Concordia: Distributed Shared Memory with In-Network Cache Coherence

Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/fast21/presentation/wang

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Concordia: Distributed Shared Memory with In-Network Cache Coherence

Qing Wang, Youyou Lu[*], Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu[*]

Tsinghua University

Abstract

Distributed shared memory (DSM) is experiencing a resurgence with emerging fast network stacks. Caching, which is still needed for reducing frequent remote access and balancing load, can incur high coherence overhead. In this paper, we propose CONCORDIA, a DSM with fast in-network cache coherence backed by programmable switches. At the core of CONCORDIA is FLOWCC, a hybrid cache coherence protocol, enabled by a collaborative effort from switches and servers. Moreover, to overcome limitations of programmable switches, we also introduce two techniques: (*i*) an ownership migration mechanism to address the problem of limited memory capacity on switches and (*ii*) idempotent operations to handle packet loss in the case that switches are stateful. To demonstrate CONCORDIA's practical benefits, we build a distributed key-value store and a distributed graph engine on it, and port a distributed transaction processing system to it. Evaluation shows that CONCORDIA obtains up to 4.2×, 2.3× and 2× speedup over state-of-the-art DSMs on key-value store, graph engine and transaction processing workloads, respectively.

1 Introduction

Distributed Shared Memory (DSM) enjoyed a short heyday (circa early 1990s) by offering a unified global memory abstraction. Yet, it later failed to entertain a greater audience due to the unsatisfying performance atop the low-speed network [28]. Recent advancement in high-performance network technologies prompts a new look into DSM. With optimizations across the network stack (e.g., RDMA), the bandwidth can now surge to 100Gbps or even 200Gbps [1], and the latency drops to less than 2μs [14]. Researchers take this opportunity to systematically rethink the DSM design and have achieved a string of successes in both academia and industry [4, 20, 47, 51, 61]. For example, Microsoft has developed FaRM [4], a fast RDMA-based DSM; on top of FaRM, engineers have built key-value stores [25], distributed transaction engines [26, 59], and a graph database called A1 [19] (A1 is used by Microsoft's Bing search engine).

But there is still at least one more hurdle to cross: cache coherence. Caching is still important to DSM for obtaining competitive performance (e.g., local memory has 2-4× higher throughput and 12× lower latency against RDMA [60]) and balancing load (e.g., caching the hot data to multiple servers [27]). Yet, distributed caching always comes with coherence, which is notorious for its complexity and overhead[1]. Specifically, coherence incurs excessive communication for coordination, such as tracking cache copies' distribution, invalidation and serializing conflicting requests (i.e., requests targeting the same cache block), thereby severely impacting the overall throughput. For example, even in the RDMA-based ccNUMA [27], introducing a slight dose of cache coherence by increasing the write ratio from 0 to 5% can reduce the performance by 50%.

The advent of programmable switches has changed the landscape of various classic system architectures [23, 33, 34, 38, 41, 42, 56, 57, 65, 67, 68]. Here, we argue that leveraging a customizable switch to reduce the overhead of cache coherence in DSM is promising. First, since a switch is the centralized hub of inter-server communication, reconfiguring it to handle cache coherence can significantly reduce coordination between servers. Second, the latest switches can usually process several billion packets per second, enabling them to quickly handle coherence requests. Third, the switch owns an on-chip memory, which allows storing cache block metadata in the switch. Moreover, the on-chip memory can support atomic read-modify-write operations [33], thereby easing the effort to synchronize conflicting coherence requests.

However, simply shoehorning all cache coherence logic into switches can be impractical. First, cache coherence protocols are too complicated for programmable switches [13, 35] which only have limited expressive powers due to their restricted programming model and demanding timing requirements [49]. Second, the on-chip memory is usually small (e.g., 10-20 MB), making it unlikely to accommodate the metadata of the entire cache set. Third, failure handling can be tricky. Cache coherence protocol generally uses a state machine to perform the state transitions. Hence, if deployed on switches, a common fault, such as a packet loss, could lead a switch into an erroneous state (e.g., deadlock by repeated locking due to retransmission).

In this paper, we present the CONCORDIA, a high-performance rack-scale DSM with in-network cache coherence. The core of CONCORDIA is FLOWCC (in-Flow Cache Coherence), a cache coherence protocol that is jointly supported by switches and servers. In FLOWCC, switches, as the data plane, serialize conflicting coherence requests and

[*]Jiwu Shu and Youyou Lu are the corresponding authors.
{shujw, luyouyou}@tsinghua.edu.cn

[1]"There are only two hard things in Computer Science: cache invalidation and naming things."—Phil Karlton

multicast them to the destinations, reducing coordination between servers. Servers, on the other hand, act as the control plane that performs state transitions and sends corresponding updates to switches. Specifically, we utilize the on-chip memory to store metadata of cache blocks and implement reader-writer locks for concurrency control.

CONCORDIA also designs an ownership migration mechanism to manage the metadata of cache blocks. The mechanism moves the ownership of cache blocks between switches and servers dynamically according to the coherence traffic, namely only keeping the hot ones in the switches. To handle packet loss, we make all operations both in servers and switches idempotent, guaranteeing exactly-once semantics.

We implement CONCORDIA with Barefoot Tofino switches and commodity servers and evaluate it with three real datacenter applications: distributed key-value storage, distributed graph computation and distributed transaction processing. Experimental results show that CONCORDIA obtains up to 4.2×, 2.3× and 2× speedup on key-value store, graph engine and transaction processing, respectively, over two state-of-the-art RDMA-based DSMs, Grappa [51] and GAM [20].

To sum up, we make the following contributions:

- We propose CONCORDIA, a high-performance rack-scale DSM that incorporates an in-network cache coherence protocol, namely FLOWCC.
- We design an ownership migration mechanism and idempotent operations to overcome the restriction of current programmable switches.
- Evaluation shows that CONCORDIA gains significant performance improvement against two state-of-the-art DSMs in various applications.

2 Background

In this section, we provide background on cache coherence protocols and programmable switches.

2.1 Cache Coherence Protocols

Cache coherence protocols are studied extensively in both the systems and architecture communities [50, 53]. We briefly describe the two main protocol types used in DSMs below.

Directory-based protocols keep track of servers that hold cache copies. Each data block has a home node[2] that keeps the states and locations of cache copies. The home node updates and notifies the state of the data block to all cache copies, and serializes conflicting cache coherence requests. The limitation is that the home nodes induce extra round trips, and the home nodes for hot data are under heavy load.

Snooping protocols do not keep track of cache copies. Instead, they broadcast cache coherence requests to all the servers. The limitation is that the broadcast can easily overwhelm the network, and wastes the CPU cycles of servers that do not contain the requested cache block. In addition, since ordered and reliable broadcast systems are equivalent to consensus [24], it is intractable to coordinate conflicting cache coherence requests in snooping protocols.

[2]In this paper, we use "server" and "node" interchangeably.

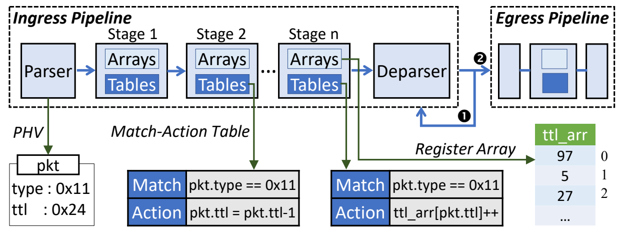

Figure 1: Pipelines in Programmable Switches. In this figure, for a packet whose `type` field is `0x11`, the switch reduces its `ttl` (i.e., time to live) field via the first match-action table, and then updates the register array `ttl_arr` via the second table. The `k`th item in `ttl_arr` records the number of packets whose `ttl` field is `k`.

2.2 Programmable Switch

Emerging programmable switches like Barefoot Tofino [2] provide programmable capacity. Such a switch follows reconfigurable match table (RMT) architecture [18] and usually has multiple ingress and egress pipelines. Each pipeline contains multiple stages, and packets are processed by these stages in sequential order, as shown in Figure 1.

