write in MINOS-O and MINOS-B. Specifically, it shows the write transaction speedup of MINOS-O over MINOS-B under different values of persist latency, key distribution, and database size. In the first set of bars, we vary the time it takes to persist a 1KB record from 100ns to 100μs. These values are selected to represent current and future durable mediums. For example, today, persisting a 64B cache line to NVM takes about 100 ns [58], and persisting a block to SSD takes about 100 μs [63].

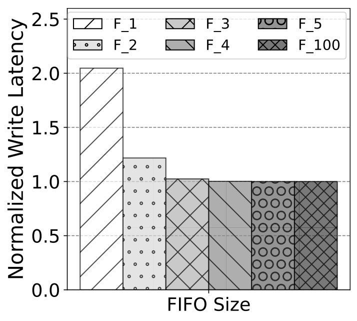


Fig. 13 – Sensitivity to FIFO size.

We can see that MINOS-O speeds-up write transactions in all cases. The speedups increase with the persist latency and are, on average, 2.2x.

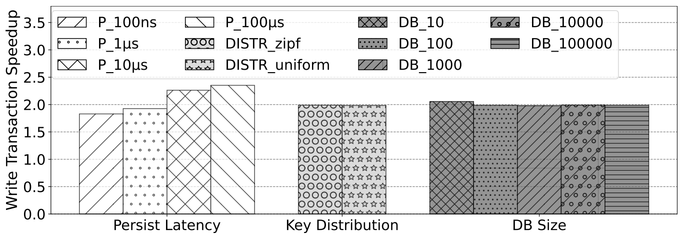


Fig. 14 – Sensitivity to other parameters.

In the second set of bars, we vary the key distribution between zipfian (default) and uniform. In general, with zipfian, we would expect more conflicts on the same keys, therefore reducing performance. As already mentioned, however, our algorithms can support concurrent and conflicting writes in both MINOS-B and MINOS-O. As a result, the figure shows that MINOS-O delivers a speedup of 2x over MINOS-B in both distributions.

Finally, in the third set of bars, we vary the database size from 10 records to 100K records. In general, the smaller the database size is, the more the conflicts that are expected. Since both algorithms handle conflicting writes well, the changes as we increase the database size are minimal. On average, MINOS-O delivers a speedup of 2x over MINOS-B.

IX. Related Work

Consistency and persistency models. There are several key works in this area. Pelley et al. [46] introduce memory persistency models. Ganesan et al. [19] introduce the Read-Enforced persistency model. For consistency models, Hermes [27] proposes a leaderless design for Linearizable consistency. DDP [30], which is closest to our work, introduces a set of combinations of consistency and persistency models and their high-level operation.

NICs in datacenters. NICs can improve performance in datacenters. Caulfield et al. [10] use NICs to enable direct communication between pools of FPGAs. FLOEM [47] introduces a framework to program NIC-accelerated applications. Some works extend RDMA capabilities: Reda et al. [49] support self-modifying RDMA chains, PRISM [7] introduces new primitives, and Hyperloop [28] supports distributed transactions. RAMBDA [64] offloads CPU tasks into a cache-coherent accelerator that can directly interact with RDMA NICs.

Programmable NICs and switches. They are used in datacenters [41], [56]. The NetCache [23] and IncBricks [36] switches cache data in the network, and PMNet [53] also persists data in the network (whether it is a programmable switch or a NIC). Several works optimize/extend existing RDMA primitives using SmartNICs [3], [33], [51], or offload storage operations [42]. In SmartNICs, LineFS [29] supports distributed file systems with support to persist at the client's side, TURBO [54] load-balances light-tailed RPCs, and Xenic [52] accelerates distributed transactions by maintaining locks and hot data at the NIC. iPipe [35] provides a new programming model to offload distributed applications onto SmartNICs, and SKV [55] offloads key-value stores to SmartNICs while persisting at the host. MINOS is different from prior works in that it targets offloading both consistency and persistency protocols to the SmartNIC, and uses rigorous definitions of consistency and persistency models.

