2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA) | 979-8-3503-2658-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCA59077.2024.00062

HADES: Hardware-Assisted Distributed Transactions in the Age of Fast Networks and SmartNICs

Apostolos Kokolis, Antonis Psistakis, Benjamin Reidys, Jian Huang, Josep Torrellas
University of Illinois Urbana-Champaign
{kokolis2,psistaki,breidys2,jianh,torrella}@illinois.edu

***Abstract*—Transactional-based distributed storage applications such as key-value stores and databases are widely used in the cloud. Recently, the hardware on which these applications run has been rapidly improving, with faster networks and powerful network interface cards (NICs). A result of these hardware advances is that the inefficiencies of distributed software have become increasingly obvious.**

To address this problem, we analyze the sources of software overhead in these distributed transactional applications and propose new hardware structures to eliminate them. The proposed hardware includes Bloom filters for a variety of tasks and SmartNICs for efficient remote communication. We then develop *HADES*, a new distributed transactional protocol that leverages this hardware to support low-overhead distributed transactions. We also propose a hybrid hardware-software implementation of HADES. Our evaluation shows that HADES increases the throughput of distributed transactional workloads by 2.7× on average over a state-of-the-art distributed transactional system.

I. Introduction

Distributed storage systems such as key-value stores and databases are particularly important to the cloud infrastructure [11], [14], [18], [45], [51], [67]. These applications ensure that distributed data is safely stored and accessible to users on demand. Many of these storage systems use the transactional model, whereby queries are written as transactions that either complete or fail without leaving any side effect. Using transactions in storage systems is very popular [1], [2], [21], [22], [35], as it results in simpler application design.

Recently, the cloud hardware infrastructure has been rapidly improving. Networking hardware has become steadily faster. Both commercial [28], [48] and custom-designed network solutions have substantially reduced the round-trip latency of node-to-node communication—to under one microsecond in a data center [76]. Moreover, network interface cards (NICs) are including progressively more advanced hardware support [24], [42], [46], [47], [49]. Such support can enable the development of efficient RDMA operations, reducing communication overheads and off-loading work from the processor.

A result of these hardware changes is that the existing inefficiencies of distributed software protocols are becoming increasingly obvious. Applications wait for short times that cannot be effectively hidden using current hardware and software latency-hiding techniques (i.e., the well-documented killer microsecond [6], [13]). More importantly for our analysis, protocols have hefty housekeeping software overheads on the critical path that limit their performance.

Consider distributed transactional storage systems that are based on Microsoft's FaRM protocol [12], [21], [22], [71]. They have major software overheads resulting from managing and checking the read and write sets of transactions—i.e., the records that a transaction accesses plus their metadata, including versions, values, and source nodes. Other overheads result from the fact that reads and writes are supported at record granularity—forcing whole-record transfers when only some fields are needed. Additional software overheads result from many operations to lock and unlock records, poll for lock and unlock completion, and re-read records before committing to check for transaction conflicts. In our analysis, we find that such overheads are responsible for 60-70% of the execution time of various workloads in an optimized implementation of FaRM.

Given the key importance of these workloads for a thriving cloud, and that these trends are only likely to accelerate, in this paper, we introduce new hardware structures to eliminate high-overhead software operations in distributed transactional systems. We start by analyzing the sources of software overhead. Based on the analysis, we propose novel hardware that includes Bloom filters for a variety of tasks and smart network interface card (SmartNIC) support for efficient remote communication. We then develop *HADES*, a new optimistic concurrency control (OCC)-based distributed transactional protocol that leverages this hardware to provide high-performance distributed transactions. HADES is easy to use in different transactional systems, as it is agnostic to the data layout and does not require any extension to the data records. Finally, we also propose a cheaper, hybrid hardware-software implementation of HADES called HADES-H.

Using a simulation-based evaluation, we show that, compared to an optimized implementation of FaRM, HADES and HADES-H increase the average throughput of a set of distributed transactional workloads by 2.7× and 2.3×, respectively. Further, HADES shows scalability with 200 cores.

Overall, this paper's contributions are:

- Identifying and analyzing the main sources of software overhead in a state-of-the-art distributed transactional system.
- New hardware structures to eliminate these overheads and allow for large distributed transactions.
- Two new distributed transactional protocols, HADES and HADES-H, that use this hardware to provide fast distributed transactions.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00062

• A performance evaluation of HADES and HADES-H with up to 200 cores.