Developers can program three components for switches: the parser, register arrays and match-action tables. The parser defines packet formats. A register array is a collection of memory items (e.g., `tll_arr` in the Figure 1); we can read, write, and conditionally update these items via index numbers (i.e., positions). A match-action table specifies (*i*) a *match key* from a set of packet fields (e.g., in the Figure 1, the `type` field is the match key of both tables), and (*ii*) a set of actions, each of which consists of instructions about modifying packet fields and register arrays. A register array or match-action table belongs to only one stage of a certain pipeline, which can be specified by developers.

When a packet arrives at an ingress port, the parser analyzes it and generates a packet header vector (PHV), which is a set of header fields (e.g., UDP port). The PHV is then passed to match-action tables in the ingress pipeline in a stage-by-stage manner. If the specific fields of the PHV match an entry in a match-action table, the corresponding action is executed. Before leaving the ingress pipeline, the packet is reassembled by the deparser. Then there are two cases (realized by setting a metadata field): ❶ the packet is *resubmitted*, i.e., it re-enters the ingress pipeline. ❷ the packet is switched to the egress pipeline, experiencing processing similar to the ingress pipeline, and it finally is emitted via an egress port.

We summarize two properties of an RMT pipeline, which can simplify the design of stateful protocols in switches:

P1. Atomicity Property. Due to the pipelined architecture, only one packet is processed in a stage at any time. In other words, operations for multiple register arrays in the same stage are atomic.

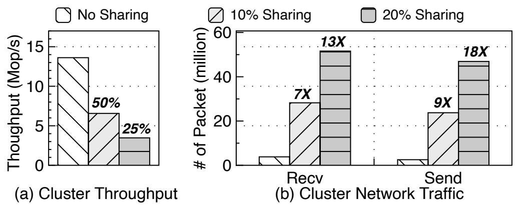

Figure 2: Impact of Cache Coherence.

P2. Ordering Property. Suppose there are two packets A, B, and they are being processed in stage S_A and S_B, respectively. If $S_A < S_B$ in time t (e.g., A in stage 1 and B in stage 2), $S_A < S_B$ holds at any time after t within this pipeline.

3 Motivation

This section revisits cache coherence under fast network environments via an experiment and discusses challenges in designing an in-network cache coherence protocol.

3.1 Revisit Cache Coherence with Fast Network

To understand the performance impact of cache coherence, we use a micro benchmark to evaluate GAM [20], a state-of-the-art RDMA-based DSM backed by a directory-based protocol. In this benchmark, each node in the 8-node cluster launches four threads to issue 8-byte write/read operations to global memory with a write ratio of 50%. Here, we define the sharing ratio as the percentage of operations that access shared data. By varying the ratio from 0 (i.e., no sharing) to 20%, we can see, in Figure 2(a), that the throughput degrades by 75%. Further, we collect the packets received and sent by all nodes (Figure 2(b)). The number of packets across network increases dramatically (up to 18×) when more data is shared because of expensive distributed communication in cache coherence protocols. From the benchmark, we conclude that *even with fast networks, existing cache coherence protocols dramatically limit system performance*.

3.2 Challenges

There are three challenges to design a fast cache coherence protocol using programmable switches:

• *The mismatch between the complexity of cache coherence protocols and the restricted expressive power of programmable switches*. Existing cache coherence protocols are intricate, because of complex state transitions in the face of concurrent and asynchronous requests. However, the expressive power of programmable switches is limited: all procedures must be represented as match-action tables. Moreover, the processing pipeline must meet the hardware resource and timing requirements of switch ASICs [34]. For example, tables with dependencies must be placed in different stages.

• *Limited switch memory capacity*. Current programmable switches have limited on-chip memory capacity (10-20MB) [37]. Furthermore, we need to reserve some memory to serve normal network protocols. Thus, switches are unable to manage the coherence of all cache blocks.

• *Packet loss*. Switch buffer overflow can cause packet loss. Existing in-network systems such as NetCache [34] and NetChain [33] rely on client-side retries to address this problem. However, it is hard to guarantee exactly-once semantics by simply retransmitting lost packets in DSMs, considering that a cache coherence request always involves multiple round trips and packets.

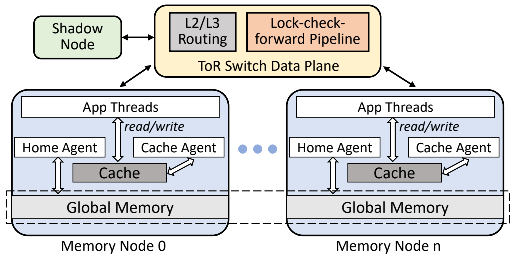

Figure 3: CONCORDIA Overview. Global memory from all memory nodes constitutes a logically unified address space (the dashed box).

4 CONCORDIA Overview

CONCORDIA is a rack-scale DSM that leverages programmable switches to accelerate cache coherence. Figure 3 shows its overview, which consists of a set of memory nodes, a top-of-rack (ToR) switch, and a shadow node.

Memory nodes. Memory nodes run distributed applications and provide memory for them. Each memory node divides its DRAM into two parts: a global memory and a private local write-back cache. The global memory from all memory nodes constitutes a logically unified 64-bit address space:

node id: 16-bit	offset: 48-bit

; each data block has a constant *home node*, which is specified by the *node id* field. The local cache is organized in *cache blocks*, which are the unit of data transfer between the local cache and global memory. A cache block is uniquely identified by its *tag* (e.g., the tag of a 4KB cache block is the highest 52 bits of its address). There are three components in a memory node: (*i*) *application threads* execute application logic and access global memory via linearizable write/read interfaces, which interact with the local cache; (*ii*) the *home agent* manages part of the global memory space within its node; (*iii*) the *cache agent* performs invalidation and data transfers for data that is cached on its memory node.

Switch. In addition to routing normal packets using standard L2/L3 protocols, the switch is responsible for executing part of the cache coherence protocol (e.g., serializing and multicasting requests) via the lock-check-forward (LCF) pipeline.

Shadow node. The shadow node helps migrate the ownership of cache blocks between the switch and home agents by recording coherence traffic of cache blocks.

As in other coherence protocols, each cache block may be in one of three states, depending on how it is cached and whether it is shared: *Unshared* (no node has it in the local cache), *Shared* (some nodes share it with read permission), and *Modified* (one node caches an exclusive copy with write permission). This state is reflected in the cache block's *global status*. The *copyset* is the set of nodes that hold the corre-

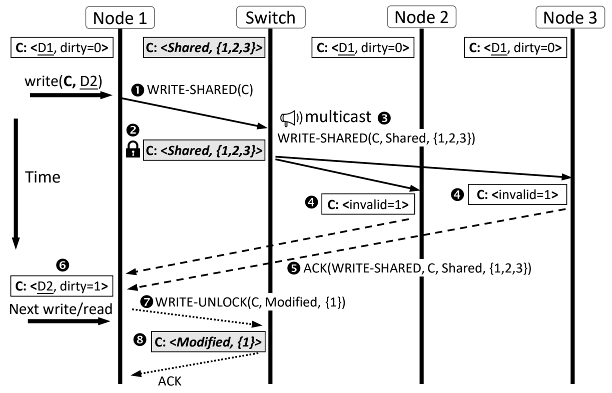

Figure 4: An Example of FLOWCC. An application thread (i.e., requester) in node 1 writes cache block `C` with data `D2`.

sponding cache block. We denote the global status and copyset of a cache block as its *global metadata*. *Owners* manage the coherence for a cache block, and store its global metadata. Ownership (and global metadata) can be migrated between the home node's home agent and the switch, depending on how actively the cache block is shared.

4.1 Key Ideas

1) Separating data and control planes. In CONCORDIA, the switch only does what it is proficient at, i.e., routing packets as the data plane; it multicasts cache coherence requests and serializes conflicting ones via in-network locks (locking can be regarded as controlling route paths of conflicting requests). In contrast, servers, as control planes, perform state transitions and send corresponding updates to the switch. Such a separation simplifies the data plane design of the switch to overcome its restricted expressive power (§5.1).

2) Unifying switch- and server-based coherence processing. Due to the limited on-chip memory capacity of the switch, CONCORDIA lets the switch only manage the coherence of hot data, and resorts to servers for processing the cold data. According to the coherence traffic that a cache block induces, CONCORDIA dynamically migrates its ownership between servers and the switch (§5.2).