X. Conclusion

To enable high-performance, programmable, and durable distributed systems, this paper has developed detailed distributed algorithms for DDP models. The algorithms support Linearizable consistency with five different types of persistency. We call them MINOS-B. Then, to improve performance, we redesigned the algorithms to offload them to a new SmartNIC architecture. The architecture introduces optimizations such as selective data coherence in hardware between host and SmartNIC, message batching, and message broadcasting. The resulting algorithms and architecture are called MINOS-O. Our evaluation of MINOS-O showed that offloading substantially reduces request latency and increases request throughput for various workloads and number of nodes. For example, compared to MINOS-B, MINOS-O reduced the average end-to-end latency of two microservice functions by 35%.

Acknowledgments

This work was supported in part by NSF under grants CNS 19-56007 and CCF 21-07470, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The authors would like to thank Burak Ocalan for his feedback.

References

[1] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue, "The Tofu Interconnect D." in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 646–654.

[2] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Protocol-Aware Recovery for Consensus-Based Storage." in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/alagappan

[3] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker, "Remote Memory Calls." in *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020*, B. Y. Zhao, H. Zheng, H. V. Madhyastha, and V. N. Padmanabhan, Eds. ACM, 2020, pp. 38–44. [Online]. Available: https://doi.org/10.1145/3422604.3425923

[4] H. Attiya and J. L. Welch, "Sequential Consistency versus Linearizability." *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, 1994. [Online]. Available: https://doi.org/10.1145/176575.176576

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency." in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 761–772. [Online]. Available: https://doi.org/10.1145/2463676.2465279

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph." in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, A. Birrell and E. G. Sirer, Eds. USENIX Association, 2013, pp. 49–60. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[7] M. Burke, S. Dharanipragada, S. Joyner, A. Szekeres, J. Nelson, I. Zhang, and D. R. K. Ports, "PRISM: Rethinking the RDMA Interface for Distributed Systems." in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 228–242. [Online]. Available: https://doi.org/10.1145/3477132.3483587

[8] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "SimGrid on GitHub." https://github.com/simgrid/simgrid, 2014.

[9] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms." *J. Parallel Distributed Comput.*, vol. 74, no. 10, pp. 2899–2917, 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.06.008

[10] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture." in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 2016, pp. 7:1–7:13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783710

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data." *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB." in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[13] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a Partitioned Network: A Survey." *ACM Comput. Surv.*, vol. 17, no. 3, p. 341–370, sep 1985. [Online]. Available: https://doi.org/10.1145/5505.5508

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store." in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 205–220. [Online]. Available: https://doi.org/10.1145/1294261.1294281

[15] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory." in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, R. Mahajan and I. Stoica, Eds. USENIX Association, 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[16] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No Compromises: Distributed Transactions with Consistency, Availability, and Performance." in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 54–70. [Online]. Available: https://doi.org/10.1145/2815400.2815425

[17] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab." in *Proceedings of the USENIX Annual Technical Conference (ATC)*, jul 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[18] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[19] A. Ganesan, R. Alagappan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Strong and Efficient Consistency with Consistency-Aware Durability." in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 323–337. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/ganesan

[20] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale." in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds. ACM, 2016, pp. 202–215. [Online]. Available: https://doi.org/10.1145/2934872.2934908

[21] HPC-AI Advisory Council Cluster Center, "HPC-AI Advisory Council Cluster Center." [Online]. Available: https://www.hpcadvisorycouncil.com/

[22] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model." in *30th International Symposium on Distributed Computing, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, ser. Lecture Notes in Computer Science, C. Gavoille and D. Ilcinkas, Eds., vol. 9888. Springer, 2016, pp. 313–327. [Online]. Available: https://doi.org/10.1007/978-3-662-53426-7_23

[23] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 121–136. [Online]. Available: https://doi.org/10.1145/3132747.3132764

[24] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design Guidelines for High Performance RDMA Systems." in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 437–450. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[25] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be General and Fast." in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, J. R. Lorch and M. Yu, Eds. USENIX Association, 2019, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/kalia

[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "eRPC on GitHub." https://github.com/erpc-io/eRPC, 2019.