II. Background

Distributed transactional systems are a key component of the infrastructure in modern data centers [18], [22], [33], [40], [60], [72], [74]. They enable multiple clients to concurrently access shared data structures across distributed servers. To attain high concurrency and performance for distributed transactions, state-of-the-art systems usually leverage RDMA primitives to enable fast remote data accesses [12], [21], [22], [71]. To ensure that concurrent transactions execute in a proper way, these systems use a distributed transactional protocol [27], [29]. They augment the data records with extra fields that the software uses to manage the structures. A typical example is shown in Figure 1. In this case, a record is augmented with fields that include the record version, a lock, the incarnation to detect whether the record has been freed, and a per-cache-line version V_{Ci} to support OCC and conflict detection between concurrent transactions that operate on the same record.

Fig. 1: Augmented record to support transactions in a typical distributed transactional system.

Typically, a transactional protocol has three main phases: *Execution*, *Validation*, and *Commit*. Figure 2 shows a phase-by-phase example of an optimized software protocol. Here, a coordinator node executes a transaction with a mix of accesses to the memory of the local and remote nodes. Records A and C are read, while B and D are written.

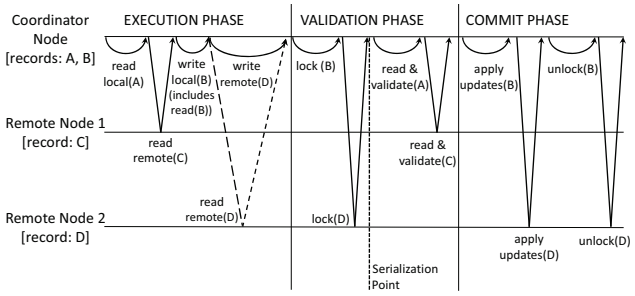

Fig. 2: Typical protocol for distributed transactions.

Execution Phase. Accesses to local records are performed locally, while accesses to remote ones are executed by sending RDMA operations to the nodes that have the records. All reads are recorded in the transaction's Read Set with the version of the records. Transactional systems usually operate at record-level granularity. As a result, before a read can be recorded in the Read Set, the atomicity of the read must be validated. This involves checking that all the cache lines of the record have the same version and, therefore, no write is interfering.

For the writes, even though a write only modifies part of a record, the system needs to first read the whole record before the write, and then apply the update. Both local and remote writes are buffered in the Write Set until the transaction commits. The transaction's Write Set includes the version, address, and data of all the written records.

Validation Phase. The coordinator needs to confirm that the transaction does not conflict with any other transaction executing on the local or remote nodes. For this reason, it first locks the local and remote parts of its Write Set. This can be done using the Compare-and-Swap (CAS) and RDMA CAS, respectively. Once locking succeeds, it can be determined whether the transaction can be serialized. Then, the coordinator fetches the data versions of all the records read, re-reads their current version numbers, and compares them to the versions that were read during Execution. The goal is to identify conflicts. If the versions have not changed, the transaction proceeds to Commit. Otherwise, the transaction is aborted and re-executed.

Commit Phase. The data versions of all the written records are updated, and the writes are performed for both local and remote records. After that, the local and remote parts of the Write Set are unlocked to allow future accesses.

III. Existing Software Overheads

We implemented an optimized version of distributed transactions based on the Microsoft FaRM protocol [21], [22]. We included optimizations as described in similar papers, including: (1) batching of messages [12], [71], [72] (i.e., sending lock/unlock operations to remote nodes in a batch during validation), (2) sending writes and unlock messages without serialization to avoid stalls [71], (3) not stalling while waiting for unlock operations [12], [71], [72], and (4) avoiding locking the read set during validation [12]. We designed the records of the key-value store as shown in Figure 1. We instrumented the code to capture the software overheads.

The left column of Table I lists the major sources of software overhead that we have seen in our optimized software implementation (called *SW-Impl*). The first source is managing the read and write sets of transactions. The *Read Set* of a transaction is the set of records that the transaction reads plus their metadata. The *Write Set* is the set of records that the transaction writes, the values written, and the metadata. In *SW-Impl*, writing a record involves two reads and two writes: a read of the record and metadata (from either a remote or a local node), then a write to the write set, and then, at commit, a read from the write set and a write to the final location.

SW-Impl also adds other software overheads in every write and read. Specifically, before performing a record write, the software needs to update the record's version. Further, on a record read, the software needs to check that all its cache lines have the same version. This is a check for atomicity, to ensure that there is no transaction writing to the record while the record is being read. This means that one cannot do zero-copy reads: one reads into a temporary location, checks the versions, and then copies the record to the destination location.