3) Minimizing the number of modifications to switch state. For every operation that modifies switch state, i.e., the values stored in the switch's register arrays, we must deal with the corresponding case of packet loss, at the cost of consuming precious hardware resources of the switch and complicating the system design. Thus, we strive to minimize the number of switch state modifications. Specifically, in each coherence event, i.e., the process of executing a cache coherence request, the switch only modifies its state twice: ① acquiring locks; ② releasing locks and installing new metadata of the targeted cache block. We make the two modifications idempotent, to handle packet loss (§5.3).

4.2 Example

We illustrate the overall operation of FLOWCC protocol with an example, as shown in Figure 4.

An application thread (i.e., requester) in node 1 issues a write to cache block `C`, whose ownership is in the switch. Since `C` is already cached by node 1 and is *not* dirty, the requester generates a cache coherence request (i.e., WRITE-SHARED) with the tag of `C`, and sends it to the switch (❶).

After receiving the request, the switch acquires the write lock for cache block `C`, to prevent conflicting cache coherence events (❷). Then, it multicasts the request to cache agents in node 2 and 3, according to the value of the copyset (❸). Of note, the multicast requests contain the global metadata of `C` (i.e., global status *Shared* and the copyset).

Upon receiving multicast requests, the cache agents in node 2 and 3 invalidate their local cache block copies by marking them as invalid (❹). Then they send ACKs, which contain the global metadata of cache block `C`, to the requester (❺).

The requester waits for ACKs from the other cache agents that are indicated in the copyset of ACKs (i.e., $\{2,3\}$). Once all ACKs are received, the requester installs new data into its local cache and marks it as dirty (❻). At this time, the cache block `C` is in a coherent state in CONCORDIA. Finally, the requester sends an asynchronous unlock request to the switch (❼), to allow concurrent or subsequent coherence requests targeting the same cache block to make progress. This unlock request contains new global metadata of the cache block. The switch releases the corresponding lock, and uses information in the unlock request to install new global metadata (❽).

Overall, this cache coherence request is processed in *only a single round-trip* (compared with directory-based protocols, note that unlock is asynchronous) with *much less network traffic* (compared with snoop protocols, note that requests are only multicast to memory nodes that hold data copies).

The example above can experience a host of undesirable situations, for which we elaborate our solutions in §5. These situations include: (*i*) the lock is occupied; (*ii*) the request is invalid when entering the switch's pipeline; (*iii*) the ownership is not in the switch; (*iv*) a network packet is dropped.

5 CONCORDIA Design In Depth

In this section, we first describe FLOWCC protocol (§5.1). Next, we detail how CONCORDIA migrates ownership (§5.2) and handles packet loss (§5.3). Finally, we discuss practical issues of CONCORDIA (§5.4).

5.1 FLOWCC Protocol

At the core of CONCORDIA is FLOWCC, a write-invalidate protocol, which divides coherence responsibility between the switch and servers. The switch serializes conflicting requests and multicasts them to correct home/cache agents via the lock-check-forward (LCF) pipeline. Servers perform state transitions and update the switch data plane.

5.1.1 Protocol Details

In FLOWCC, a cache coherence event is handled collaboratively by switches and servers in the following four phases:

Field	Meaning
type	type of the cache coherence request
tag	tag of the requested cache block
node_id	node id of the requester
global_status	global status of the requested cache block
copyset	the set of nodes that hold the cache block
is_data_provider	whether the receiver is the data provider
is_in_switch	whether the switch manages the coherence

Table 1: Packet Format.

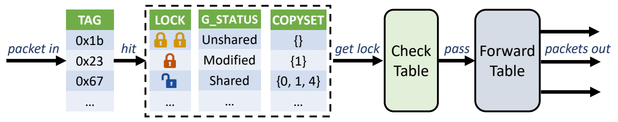

Figure 5: Lock-check-forward Pipeline. The three arrays in the dashed box belong to the same stage; the orange lock is a read lock, and the red one is a write lock. The switch just finished the lock phase of two READ-MISS requests to the cache block with `0x1b` tag.

➤ *Request Generation*

Application threads (i.e., requesters) access global memory by issuing write/read operations to the local cache, and generate *cache coherence requests* in case of cache miss and cache eviction. Table 1 shows the packet format of requests. There are five types of cache coherence requests (i.e., *type* field): ①② WRITE-MISS/READ-MISS for cache miss, ③WRITE-SHARED when issuing a write operation to a clean cache block, ④⑤EVICT-MODIFIED/EVICT-SHARED when evicting a dirty/clean cache block. The *tag* and *node_id* fields identify the requested cache block and the node of the requester, respectively. The last four fields are filled by the switch.

➤ *Lock-check-forward Pipeline*

In the LCF pipeline, we store a reader-writer lock and global metadata for each cache block. By leveraging this state, the LCF pipeline serializes conflicting requests and multicasts them to destinations.

On-chip state storage. Figure 5 shows the LCF pipeline, which consists of four register arrays and two match-action tables. The *TAG* array is a hash table that stores the *tag* of cache blocks by using 64-bit slots. The *LOCK* array contains 16-bit reader-writer locks; for each lock, the most significant bit indicates a writer, and the other 15 bits count the number of readers (i.e., 2^{15} concurrent readers at most). Lock and unlock operations are supported by conditional update of the register array. The *G-STATUS* array records the global status of cache blocks with 8-bit slots. The *COPYSET* array maintains cache block copysets by using a bitmap structure. The last three register arrays are placed in the same stage, so they can be manipulated together in an atomic manner (P1 in §2.2). The *check table* verifies the validity of cache coherence requests. The *forward table* routes requests to the correct cache/home agents.

Request processing. The LCF pipeline processes a request in three steps: (*i*) lock the requested cache block for concurrency control. (*ii*) check the request's validity. (*iii*) forward the request to its final destinations.

When a request enters the LCF pipeline, the switch first hashes the *tag* field of the request into an index number, and then uses the index number to search the *TAG* array for the *tag* (i.e., check if *TAG*[hash(*tag*)] equals *tag*). On failure, the switch forwards the request to its home agent [3] and sets the *is_in_switch* field to false. After finding the *tag*, the switch tries to acquire the read lock if the request is READ-MISS; otherwise, acquires the write lock. Note that we protect EVICT-SHARED with write locks, since two concurrent EVICT-SHARED requests to the same cache block may cause different state transitions (detailed later, see Figure 6). In parallel, the corresponding global metadata (i.e., values in *G-STATUS* and *COPYSET* array) is filled into the request. The switch returns a failed ACK to the requester when failing to acquire the lock.

Match Key pkt.type	Action
READ-MISS	check_succ ← pkt.node_id $\notin$ pkt.copyset
WRITE-MISS	
WRITE-SHARED	check_succ ← pkt.node_id $\in$ pkt.copyset $\wedge$ pkt.global_status == *Shared*
EVICT-SHARED	
EVICT-MODIFIED	check_succ ← pkt.node_id $\in$ pkt.copyset $\wedge$ pkt.global_status == *Modified*

Table 2: Check Table. If the `check_succ` is true (i.e., the request is valid), the switch passes the request to the forward table; otherwise, the switch releases the lock and sends a failed ACK to the requester.

Match Key (pkt.type, pkt.global_status)	Action
(READ-MISS, *Unshared*)	forward to the request's home agent
(WRITE-MISS, *Unshared*)	
(READ-MISS, *Modified*)	forward to the cache agent in pkt.copyset
(WRITE-MISS, *Modified*)	
(READ-MISS, *Shared*)	select a cache agent in pkt.copyset as the data provider, and forward to it
(WRITE-MISS, *Shared*)	multicast to cache agents in pkt.copyset, select a cache agent as the data provider
(WRITE-SHARED, *Shared*)	multicast to cache agents in pkt.copyset except pkt.node_id [4]
(EVICT-SHARED, *Shared*)	return to the requester
(EVICT-MODIFIED, *Modified*)	

Table 3: Forward Table. Requests multicast/forwarded to cache agents (except for READ-MISS requests) indicate invalidation.