[27] A. Katsarakis, V. Gavrielatos, M. R. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol." in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 201–217. [Online]. Available: https://doi.org/10.1145/3373376.3378496

[28] D. Kim, A. S. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems." in *Proceedings of*

the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, S. Gorinsky and J. Tapolcai, Eds. ACM, 2018, pp. 297–312. [Online]. Available: https://doi.org/10.1145/3230543.3230572

[29] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostic, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism." in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 756–771. [Online]. Available: https://doi.org/10.1145/3477132.3483565

[30] A. Kokolis, A. Psistakis, B. Reidys, J. Huang, and J. Torrellas, "Distributed Data Persistency." in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 71–85. [Online]. Available: https://doi.org/10.1145/3466752.3480060

[31] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System." *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[32] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[33] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC." in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 137–152. [Online]. Available: https://doi.org/10.1145/3132747.3132756

[34] C. Lin, V. Nagarajan, and R. Gupta, "Fence Scoping." in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, T. Damkroger and J. J. Dongarra, Eds. IEEE Computer Society, 2014, pp. 105–116. [Online]. Available: https://doi.org/10.1109/SC.2014.14

[35] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading Distributed Applications onto SmartNICs using iPipe." in *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, J. Wu and W. Hall, Eds. ACM, 2019, pp. 318–333. [Online]. Available: https://doi.org/10.1145/3341302.3342079

[36] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache." in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 795–809. [Online]. Available: https://doi.org/10.1145/3037697.3037731

[37] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 401–416. [Online]. Available: https://doi.org/10.1145/2043556.2043593

[38] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "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 2017, Boston, MA, USA, March 27-29, 2017*, A. Akella and J. Howell, Eds. USENIX Association, 2017, pp. 453–468. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

[39] Mellanox, "ConnectX®-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual." https://network.nvidia.com/pdf/user_manuals/ConnectX-3_VPI_Single_and_Dual_QSFP_Port_Adapter_Card_User_Manual.pdf, 2013.

[40] Mellanox, "NVIDIA® Mellanox® BlueField® Data Processing Unit (DPU)." https://network.nvidia.com/files/doc-2020/pb-bluefield-dpu.pdf, 2020.

[41] Microsoft, "Microsoft Catapult: Transforming Cloud Computing by Augmenting CPUs with an Interconnected and Configurable Compute Layer Composed of Programmable Silicon." https://www.microsoft.com/en-us/research/project/project-catapult/, 2018.

[42] Y. Mu, K. Yao, Y. Li, Z. Li, T. Sun, L. Lu, J. He, and M. Huang, "SOSP: A SmartNIC-Based Offloading Framework for Cloud Storage Pooling." in *2022 9th International Conference on Wireless Communication and Sensor Networks (ICWCSN)*, ser. icWCSN 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 25–28. [Online]. Available: https://doi.org/10.1145/3514105.3514110

[43] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe Performance for End Host Networking." in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, S. Gorinsky and J. Tapolcai, Eds. ACM, 2018, pp. 327–341. [Online]. Available: https://doi.org/10.1145/3230543.3230560

[44] NVIDIA, "NVIDIA BlueField 2 DPU." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2021.

[45] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud." in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 29–41. [Online]. Available: https://doi.org/10.1145/2043556.2043560

[46] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency." in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 265–276. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853222

[47] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A Programming System for NIC-Accelerated Network Applications." in *13th USENIX Symposium on Operating Systems*, ser. OSDI'18. USA: USENIX Association, 2018, p. 663–679.