TABLE I: Reducing the overhead of distributed transactional systems.

Overhead in Current Systems (*SW-Impl*)	**Proposed Hardware to Minimize Overhead**
Manage the Read and Write sets of a transaction.	Bloom filters (BFs) next to the directory/LLC (for local accesses) and in remote NICs (for remote accesses), similar to HTM [10], [59], [75]
Before performing a write, update the version of the record.	No record versions.
On a record read, check for read atomicity. Unable to do zero-copy reads.	Use the BFs to partially lock the directory while reading multiple lines.
Operation at record granularity, which causes: (i) On a read/write, bring the whole record, and (ii) Potential increase in number of transaction conflicts.	Operation at cache line granularity.
Perform many RDMA and local operations beyond reads and writes. They include: (i) lock/unlock, (ii) poll for lock/unlock completion, and (iii) re-read record versions at validation time, to check for conflicts.	Eliminate some RDMA and local operations. Support some new RDMA messages, including *Intend-to-commit*, *Ack*, and *Validation*. Off-load RDMA operations from the core to the NIC. Use the BFs to partially lock the directory while a transaction is committing.

Other overheads of *SW-Impl* stem from the fact that reads and writes are performed at record granularity. On an access, the whole record is read rather than a few fields. Moreover, transactions conflict even when they access different fields of the same record.

SW-Impl performs many RDMA and local operations beyond the basic reads and writes. They include operations to lock and unlock, poll for lock and unlock completion, and re-read records in the Validation phase before committing, to check for conflicts (Section II). These operations add overhead.

To quantify these overheads, we execute three workloads using the Yahoo! Cloud Serving Benchmark (YCSB) [15]. The first one performs only writes (*100%WR*), the second one performs the same number of reads as writes (*50%WR-50%RD*), and the third one performs only reads (*100%RD*). Based on previous work [17], [19], [23], we create transactions using five requests at a time from a client. The workloads run on a 4-node cluster, where each node has 48 Xeon E5-2687W cores, and the nodes are connected with Mellanox ConnectX-4 NICs that perform RDMA over InfiniBand.

Figure 3 shows the execution time of the workloads, with the contribution of different components. The overheads in Table I, from top to bottom, are labeled as *Manage RD/WR Sets*, *Update Version*, *Read Atomicity*, *RD before WR*, and *Conflict Detection*. The rest of the time is labeled *Other Time*. In the figure, all execution times are normalized to 100%WR. From the data, we see that these software overheads are very significant. Their combined contribution is 59%, 65%, and 71% of the total execution time for the 100%WR, 50%WR-50%RD, and 100%RD workloads, respectively.

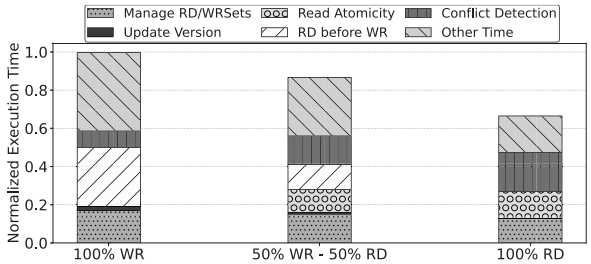

Fig. 3: Execution time with the *SW-Impl* protocol, with the contribution of the main software overheads.

In 100%WR, the highest overheads are reading records before writing them (due to operating at record granularity) and maintaining the Write Set. In 100%RD, the main overheads are: (i) re-reading the version of all records in the Read Set during validation to check for conflicts (*Conflict Detection*), (ii) ensuring the atomicity of read operations on a record read, and (iii) maintaining the Read Set. Finally, for 50%WR-50%RD, the dominant overheads are a combination of the main overheads of the other two bars.

IV. HADES Design

To improve distributed transactional systems, in this section, we introduce new hardware to eliminate some of the high-overhead software operations described above. Then, we develop *HADES*, a new distributed transactional protocol that leverages this hardware to provide fast distributed transactions.

A. Hardware to Minimize Software Overheads

The right column of Table I lists our proposed hardware designs to minimize the software overheads. First, to manage the Read and Write sets of transactions, HADES uses read and write hardware Bloom Filters (BF), similar to their use in Hardware Transactional Memory (HTM) [10], [59], [75]. A transaction owns a pair (Rd,Wr) of *local* BFs in the local node and a pair of *remote* BFs in each of the remote nodes from where the transaction accesses data. The pair of local BFs are next to the local directory/LLC. Transparently to the software, they record the addresses of the accesses to the local node's memory. A pair of remote BFs exist in the NIC of a remote node, and record accesses by the transaction to that remote node's memory. The BFs help transaction conflict detection.