Once the switch acquires the lock successfully, it passes the request to the check table (shown in Table 2), to filter out invalid requests caused by concurrent events. Let us consider an example of invalid requests: immediately after a requester issues a WRITE-SHARED request `W` for cache block `A`, the cache agent in the same node invalidates `A`; it is possible that, due to `W` being delayed by OS scheduler or network, the global status of `A` is no longer *Shared* or the requester no longer holds `A` when `W` enters the LCF pipeline. Thus, to ensure that only valid requests are forwarded to destination nodes, the switch checks the *global_status* and *copyset* in the request. If the check fails, the switch resubmits the request to release the acquired lock, and then sends a failed ACK to the requester. Compared with server-based mechanisms, such

[3] For simplicity, we denote *the home agent of a request or a cache block* as the home agent that resides in the requested cache block's home node.

[4] If copyset only contains the requester, the switch returns an ACK.

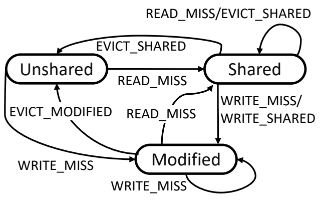

Figure 6: State Transitions of Global Status. Requesters generate global status according to this diagram, and piggyback it on unlock requests. When evicting a clean cache block (i.e., EVICT-SHARED), the global status changes from *Shared* to *Unshared* if the copyset is empty; otherwise, the global status remains unchanged.

an in-network check reduces the load of servers (checking validity of requests and sending failed ACKs).

If the request passes the check table successfully, the forward table (see Table 3) sends it to its final destinations according to the *(type, global_status)* pair. Specifically, there are three cases: (*i*) If no copy of the requested cache block exists, i.e., *global_status* is *Unshared*, the switch routes the request to its home agent for fetching data in global memory. (*ii*) If cache block copies need to be invalidated, the switch multicasts the request to the cache agents in the *copyset*. (*iii*) For an eviction request, the switch returns it to the requester directly, since the corresponding lock has been acquired.

To support efficient cache-to-cache data transfer, we adopt an in-network selection mechanism. For READ-MISS/WRITE-MISS requests, if shared nodes exist (i.e., *copyset* is not empty), the switch selects a cache block data provider among them using a load balancing strategy and sets *is_data_provider* field in the corresponding request. The current load balancing strategy is to randomly choose a data provider from shared nodes; compared with other complex strategies, the random selection does not consume any precious on-chip memory.

➤ ***Invalidation and Data Transfer***

Cache agent. Upon receiving the request, the cache agent invalidates corresponding cache block for WRITE-MISS/WRITE-SHARED. Then, the cache agent sends an ACK to the requester. If the cache agent is a data provider, it also transfers cache block data.

Home agent. Upon receiving a request, if *is_in_switch* field is *true*, the home agent replies to the requester with an ACK, which contains the corresponding cache block data from global memory. The remaining cases, i.e., the switch does not own the ownership of the requested cache block, are discussed in §5.2.

Note that all ACKs from cache/home agents carry the global metadata of the requested cache block, so as to help requesters reach coherence.

➤ ***Requester-driven Coherence Control***

The requester waits for ACKs from home/cache agents, and then performs state transitions and updates the switch data plane. Specifically, upon receiving a failed ACK, the requester retries the read/write operation. Otherwise, the requester waits for all ACKs needed for reaching coherence. Then, for READ-MISS/WRITE-MISS, the requester installs cache block data; for EVICT-MODIFIED, it writes back the cache block data to the home node. After that, the requester generates new global metadata (i.e., *copyset* and *global status*): the new copyset is (*i*) $\{id\}$ for WRITE-MISS/WRITE-SHARED or (*ii*) $cs_{old} + \{id\}$ for READ-MISS or (*iii*) $cs_{old} - \{id\}$ for eviction requests, where cs_{old} is the copyset in the ACKs and *id* is the node id of the requester. The new global status is generated according to the state transitions diagram in Figure 6. Finally, the requester sends an unlock request to the switch, with the new global metadata. The unlock request is asynchronous, enabling the requester to execute subsequent read/write operations immediately.

On receiving an unlock request, the switch releases the lock (i.e., modifies the *LOCK* array), and updates the global metadata (i.e., values in *G-STATUS* and *COPYSET* arrays). For read lock release, the new value in the *COPYSET* array is the union of the old one and the *copyset* field in the unlock request, considering maybe there are concurrent READ-MISS requests. Atomicity property (P1 in §2.2) guarantees the atomicity of update to these three arrays, since they are in the same stage.

5.1.2 Correctness

FLOWCC is based on the state transitions of classic MSI protocols. Compared with directory-based MSI protocols, it has two main changes: ① leveraging in-switch locks to serialize conflicting requests; ② leveraging the switch to multicast invalidation messages. Here, we prove FLOWCC is correct by proving the following two invariants are correct [27, 50]. We assume all messages are reliable (§5.3 describes how CONCORDIA handles packet loss).

INVARIANT 1. Exactly one writer can update a memory location at a time (*Single-Writer-Multiple-Readers Invariant*).
PROOF. Before getting the write permission of a cache block, a thread must acquire the in-switch write lock and revoke the write/read permission of other servers via invalidation; thus, only one thread can write a cache block at any time.

INVARIANT 2. If a cache block is valid, it must hold the most recent value updated by writers (*Data-Value Invariant*).
PROOF. On write requests, the switch invalidates all the copies by multicasting messages and only the requester owns the most recent data at the end; on read requests, the most recent data is populated into the requester's local cache. Thus, any operations manipulate the latest data.

5.2 Ownership Migration

To overcome limited on-chip memory capacity in the switch, CONCORDIA explicitly limits the number of cache blocks that the switch manages coherence for, using an *ownership migration mechanism*. Specifically, CONCORDIA migrates ownership between home agents and the switch dynamically: if a cache block has heavy coherence traffic, its home agent migrates its ownership to the switch. Similarly, if a cache block only induces light coherence traffic, its ownership is migrated in the opposite direction.

Home agents handle cache coherence requests for cache blocks they manage (the *is_in_switch* field in these requests is *false*). Specifically, home agents process cache coherence requests in the same way as the LCF pipeline of the switch: they store global metadata for multicast and use reader-writer locks to serialize conflicting requests.

5.2.1 Migration Requests

We design two type of requests for ownership migration.

• ADD-TO-SWITCH. When a home agent needs to migrate the ownership of a cache block to the switch, it acquires the write lock for the cache block (to prevent conflicting requests), and then sends an ADD-TO-SWITCH request to the switch. The request consists of the cache block's *tag*, *global status* and *copyset*. The switch inserts these three values into *TAG*, *G-STATUS*, and *COPYSET* array, respectively. Ordering property P2 (in §2.2) ensures other requests will see all the updates or none. The home agent releases the lock after receiving an ACK from the switch.

• REMOVE-FROM-SWITCH. When a home agent is informed by the shadow node to migrate the ownership of a cache block from the switch to itself (see §5.2.2), it sends a REMOVE-FROM-SWITCH request to the switch with the *tag* of the cache block. Upon receiving the request, the switch acquires the cache block's write lock in the *LOCK* array. On success, the switch resubmits the request. When receiving the resubmitted request, the switch clears the *tag* in the *TAG* array; then, it releases the lock and sends an ACK with up-to-date *global status* and *copyset* (i.e., global metadata) to the home agent. The home agent stores these two values locally.

5.2.2 Migration Workflow

Figure 7 shows the ownership migration of a cache block.

Home agents ➟ Switch. Home agents identify hot cache blocks managed by themselves and migrate their ownership to the switch. We divide time into continuous epochs (e.g., 10ms), and each home agent records the hotness of cache blocks in the current epoch. If a cache coherence request needs to be multicast to *n* cache agents, the hotness of corresponding cache block is incremented by *n*. At the end of an epoch, the home agent migrates the top-k (e.g., 1000) hottest cache blocks to the switch by issuing ADD-TO-SWITCH.

Switch ➟ Home agents. We introduce a *shadow node* to collect hotness statistics for cache blocks managed by the switch. The shadow node duplicates the switch's *TAG* array into its in-memory array called *SHADOW-TAG*. Cache agents record the number of invalidations for cache blocks managed by the switch, and report their statistics to the shadow node at the end of each epoch. The shadow node applies these statistics to *SHADOW-TAG*. The shadow node scans the *SHADOW-TAG* array periodically to remove the coldest cache blocks by informing home agents to issue REMOVE-FROM-SWITCH for

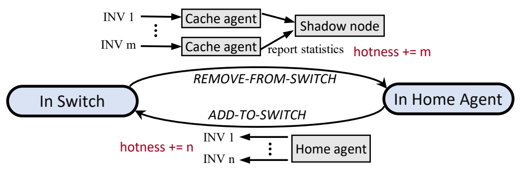

Figure 7: Ownership Migration of a Cache Block. `INV` means invalidation messages; the migration is performed by home agents.

these cache blocks.