[48] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services." in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 13–24. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853195

[49] W. Reda, M. Canini, D. Kostic, and S. Peter, "RDMA is Turing complete, we just did not know it yet!." in *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, A. Phanishayee and V. Sekar, Eds. USENIX Association, 2022, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/reda

[50] J. Ren, "Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB." https://github.com/basicthinker/YCSB-C, 2014.

[51] A. Ryser, A. Lerner, A. Forencich, and P. Cudré-Mauroux, "D-RDMA: Bringing Zero-Copy RDMA to Database Systems." in *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. [Online]. Available: https://www.cidrdb.org/cidr2022/papers/p77-ryser.pdf

[52] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: SmartNIC-Accelerated Distributed Transactions." in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 740–755. [Online]. Available: https://doi.org/10.1145/3477132.3483555

[53] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. M. Khan, "PMNet: In-Network Data Persistence." in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 804–817. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00068

[54] H. Seyedroudbari, S. Vanavasam, and A. Daglis, "TURBO: SmartNIC-enabled Dynamic Load Balancing of μs-scale RPCs." in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 1045–1058. [Online]. Available: https://doi.org/10.1109/HPCA56546.2023.10071135

[55] S. Sun, R. Zhang, M. Yan, and J. Wu, "SKV: A SmartNIC-Offloaded Distributed Key-Value Store." in *IEEE International Conference on Cluster Computing, CLUSTER 2022, Heidelberg, Germany, September 5-8, 2022*. IEEE, 2022, pp. 1–11. [Online]. Available: https://doi.org/10.1109/CLUSTER51413.2022.00016

[56] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers." in *ASPLOS '20: Architectural Support for Programming Languages and Operating*

Systems, Lausanne, Switzerland, March 16-20, 2020, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 117–131. [Online]. Available: https://doi.org/10.1145/3373376.3378528

[57] S. Tsai and Y. Zhang, "LITE Kernel RDMA Support for Datacenter Applications." in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* ACM, 2017, pp. 306–324. [Online]. Available: https://doi.org/10.1145/3132747.3132762

[58] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Building Blocks for Persistent Memory." *VLDB J.*, vol. 29, no. 6, pp. 1223–1241, 2020. [Online]. Available: https://doi.org/10.1007/s00778-020-00622-9

[59] P. Viotti and M. Vukolic, "Consistency in Non-Transactional Distributed Storage Systems." *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, 2016. [Online]. Available: https://doi.org/10.1145/2926965

[60] W. Vogels, "Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs Between Consistency and Availability." *Queue*, vol. 6, no. 6, p. 14–19, oct 2008. [Online]. Available: https://doi.org/10.1145/1466443.1466448

[61] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and Modeling Non-Volatile Memory Systems." in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020.* IEEE, 2020, pp. 496–508. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00049

[62] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast In-Memory Transaction Processing Using RDMA and HTM." in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 87–104. [Online]. Available: https://doi.org/10.1145/2815400.2815419

[63] K. Wu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Towards an Unwritten Contract of Intel Optane SSD." in *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*, D. Peek and G. Yadgar, Eds. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/hotstorage19/presentation/wu-kan

[64] Y. Yuan, J. Huang, Y. Sun, T. Wang, J. Nelson, D. R. K. Ports, Y. Wang, R. Wang, C. Tai, and N. S. Kim, "RAMBDA: RDMA-driven Acceleration Framework for Memory-Intensive μs-scale Datacenter Applications." in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023.* IEEE, 2023, pp. 499–515. [Online]. Available: https://doi.org/10.1109/HPCA56546.2023.10071127

[65] R. Zambre, M. Grodowitz, A. Chandramowlishwaran, and P. Shamis, "Breaking Band: A Breakdown of High-performance Communication." *CoRR*, vol. abs/2002.02563, 2020. [Online]. Available: https://arxiv.org/abs/2002.02563