HADES eliminates the software overhead of updating the versions of records because there are no versions. Instead, HADES uses hardware to detect conflicts. Further, it also eliminates the software overhead of checking for read atomicity. The reason is that HADES introduces a hardware mechanism where a transaction can use its BF to partially lock the directory, preventing other transactions from concurrently writing the same lines that the transaction is reading.

HADES eliminates the overheads stemming from performing reads and writes at record granularity because its hardware nature enables it to operate at cache line granularity.

To further reduce overhead, HADES eliminates some of the RDMA and local operations performed by the conventional system. Further, HADES supports some efficient new RDMA

operations, including *Intend-to-commit*, *Ack*, and *Validation*, which trigger actions at the receiving NIC. In addition, several of these operations are off-loaded from the core and executed in the NIC. Finally, HADES leverages the partial directory-locking hardware mechanism mentioned above while committing a transaction, preventing other transactions from issuing conflicting accesses.

B. Chosen Distributed Transactional System

To showcase the impact of HADES, we use a cluster of N nodes, each with C cache-coherent cores. Each database record has its home in one of the nodes. Hence, when a core accesses a record for the first time in the transaction, it issues a local or a remote access depending on where the record's home is. During the transaction, the record is reused locally. When the transaction commits, all the remote records that it has updated are written to their home nodes.

Remote data are accessed via RDMA requests that take as argument the range of contiguous addresses accessed. We use one-sided RDMA since it reduces core costs and latency [61]. Local data are accessed with loads and stores. The same is the case for accesses to local copies of remote data.

Both remote and local accesses from a transaction i can conflict with accesses from another transaction j running on the same node (i.e., a local transaction) or on another node (i.e., a remote transaction). We call a conflict between two local accesses an L–L conflict, a conflict between a local and a remote access an L–R conflict, and a conflict between two remote accesses an R–R conflict.

Figure 4a shows a local (L) and a remote (R) access. Figure 4b shows an L–R conflict and an L–L conflict. An R–R conflict occurs in a node that is accessed remotely by two transactions.

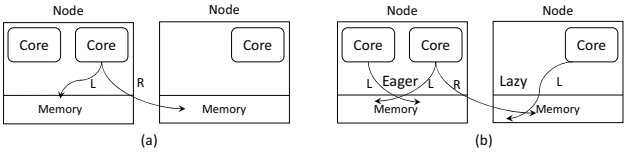

Fig. 4: Example of transaction conflicts.

We design the HADES transactional protocol as follows. Conflicts that involve at least one R access are detected *lazily* when the first of the two conflicting transactions commits; the transaction that commits first does squash the other one. On the other hand, conflicts where both of the accesses are L are detected *eagerly* as soon as the second access occurs; the transaction that issues the second access squashes itself. When we describe the protocol, it will be apparent that these decisions are natural given the hardware envisioned.

C. Overview of the HADES Hardware

The top left part of Figure 5 shows a node with multiple cores. The five circles numbered ① to ④b denote where the HADES hardware extensions are. Then, the rest of Figure 5

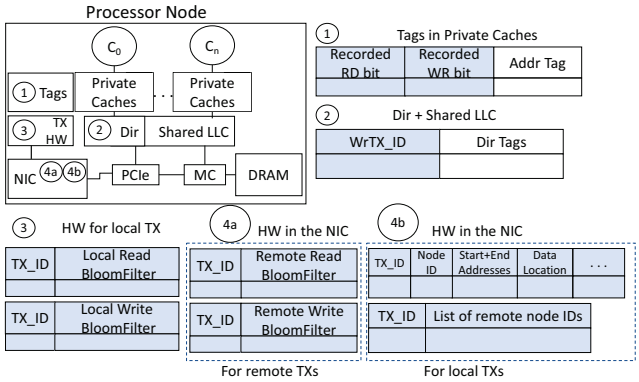

Fig. 5: Node with the HADES hardware modules shaded.

expands each of the five modules, shading the actual HADES hardware. In the figure, *TX* means transaction.