Handling collisions. Since the switch manages the *TAG* array as a hash table (recall §5.1.1), the candidate locations of an ADD-TO-SWITCH operation may be occupied by other cache blocks (i.e., hash collisions). In such a case, the switch returns a failed ACK to the home agent that issues the ADD-TO-SWITCH; the home agent will retry the ADD-TO-SWITCH at the next epoch (if the corresponding cache block is still hot). The switch copies the ACK of every ADD-TO-SWITCH to the shadow node: if the ACK indicates success, the shadow node adds the corresponding *tag* into the *SHADOW-TAG*; otherwise, it removes the coldest cache block in the conflicting locations. By leveraging the centralized shadow node to handle collisions, CONCORDIA can ensure that though each home agent independently chooses its hottest cache blocks, the switch can manage the globally hottest cache blocks.

5.3 Packet Loss Handling

In the FLOWCC protocol, the switch is stateful due to storing locks and global metadata; thus, traditional end-to-end mechanisms, e.g., TCP retransmission, can not handle packet loss correctly. We make operations both in servers and the switch idempotent, to address this problem[5].

5.3.1 Server Idempotence

We use sequence numbers to guarantee idempotence of server operations [40]. Each requester assigns a unique requester-local sequence number for a cache coherence event; all requests (i.e., cache coherence requests and unlock requests) in the cache coherence event contain the sequence number. Sequence numbers are allocated in increasing integer order. On suspecting a lost request via timeout, the requester retransmits it. We set the timeout value of cache coherence requests to 6 RTTs (round-trip times) and unlock requests to 3 RTTs.

Each home/cache agent maintains a *LAST-EXECUTED* table, which records the largest sequence number ever received from each requester. When a home/cache agent receives a cache coherence request, it compares the request's sequence number with the corresponding value in the *LAST-EXECUTED* table. If the request's sequence number is larger, the home/cache agent executes the request, updates the *LAST-EXECUTED* table, and responds. If the request's sequence number is lower, the request is ignored. If the two are the same, the home/cache agent sends an ACK without modifying any state.

[5]For space reasons, in this subsection (§5.3), we only present how to handle packet loss in the FLOWCC protocol. We use the same mechanism for the ownership migration.

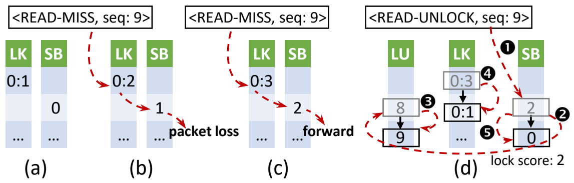

Figure 8: Idempotent Lock/Unlock Operations. The boxes on the top show packets with only the relevant details (*seq*: sequence number). `LK` is the *LOCK* array, and `SB` is the *SCOREBOARD* array, and `LU` is the *LAST-UNLOCK* array. Each 16-bit reader-writer lock in *LOCK* has the form `x:y`, where `x` is most significant bit marking the writer and `y` is the other 15 bits counting the number of readers. **(a)** Initialization state. **(b)** A requester issues a READ-MISS. The switch acquires the read lock by increasing the reader counter, and then increases the requester's lock score. Unfortunately, the READ-MISS is dropped after leaving the switch pipeline. **(c)** Upon timeout, the requester re-sends the READ-MISS with the same sequence number. After updating the *LOCK* and *SCOREBOARD* arrays, since the lock score is more than 0, the switch thinks the lock is acquired successfully. **(d)** The switch handles an unlock request.

5.3.2 Switch Idempotence

Since modifications to the *TAG*, *G-STATUS*, and *COPYSET* arrays in the switch are already idempotent, we only need to make lock and unlock operations idempotent. We introduce a *SCOREBOARD* array and a *LAST-UNLOCK* array to solve this problem, at the cost of slight on-chip memory usage.

The *SCOREBOARD* array is in the LCF pipeline and is placed before the check table to **make lock operations idempotent**. For each requester, it records a *lock score*, which refers to the number of successful lock acquisitions. When the switch processes a cache coherence request and manages to acquire the lock, the requester's lock score is incremented. The switch passes the request to the subsequent check table *only if the lock score is greater than 0*. For read lock operations, the lock score may be greater than 1 due to retransmitted requests; Figures 8(a)-(c) show an example of such a case.

The *LAST-UNLOCK* array is placed before the LCF pipeline to **make unlock operations idempotent**. For each requester, it records the largest sequence number of executed unlock requests, to avoid repeated execution of the same unlock request. When receiving an unlock request, the switch reads the requester's lock score (❶ in Figure 8(d)), and then resubmits the request with the lock score (❷). If the sequence number in the resubmitted request is larger than the value in the *LAST-UNLOCK*, the switch updates the *LAST-UNLOCK* array (❸) and executes the unlock operation (❹). For a read unlock operation, the switch subtracts lock score from the reader counter field of the reader-writer lock. Finally, the requester's value in the *SCOREBOARD* is reset (❺).

Using only one lock score for a requester disables asynchronous unlock requests, since we cannot allow requests in two cache coherence events to manipulate the same lock score concurrently. Thus, to enable asynchronous unlock requests, each requester is associated with two lock scores in *SCOREBOARD* array: one for requests with odd sequence numbers, the other for even ones. Before issuing an unlock request, a requester must ensure that it has received the ACK of the unlock request in the last cache coherence event.

5.4 Practical Issues

Scalability. We focus on rack-scale DSMs in this paper, following the growing tread towards rack-scale computers, which have the potential as building blocks for datacenters (e.g., Microsoft Rack-scale computers [9], Facebook Open-Rack [3], Intel RSD [7]). By packing many servers into the same rack, a rack-scale computer can provide extremely high network bandwidth and low communication latency, to efficiently support various data-intensive applications (e.g., parameter servers [48]). CONCORDIA abstracts a rack into one giant machine with massive cache-coherent shared memory, easing the programming on rack-scale computers.

Within a rack, CONCORDIA is capable of scaling well. In our FLOWCC protocol, the most difficult tasks to scale (i.e., concurrency control and multicast) are offloaded to the switch; during a cache coherence event, involved home/cache agents only process constant-time tasks (independent of cluster size). The switch can process several billion packets per second, which is enough to handle coherence traffic within a rack.

If CONCORDIA spans multiple racks, each ToR switch will execute the LCF pipeline for home nodes that reside in its rack. Yet, there are several challenges that we must address. ① It is impractical to use a bitmap to encode the *copyset* as we do now, considering the number of servers in large-scale clusters; we need to design a more compact *copyset* format like coarse vectors [31]. ② To avoid cache invalidation storms in a large scale, a weaker memory consistency level (e.g., acquire-release consistency [61]) may be needed. A full exploration is our future work.

Crash safety. We handle failures of different components.

• *Switch*. If a switch fails, operators can replace it with a backup switch [34], but lost global metadata needs to be restored. Here, we mark the set of cache blocks managed by the crashed switch as S. After all application threads abort their ongoing write/read operations, the system enters into a recovery process: ① Every cache agent gathers the local states (i.e., dirty bit) of cache blocks that are in both S and its cache, and then sends them to corresponding home agents. ② By analyzing replies, home agents reconstruct the global metadata of S; now, the ownership of S is transferred to home agents safely. ③ Finally, the shadow node clears its *SHADOW–TAG*. Leveraging the information stored in the shadow node may accelerate recovery, and we leave it for future work.

• *Memory nodes*. We rely on applications atop CONCORDIA to mask failures of memory nodes, instead of implementing an internal fault-tolerance mechanism. Take the three applications in our evaluation (§7) as examples: (*i*) In-memory key–value stores are typically used to cache frequently accessed data of back-end databases [52]. Hence, we recover lost data from back-end databases. (*ii*) Transaction processing systems can record transaction logs to remote servers [36,63], to tolerate failures of memory nodes. (*iii*) For a graph computation engine, the most common way to achieve fault-tolerance is checkpointing [29,51].