Module ② is a *Writing-Transaction* ID tag (WrTX_ID) added to each directory/LLC entry. It records the ID of the in-progress transaction that wrote to that line. Module ① is *Recorded RD* and *Recorded WR* bits added to the lines of the private caches that act as filters to avoid accessing WrTX_ID at every access.

Module ③ is the *Local Read Bloom Filters* and the *Local Write Bloom Filters* of all the local transactions. They encode the local addresses read and written, respectively, by the local transactions. Executing transactions dynamically pick their BFs from a set of BFs. Although Figure 5 shows a monolithic LLC, the LLC and the BFs are sliced.

Module ④a is the *Remote Read Bloom Filters* and the *Remote Write Bloom Filters* of all the in-progress remote transactions that have accessed data homed in this node. They encode the local addresses read and written, respectively, by the remote transactions.

Module ④b records, for each local transaction: (1) upper structure: the addresses of the remote locations that it *wrote*, tagged by the remote node ID—together with a pointer (*Data Location* in the figure) to a local buffer that contains the values written; and (2) lower structure: a list of the IDs of the remote nodes that home the data *read or written* by the local transaction. All this information is used when a local transaction commits.

Each entry in modules ③, ④a, and ④b is tagged with the ID of the owner transaction (TX_ID).

D. Data Buffering and Conflict Detection

Consider a HADES transaction running on Core i of Node x. Data is local if its home is x and remote otherwise. The local data written by the transaction is buffered in the local cache hierarchy (including the shared LLC) and cannot be evicted to memory. The record of local lines read is encoded in a Local read Bloom filter (BF). The record of local lines written is encoded in a Local write BF and in Writing-Transaction ID (WrTX_ID) tags in the directory/LLC.

The remote data written by the transaction is buffered in the local NIC. The record of remote lines read and remote lines

written by the transaction that are homed in a remote Node y is encoded in a Remote read BF and a Remote write BF, respectively, at the NIC of Node y. These BFs detect conflicts.

We now consider how conflicts between transactions are detected. Recall that L–L conflicts are detected eagerly. On a local read by transaction i, the address is checked against the WrTX_ID tag in the directory/LLC. On a local write by transaction i, the address is checked against the WrTX_ID tag in the directory/LLC and against the Local read BFs of all the other local transactions.

Recall that L–R and R–R conflicts are detected lazily when the first transaction of the pair commits. Assume, without loss of generality, that transaction i running on Node x commits first. At the local node, the local addresses written by i are checked against the NIC-resident Remote read and Remote write BFs of all the remote transactions that accessed data homed in x. In addition, at any remote Node y that homes remote data written by i, the following is done. The addresses of the remote data written by i are checked: (i) against the Remote read and Remote write BFs of all the other remote transactions in y to detect R–R conflicts, and (ii) against the Local read and Local write BFs of all the local transactions in y to detect L–R conflicts at y.

V. HADES Transactional Protocol

We propose two versions of HADES: a hardware-only one and a hybrid one. The latter replaces the local component of the hardware-only protocol with software, to simplify the hardware design. Next, we describe both versions.

A. Hardware-only HADES Protocol

In this discussion, we label transactions (sometimes referred to as Cores) as $\{i, j, ...\}$ and nodes as $\{x, y, ...\}$. Table II details the protocol followed by Transaction i running on Core i of Node x. We now describe each operation in turn.

Remote Read/Write. Assume that i accesses a remote datum homed in y. In this case, Core i sends an RDMA request to Node y. If this is a read, the addresses of the set of cache lines requested are encoded in the Remote read BF (RemoteReadBF) of i in the NIC of y (Module (4a) of Figure 5). Then, the lines are fetched to Node x. If this was a write, a similar process is followed, except that we only need to care about the cache lines that are partially written. Such lines can be found at the beginning and end of the range of addresses written. The addresses of such lines are encoded in the RemoteWriteBF of i in the NIC of Node y (Module (4a)) and the lines are fetched to Node x. The other lines are not fetched to Node x because they will be overwritten, and do not need to be inserted in the BF as we will see. From now on, Node x buffers i's updates to all the addresses of the datum.

Local Read/Write. Assume that i accesses a datum homed in Node x. The hardware accesses the WrTX_ID tag in the directory (Module (2) of Figure 5) to check if another local transaction has written the line. If so, i is squashed. Note that the filter bits in the private caches (Module (1)) are first checked and, if the *Recorded WR* bit is set, there is no need to access the directory because it is guaranteed that the WrTX_ID tag in the directory is set to i. For simplicity we do not describe Module (1), but note that, on context switch, the Module (1)

TABLE II: Operation of a Transaction i running on a Node x. References in circles correspond to the modules in Figure 5.

Remote Read/Write by i
* Send request to Node y * If Read - Add addresses of lines read to RemoteReadBF$_i$ in NIC of Node y (4a) - Fetch the lines to local node * If Write - Add the addresses of partially written lines to RemoteWriteBF$_i$ in NIC of Node y (4a) - Fetch the partially written lines to local node - From now on: buffer the updates to all the datum's lines (not just to the partially written lines) in Node x
Local Read/Write by i
* Use WrTX_ID tag in the local directory (2) to check if another local transaction wrote the line. If so, squash yourself * If Write - *Additionally* check the other LocalReadBF$_{j,k,..}$ (3) to see if another local transaction read the line. If so, squash yourself * If Read - Add address read to LocalReadBF$_i$ (3) * If Write - Add address written to LocalWriteBF$_i$ (3) - Update the WrTX_ID tag in the local directory (2)
Transaction Commit by i**. At Local Node** x
* i partially locks the local directory or gets squashed * Detect any conflict on local data between i and a remote trans. - Find the lines with i's tags in the local directory (2) and probe for membership in all RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in x's NIC (4a) - Send squashes to any conflicting remote transactions * Request the commit of i in remote nodes - Send *Intend-to-commit* RDMA message to all remote nodes involved in the transaction, passing the address ranges written - Receive *Acks* from all the remote nodes involved in the trans. - After this, i cannot be squashed anymore * Clear i's local speculative state - Find the lines with i's tags in the local dir. (2) & clear their tag * Send *Validation* plus *updates* to all the remote nodes involved in the transaction to clear i's remote state and push the updates * Unlock local dir. & clear LocalReadBF$_i$ and LocalWriteBF$_i$ (3)
Transaction Commit by i**. At Remote Node** y
* NIC receives i's *Intend-to-commit* RDMA message with the addresses written * Partially lock y's directory for i or squash i * Detect any conflict on y's local data between i and any transaction local or remote to y - Take each address written by i homed in y, and check for membership in: = All other RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in y's NIC (4a) and = All LocalReadBF$_{l,m,..}$ and LocalWriteBF$_{l,m,..}$ in y (3) - Squash all transactions conflicting with i * Send *Ack* to i in x * Receive *Validation* plus *updates* from i * Push the updates to y's local memory or LLC * Unlock y's directory for i and clear RemoteReadBF$_i$ and RemoteWriteBF$_i$ (4a)

bits are cleared.

On a write, we *additionally* check the LocalReadBF$_{j,k,..}$ of all the other local transactions (Module ③) to see if another local transaction read the line. If so, i is squashed.

If i survives, the hardware performs: on a read, the line address is encoded in LocalReadBF$_i$ (Module ③); on a write, the line address is encoded in LocalWriteBF$_i$ (Module ③) and the WrTX_ID tag in the directory is set (Module ②).

Transaction Commit. To commit i, HADES requires several steps in x and some steps in each of the remote nodes $\{y, z...\}$ from where i accessed remote data.

a) Actions in Node x. There are six steps in Node x (Table II):

Step 1. To ensure that commits have a total order, the commit of i starts with i partially locking the local directory. This mechanism will be explained in Section V-B and consists of using the LocalReadBF$_i$ and LocalWriteBF$_i$ (Module ③) to temporarily and selectively block accesses to the lines in the directory whose addresses are encoded in these BFs. This operation prevents other transactions from performing conflicting accesses while i commits. If i fails to lock the directory because another transaction is already locking common lines, i gets squashed. After partially locking the directory, i is guaranteed not to get squashed due to any type of local conflict.

Step 2. HADES detects any conflict on local data between i and a remote transaction. For this, the hardware takes each of the directory lines whose WrTX_ID tag (Module ②) matches i, and checks them for membership in all the RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in the NIC of Node x (Module ④a). If a match is detected, a squash is sent to the conflicting remote transaction. Section V-C shows the hardware structures proposed to easily obtain the directory lines whose tag matches a certain WrTX_ID, and the structures proposed to check for BF membership.

Step 3. HADES requests the commit of i in remote nodes. For this, the local NIC sends an *Intend-to-commit* RDMA message to all remote nodes $\{y, z...\}$ involved in the transaction, passing the range of addresses homed in the corresponding node that were written by i. On reception of the message, such nodes will inititate the commit of i by performing the actions that will be described below. If the operations are successful, the nodes will return an *Ack* to i. When x's NIC has received all *Acks*, i cannot be squashed anymore.