When a memory node fails but does not release locks, conflicting requests from other nodes cannot proceed. We adopt a conservative "stop-the-world" mechanism to address this problem: once notified that there is a failed memory node, all application threads in CONCORDIA drain their ongoing write/read operations, then halt. Next, all the locks in CONCORDIA are released safely. After that, the system continues to run. Using more flexible mechanisms, e.g., leasing [30], will consume extra on-chip memory and complicate the design of the switch data plane; we leave it as our future work.

• *Shadow node*. When the serving shadow node crashes, we designate a new shadow node among backups with the help of ZooKeeper [32]. The new shadow node reconstructs the *SHADOW-TAG* by reading the *TAG* array of the switch.

6 Implementation

We implement a prototype of CONCORDIA in approximately 7600 lines of C++ and 1500 lines of P4 [17]. Like TreadMarks [15] and GAM [20], CONCORDIA is a user-space DSM system. Applications are linked to the CONCORDIA library and call the read/write interface. In each memory node, the cache agent and home agent run on two different background threads by default, but it is easy to scale up them.

Network. We use RoCE (RDMA over Converged Ethernet) to enable high-performance network communication. All the cache block data is sent via RDMA WRITE verbs over Reliable Connected (RC) mode to avoid data copying. Other packets (e.g., cache coherence requests) use RDMA Raw Packets [10]; we fill a UDP header and use a reserved source port for these packets. The switch executes FLOWCC for these packets, and sets the UDP destination port to the queue pair number (qpn) of the targeted queue pair. Each Raw Packet queue pair registers a steering rule to intercept the packets received by the NIC whose source port equals reserved port and destination port equals its qpn.

Cache. The cache in each memory node is organized into a bucket-based hash table. When looking up a cache block, we hash its *tag* into a bucket, which contains a certain number of entries (e.g., 8). Each entry records a cache block's local metadata, including *dirty bit*, *tag*, *timestamp* and a pointer to the cache block data. We record the current time into the *timestamp* field when accessing a cache block, and employ an LRU policy for cache eviction. By default, the cache block size is 4KB, which achieves a good balance between network bandwidth and latency in the high-speed RDMA environment.

Switch data plane. The switch data plane is written in P4 and is compiled to Barefoot Tofino ASIC [2]. The *COPYSET* array is comprised of 32-bit slots, since our ToR switch owns 32 ports. We use one ingress pipeline to process the cache coherence traffic by realizing our LCF pipeline. Since the memory resources in a single stage of the ingress pipeline are limited, we scatter the values of *TAG*, *LOCK*, *G-STATUS*, and *COPYSET* arrays across 10 stages, forming set-associative structures. Thus, a cache block has multiple candidate locations in different stages: if the hash value of its tag is `k`, its tag and global metadata can be stored in the `k`th items of arrays in any stage. Our current implementation can manage 375K cache blocks (i.e., about 1.5GB cache data) in the switch, consuming about 6MB on-chip memory.

Synchronization primitive. We expose per-cache-block reader-writer locks in the FLOWCC protocol as a synchronization primitive (i.e., rwlock) to applications.

Global memory allocation. We use a two-level approach to allocate global memory for applications [60]. Application threads select a memory node and send allocation requests to its home agent. The home agent returns a huge free chunk (i.e., 32MB). Then application threads perform allocation locally in a fine-grained way, to reduce remote access.

7 Evaluation

This section presents the performance evaluation of CONCORDIA with a set of micro benchmarks and three applications. All experiments are conducted on a cluster of 8 machines, each with two 12-core Intel Xeon E5-2650 v4 2.20GHz CPUs, 128GB DRAM, and one 100Gbps Mellanox ConnectX-5 network adapter. A 3.3Tbps Barefoot Tofino switch (32 ports) connects all of the machines. All machines run the CentOS 7.4.1708 distribution and the 3.10.0 Linux kernel.

7.1 Systems in Comparison

We compare CONCORDIA with two state-of-the-art DSMs:

Grappa. Grappa [6,51] is a DSM *without* a cache for data-intensive applications. Instead of fetching data to computation, Grappa ships computation to data via delegate operations. Message transmission in Grappa is done purely in user-mode using MPI, which in turn uses RDMA verbs.

GAM. GAM [5, 20] is a recent DSM that implements a directory-based cache coherence protocol over RDMA. In addition to application threads, GAM uses two background threads to handle cache coherence events by default. We disable GAM's logging scheme for a fair comparison.

7.2 Micro Benchmarks

In micro benchmarks, we launch a four-thread process in each memory node to generate mixed read-write workloads. Each operation in workloads accesses an 8-byte object in the global memory, which is the same as in GAM [20]. The working set of these micro benchmarks is 8GB, and the cache size is 1GB. The access pattern of the workloads is controlled via three parameters: *read ratio*, *data locality*, and *sharing ratio*. The *read ratio* is the percentage of read operations. The *data locality* is the probability of an operation accessing the same cache block as the previous one. The *sharing ratio* is the percentage of operations that access data shared across all nodes, and the shared data is about 250MB. We run all

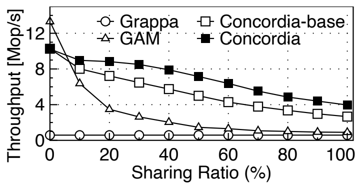

Figure 9: Performance Impact of Sharing Ratio.

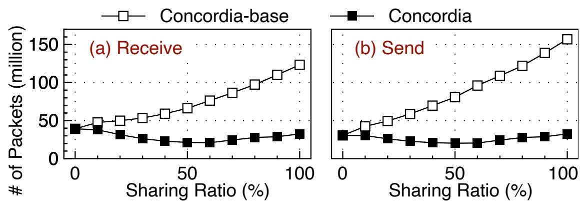

Figure 10: Communication Reduction from LCF Pipeline. The figure shows the number of packets received/sent by home agents.

the micro benchmarks on 8 nodes. By default, the read ratio is 50% and the data locality is 0%. The baseline version of CONCORDIA (i.e., CONCORDIA-base), in which servers manage ownership of all cache blocks, is also evaluated.

7.2.1 Sharing Ratio

Figure 9 presents the results with various sharing ratio. Because the operations are very small (i.e., 8 bytes read/write), which induces too many remote delegation operations and worker scheduling events in Grappa, Grappa's throughput is far lower than other systems.

When the sharing ratio is 0%, i.e., no cache coherence traffic, GAM has higher throughput than CONCORDIA and CONCORDIA-base. This is because the cache in GAM is a fully associative mapping structure, which causes fewer cache block evictions than the set-associative cache of CONCORDIA.

As the sharing ratio becomes higher, the introduced cache coherence traffic becomes severe, which causes performance degradation of all the DSMs. The throughput of GAM is reduced by 17× when the sharing ratio increases from 0% to 100%. However, CONCORDIA-base's throughput is only reduced by 3.8× even with considerable coherence traffic (i.e., 100% sharing ratio), and it outperforms GAM by 1.25× to 3.5× when shared data access exists. The reason is that CONCORDIA-base offloads home node tasks such as invalidation aggregation to the requesters, thus reducing the load on home nodes. CONCORDIA outperforms CONCORDIA-base by 1.3× to 1.48× when sharing ratio is larger than 20%. This is mainly because our in-network protocol takes home nodes off the critical path of cache coherence as well as reduces network hops and coordination, leveraging the extremely high processing power of the switch.

To quantitatively understand the effect of the LCF pipeline, we collect the number of packets sent and received by all home agents in CONCORDIA and CONCORDIA-base, as shown in Figure 10. When the sharing ratio grows, the number of packets sent and received by CONCORDIA-base's home agents increases dramatically due to increased coherence traffic. However, owing to the help of the LCF pipeline, home agents in CONCORDIA send and receive a steady number of packets, regardless of the sharing ratio. Such a result indicates that the LCF pipeline effectively mitigates load for home agents and reduces coordination between servers, which ensures CONCORDIA's performance gain over CONCORDIA-base. Of note, though CONCORDIA reduces transferred packets significantly (up to 4.8×), it improves throughput by up to 1.48× (Figure 9). This is because part of the accesses are served by the local cache (39% in case of 100% sharing ratio) and do not incur coherence traffic; these local accesses cannot benefit from the LCF pipeline.