Before i receives all the *Acks*, however, i can still receive squash messages, which will result in the squash of i and the notification of it to all the nodes involved in the transaction. We explain this case later.

Step 4. Since i is now free of squashes, it clears i's local speculative state. Specifically, HADES finds all the lines with i's WrTX_ID tags in the local directory (Module ②) and clears their tag.

Step 5. The NIC in x sends a *Validation* RDMA message to all the remote nodes involved in the transaction, asking them to clear i's remote state. The message includes i's updates to the data homed in the corresponding remote node, if any. The receiving nodes clear RemoteReadBF$_i$ and RemoteWriteBF$_i$ in their NIC (Module ④a) and push the updates to their local memory or LLC.

Step 6. As the *Validation* messages are sent, i unlocks the local directory (Section V-B) and clears LocalReadBF$_i$ and LocalWriteBF$_i$ (Module ③). All of i's state has disappeared.

b) Actions in Nodes $\{y, z...\}$. Recall that remote nodes $\{y, z...\}$ receive the *Intend-to-commit* message from i, with the addresses of data homed in those nodes that i wrote, if any. Each of the nodes, say y, performs five steps (Table II):

Step 1. To ensure correctness, the hardware attempts to partially lock y's directory for i. The operation involves using RemoteReadBF$_i$ and RemoteWriteBF$_i$ (Module ④a) to temporarily and selectively block access to lines in y's directory that are encoded in these Bloom filters. If the hardware fails to lock the directory, a squash is sent to i. After partially locking the directory, i is guaranteed not to get squashed due to any type of conflict in Node y.

Step 2. HADES detects any conflict on y's local data between i and any transaction local or remote to y. For this, the hardware takes each address written by i that is homed in y and checks for membership in: (i) all other RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in y's NIC (Module ④a) and (ii) all LocalReadBF$_{l,m,..}$ and LocalWriteBF$_{l,m,..}$ in y (Module ③). If a match is detected, a squash is sent to the transaction conflicting with i.

Step 3. y's NIC sends an *Ack* to i in x and waits for *Validation*.

Step 4. On reception of the *Validation* plus the local *updates* from i, HADES pushes the updates to y's memory or LLC.

Step 5. y unlocks its directory for i and clears RemoteReadBF$_i$ and RemoteWriteBF$_i$ (Module ④a).

Figure 6 is the HADES protocol using the Figure 2 layout.

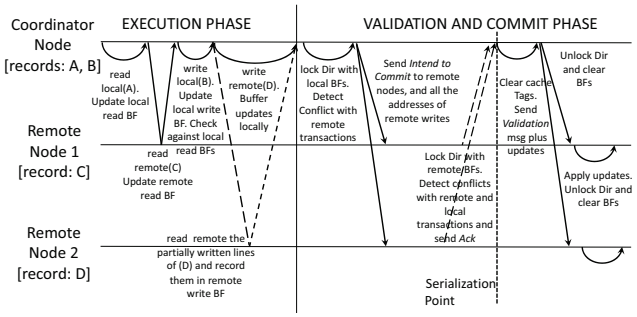

Fig. 6: HADES protocol for distributed transactions.

Transaction Squash. The previous discussion showed that i is squashed while trying to commit if it cannot lock the directories. It may also be squashed when a conflict with another transaction is detected—either while i is not committing or while i is committing but before it receives all *Acks*.

i can conflict with another local or a remote transaction on local or remote data. A conflict with another local transaction j on local data is detected eagerly when i attempts to write to a line that has been read or written by j, or i attempts to read a line that has been written by j. A conflict with a local

transaction j on remote data or with a remote transaction k on local or remote data is detected when the first transaction of the two commits. Finally, i is also squashed when a line written by i is evicted from the LLC.

HADES supports context switches in a transaction without squashing it: on a context switch, the Recorded RD and Recorded WR bits in the private caches (Module ① in Figure 5) are cleared, but the WrTX_ID tags in the LLC (Module ② in Figure 5) remain. A read or write to a private cache line with a cleared Recorded RD or Recorded WR bit, respectively, is a cache miss. After the access, the corresponding bit is set.

Fault-Tolerance and Durability. Since this topic requires extensive discussion that is beyond what we can cover, we only outline our approach. HADES can attain fault-tolerance by replicating variables in one or multiple nodes [34]. This requires extending the protocol so that a write operation now creates messages that update the replicas in other nodes [38]. The update of these replicas needs to be completed by the time the transaction commits. Also, to ensure durability, these updated replicas need to be persisted to SSDs, HDDs, or NVM by the time the transaction commits.