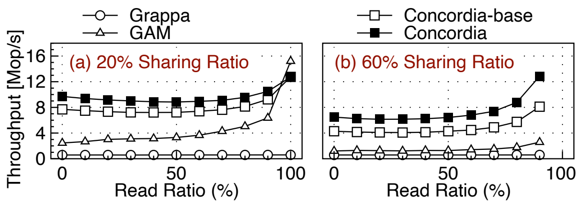

Figure 11: Performance Impact of Read Ratio.

7.2.2 Read Ratio

This experiment studies how the read ratio affects the performance. Figure 11 shows the results. When the read ratio is lower than 50%, the throughput of these four DSMs is stable regardless of read ratio, since the performance of systems except Grappa is bottlenecked by coherence traffic. CONCORDIA outperforms CONCORDIA-base by 1.22× and GAM by 3.5× to 3.9× in case of light data sharing (i.e., 20% sharing ratio) and outperforms CONCORDIA-base by 1.48× to 1.5× and GAM by 4.5× to 5.1× in case of heavy data sharing (i.e., 60% sharing ratio). CONCORDIA performs better because the FLOWCC protocol accelerates coherence processing.

The throughput of GAM, CONCORDIA-base, and CONCORDIA grows as the read ratio increases from 50% to 100%, since the coherence traffic becomes light. More specifically, ① GAM outperforms CONCORDIA in case of 100% read ratio and 20% sharing ratio. This is because no coherence traffic exists and GAM triggers fewer cache eviction events than CONCORDIA. ② With 100% read ratio and 60% sharing ratio, the throughputs of GAM and CONCORDIA/CONCORDIA-base are 16Mops and 60Mops, respectively, which we do not plot in the figure to avoid obscuring other results. This is because (*i*) there is almost no cache eviction, and (*ii*) CONCORDIA's cache structure is optimized for fast access, but GAM needs to acquire/release four locks and maintain LRU lists when accessing a cache block, which severely stalls CPU pipeline even without data race.

7.2.3 Data Locality

Figure 12 investigates how data locality affects the throughput of these systems. With higher data locality, the performance of CONCORDIA and GAM improves substantially as a result of the cache. On the contrary, since Grappa does not use a cache, it can not benefit from the locality. We do not plot results when the locality is 100% because the corresponding

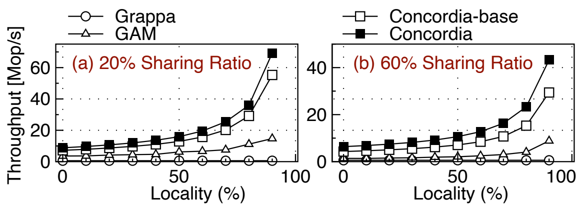

Figure 12: Performance Impact of Data Locality.

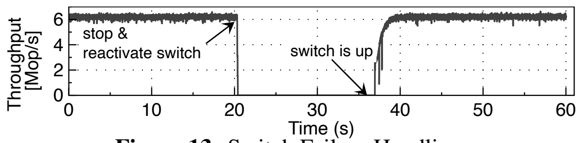

Figure 13: Switch Failure Handling.

values are too high: 23Mops for GAM and 133Mops for CONCORDIA and CONCORDIA-base, which would obscure other results. This is because CONCORDIA's cache structure is much faster, as stated in §7.2.2.

7.2.4 Switch Failure Handling

Here, we evaluate how CONCORDIA handles a switch failure. We set the sharing ratio to 60%. Figure 13 shows the total throughput over time. At time 20 s, we stop the switch by killing its daemon process, and then reactivate it; since the switch cannot route packets, the throughput immediately drops to 0. After initialization of the switch ASIC and drivers (about 16 seconds), the switch is up and can route packets. Then, CONCORDIA recovers the global metadata of cache blocks managed by the previous switch instance, consuming about 1.9 seconds; after this step, CONCORDIA can continue to run its workloads. Finally, CONCORDIA migrates the ownership of shared cache blocks to the switch (1.5 seconds), and the throughput reaches a stable peak.

7.3 Distributed Key-Value Store

We build a distributed key-value store atop CONCORDIA, which is implemented by a distributed array of buckets across nodes. A key is hashed to a bucket, and then PUT/GET operations are translated into a series of DSM calls (i.e., write, read, lock and unlock). GAM has a similar version of key-value store implementation. For Grappa, each thread maintains part of the key-value store and handles delegation requests.

We run skewed workloads using a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same as YCSB's [22]. The keyspace size is 64 million, and we use fixed 8-byte keys and 128-byte values. 95% GET workloads are read-intensive, and 50% GET workloads are write-intensive. Each node launches a four-thread process to generate workloads. The cache size is 2GB. Figure 14 shows the results with varied node counts.

For read-intensive workloads, when there are more than 2 nodes, CONCORDIA outperforms Grappa by 3.9× to 5× and GAM by 1.2× to 2.5×. Grappa has the lowest performance among the three systems, because each PUT/GET operation needs remote delegation and the nodes serving the most popu-

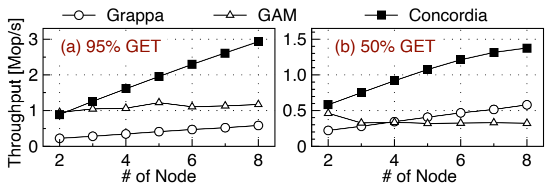

Figure 14: Throughput of Key-Value Store.

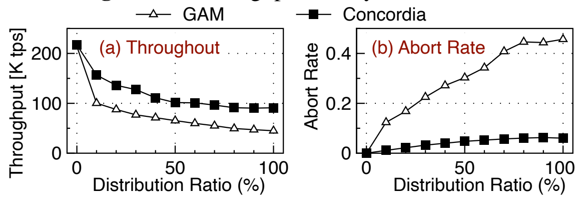

Figure 15: Performance of TPC-C Benchmark.

lar objects suffer from hot spots in the presence of skew. The cache absorbs many GET operations, and thus the throughputs of GAM and CONCORDIA are higher than that of Grappa. The workload with 5% PUT operations causes a small amount of cache coherence traffic in CONCORDIA and GAM. However, GAM is more vulnerable to coherence traffic than CONCORDIA, due to its traditional directory-based protocol design, and its throughput plateaus or even decreases somewhat when the node count becomes larger. In contrast, CONCORDIA can benefit from the cache, while minimizing coherence overhead with the help of FLOWCC protocol.

For write-intensive workloads, CONCORDIA outperforms GAM by 1.2× to 4.2× and Grappa by 2.3× to 2.6×. GAM's total throughput drops and is lower than Grappa's when adding more nodes because of heavy coherence overhead. The ownership migration in CONCORDIA lets the switch manage coherence traffic of the hottest cache blocks in skewed workloads, which guarantees CONCORDIA's scalability.

7.4 Transaction Processing

We port the transaction engine of GAM into CONCORDIA; this engine implements two-phase locking for concurrency control and non-waiting scheme for deadlock prevention. In this experiment, we use TPC-C [11] to compare the performance of CONCORDIA against GAM. All tables and indices are uniformly distributed in the global memory, following the growing trend towards shared-everything architectures [64, 66]. We launch 4 threads in each node, and each thread holds a TPC-C warehouse (i.e., 32 warehouses in total). Each thread accesses other warehouses with a probability called the *distribution ratio*. The cache size is 2GB.

Figure 15(a) shows the throughput of the TPC-C benchmark. Both systems have almost the same throughput when there is no data sharing. However, CONCORDIA outperforms GAM by 1.55× to 2× when different threads access the same warehouses. This is because CONCORDIA's FLOWCC protocol is faster than GAM's, causing lower transaction execution time and further reducing transaction aborts due to contention. Figure 15(b) plots the transaction abort ratio. The abort ratio of the two systems increases when the distribution ratio gets

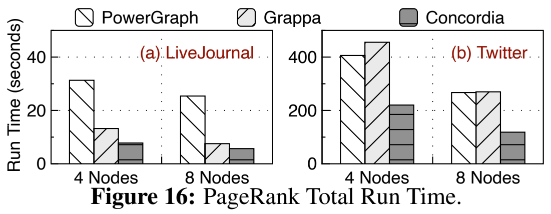

Figure 16: PageRank Total Run Time.

higher, but CONCORDIA has a much lower abort ratio.

7.5 Distributed Graph Computing

We build a distributed graph processing engine on CONCORDIA. Similar to PowerGraph [29], we store graphs using a vertex-centric representation with random graph partitioning. For a graph algorithm, each thread executes the gather, apply and scatter phases on a set of vertices.

We compare CONCORDIA's graph engine with Grappa and PowerGraph using two real datasets: LiveJournal (4M vertices, 34.7M directed edges) [8] and the latest Twitter graph (61M vertices, 14B directed edges) [12, 39] . We run PageRank [54] with CONCORDIA, PowerGraph, and Grappa under 4 and 8 node settings, and each node launches 4 threads. We use PowerGraph's default threshold criteria, which results in the same number of iterations for all systems.

Figure 16 shows the total PageRank run time. CONCORDIA outperforms Grappa by 1.3× to 2.3× and PowerGraph by 1.8× to 4.4×. This is because, CONCORDIA's network stack takes full advantage of RDMA performance, using WRITE verbs without any data copying. In addition, CONCORDIA's cache mechanism minimizes the amount of data transferred across the network.

8 Related Work

Distributed Shared Memory. In the past few decades, many DSMs and cache coherence approaches have been proposed [15, 16, 21, 44, 58, 62]. IVY [44] provides sequential consistency at the cost of frequent cache invalidations. Other systems [15,21,58] relax the consistency model or avoid false sharing to reduce communication overheads. CONCORDIA provides strong consistency to ease the programming, while minimizing coherence overheads.

As recent RDMA technology makes the latency and throughput of the network approach that of memory, new DSMs have emerged [20, 25, 47, 51, 61]. FaRM [25, 26, 59] offers general distributed transactions to global shared memory. Octopus [47] redesigns a distributed file system over a shared persistent memory pool. Hotpot [61] leverages non-volatile memory to incorporate both distributed memory and distributed storage. Different from the above systems, CONCORDIA focuses on cache coherence and accelerates it by exploiting new programmable network hardware.

In-Network Computation. Emerging network hardware like programmable switches poses new opportunities for in-network computation [49, 55]. NetCache [34] and DistCache [46] propose in-switch caches for load balancing. NetChain [33] designs a replicated, in-switch key-value store for distributed coordination. IncBricks [45] supports caching in the network using a programmable network middlebox. These in-network caching or key-value store systems need to keep data in the network hardware and servers coherent. They adopt similar *server-based* mechanisms to solve this problem. For example, in IncBricks, servers record the sharers list and issue invalidation commands to network accelerators when clients send `SET` or `DELETE` requests. In contrast, CONCORDIA exploits the programmable network to coordinate coherence among the cache of servers and only stores the cache's metadata in the switch.

The most similar work to ours is Pegasus [43]. Pegasus replicates the most popular objects to distribute load and leverages switches to track servers that store replicated objects. When a client issues a read request, the switch routes the request to a replica by a load-aware scheduling policy. When a client issues a write request, the switch resets the replica set to include only one server by a version-based mechanism. Pegasus's protocol is specialized and simplified for *client-server model* systems. In contrast, our FLOWCC protocol is designed for the general cache coherence problem in *symmetric model* systems (i.e., DSMs), with complex state transitions and concurrency control. In addition, CONCORDIA is a DSM but Pegasus is an object store.

9 Conclusion

We present CONCORDIA, a rack-scale DSM with in-network cache coherence; it divides coherence responsibility between switches and servers to reduce coherence overhead within the functionality and resource limit of switches. Specifically, CONCORDIA incorporates (*i*) a protocol that leverages switches to serialize and multicast requests, (*ii*) a mechanism that moves the ownership of cache blocks between switches and servers dynamically, and (*iii*) a method that makes operations in both servers and switches idempotent. CONCORDIA significantly outperforms existing solutions.

We believe that our in-network coherence protocol can also benefit other systems such as distributed file systems and hardware memory disaggregation. As storage hardware becomes extremely fast (e.g., non-volatile memory), new distributed file systems need microsecond-scale coherence for metadata caching, which can be provided by our in-network coherence protocol. Our in-network coherence protocol can also enable compute node caching for shared data in hardware disaggregated memory architectures; such caching is indispensable for reducing network accesses.

10 Acknowledgements

We sincerely thank our shepherd Kimberly Keeton for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300, 61877035), and Huawei (Grant No. YBN2019125112).

References

[1] Introducing Amazon EC2 C5n Instances Featuring 100 Gbps of Network Bandwidth. "https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-c5n-instances/", 2018.

[2] Barefoot Tofino. "https://barefootnetworks.com/products/brief-tofino/", 2020.

[3] Facebook OpenRack. "https://engineering.fb.com/data-center-engineering/open-rack/", 2020.

[4] FaRM Project. "https://www.microsoft.com/en-us/research/project/farm/", 2020.

[5] GAM Repository. "https://github.com/ooibc88/gam", 2020.

[6] Grappa Repository. "https://github.com/uwsampa/grappa", 2020.

[7] Intel RSD. "https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html/", 2020.

[8] LiveJournal Dataset. "http://snap.stanford.edu/data/soc-LiveJournal1.html", 2020.

[9] Microsoft Rack-Scale Computers. "https://www.microsoft.com/en-us/research/project/rack-scale-computing/", 2020.

[10] RDMA Raw Packet. "https://community.mellanox.com/s/article/raw-ethernet-programming--basic-introduction---code-example", 2020.

[11] TPC-C. "http://www.tpc.org/tpcc/", 2020.

[12] Twitter Dataset. "http://an.kaist.ac.kr/traces/WWW2010.html", 2020.

[13] Dennis Abts, Steve Scott, and David J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, page 11.2, USA, 2003. IEEE Computer Society.

[14] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 121–127, New York, NY, USA, 2017. ACM.

[15] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[16] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. Technical report, Pittsburgh, PA, USA, 1993.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[19] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 329–344, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[21] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[23] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[24] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[26] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[27] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[28] Kourosh Gharachorloo. The Plight of Software Distributed Shared Memory. In *Invited talk at 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*. Citeseer, 1999.

[29] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[30] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 202–210, New York, NY, USA, 1989. Association for Computing Machinery.

[31] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. In *Scalable shared memory multiprocessors*, pages 167–192. Springer, 1992.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT Coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 35–49, Berkeley, CA, USA, 2018. USENIX Association.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[35] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking Cache-Coherence Protocols with TLA+. *Form. Methods Syst. Des.*, 22(2):125–131, March 2003.

[36] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[37] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic External Memory for Switch Data Planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 1–7. ACM, 2018.

[38] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-class Datacenter Citizens. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 863–879, Berkeley, CA, USA, 2019. USENIX Association.

[39] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[40] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM.

[41] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM.

[42] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network

Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 467–483, Berkeley, CA, USA, 2016. USENIX Association.

[43] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406. USENIX Association, November 2020.

[44] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 229–239, New York, NY, USA, 1986. ACM.

[45] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore 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 '17, pages 795–809, New York, NY, USA, 2017. ACM.

[46] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, pages 143–157, Berkeley, CA, USA, 2019. USENIX Association.

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

[48] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter Hub: A Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 41–54, New York, NY, USA, 2018. Association for Computing Machinery.

[49] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on Load Distribution and the Role of Programmable Switches. *SIGCOMM Comput. Commun. Rev.*, 49(1):18–23, February 2019.

[50] Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. A Primer on Memory Consistency and Cache Coherence, Second Edition. *Synthesis Lectures on Computer Architecture*, 15:1–294, 02 2020.

[51] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[52] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 385–398, USA, 2013. USENIX Association.

[53] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[54] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[55] Dan R. K. Ports and Jacob Nelson. When Should The Network Be The Computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 209–215, New York, NY, USA, 2019. ACM.

[56] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[57] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[58] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 297–306, New York, NY, USA, 1994. ACM.

[59] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 433–448, New York, NY, USA, 2019. ACM.

[60] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 69–87, Berkeley, CA, USA, 2018. USENIX Association.

[61] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 323–337, New York, NY, USA, 2017. ACM.

[62] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGARCH Comput. Archit. News*, 20(1):5–44, March 1992.

[63] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[64] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[65] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 126–138, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[67] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.

[68] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.