The process to update or persist one or more of these replicas can fail—e.g., a message can get lost or a memory module may fail. These events are detected and handled correctly by leveraging HADES' two-phase commits (like FaRM). Specifically, the node with the committing transaction first sends the *Intend-to-commit* to all the nodes with replicas. Each of these nodes updates the replica and persists it in a temporary durable storage before responding with an *Ack*. When the initiator receives all the *Acks*, it knows that the transaction has succeeded and sends a *Validation* to all replica nodes (so they can move their replica from the temporary durable storage to a permanent one). If, instead, at least one of the replicas does not return an *Ack*, the initiator sends an abort message to all replica nodes and the transaction fails.

B. Hardware Primitive to Support Atomicity

For correct operation, while transaction i is committing, no other transaction should perform accesses to addresses that conflict with i's accesses. To ensure this capability, HADES introduces a hardware primitive that allows i to *partially lock the directory*, conservatively preventing other transactions from performing conflicting accesses. As a transaction i proceeds to commit, it first invokes such primitive.

The idea is to copy the Read and Write BFs of i to a *Locking Buffer* next to the directory (Figure 7). Then, every write that accesses the directory/LLC is checked for membership in the Read and Write BFs, while every read is checked for membership in the Write BF. If any of these checks is positive, the access is denied and needs to retry. Otherwise, the access proceeds as usual. Note that these checks are performed in parallel with the directory/LLC tag check.

In the HADES protocol, when transaction i tries to commit, it first locks its local directory with LocalReadBF$_i$ and LocalWriteBF$_i$, and then the directory in each relevant remote Node y with RemoteReadBF$_i$ and RemoteWriteBF$_i$. Note that

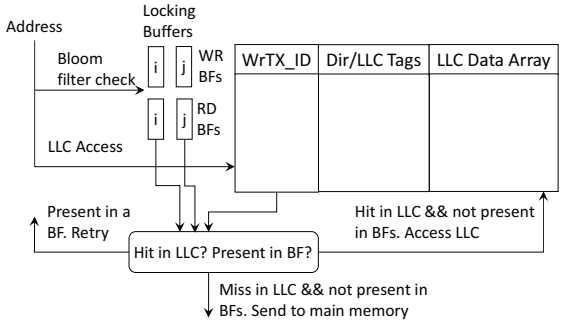

Fig. 7: Partially locking the directory by transactions i and j using their Bloom filters (BFs).

blocking access to the directory/LLC is enough. There is no need to block access to the private cache hierarchies because every first write and first read of a transaction to a line needs to propagate to the directory/LLC to check and (for writes) set the WrTX_ID tag. The purpose of the Recorded RD and WR bits in Module ① of Figure 5 is to filter the subsequent accesses.

At a given node, multiple transactions can commit at a time if they do not have conflicts. To support this case, as shown in Figure 7, our hardware has multiple Locking Buffers to store the BFs of multiple committing transactions. To see how it works, consider when transaction i wants to commit in Node x and finds that j is already partially locking the directory.

i's first step is to generate the list of cache line addresses it wrote. Generating such list is easy. If x is a remote node for i, then the list is available in the just-received *Intend-to-commit* message. If x is the local node for i, the list is obtained from the directory/LLC's WrTX_ID tags with the hardware that will be described in Section V-C.

Once the list of write addresses is available, the addresses are checked for membership in the Read and Write BFs of j. If there is a match, the two transactions conflict and i is squashed—they cannot both commit concurrently. Otherwise, i's BFs are loaded into the second buffer of Figure 7, effectively adding a second partial lock to the directory. The BFs loaded are RemoteReadBF$_i$ and RemoteWriteBF$_i$, or LocalReadBF$_i$ and LocalWriteBF$_i$, depending on the case. At the end of commit, the unlock operation clears the Locking Buffer in the local node and in any relevant remote nodes.

This primitive is also used by HADES to avoid checking for read atomicity when a transaction performs a read that covers multiple cache lines (Row 3 of Table I). In this case, the hardware hashes the addresses of the multiple lines into one of the read BFs of the Locking Buffers. Any concurrent write that tries to access one of these lines stalls in the directory/LLC while the reads are in progress.

The accesses to the Locking Buffers do not impact the critical-path of LLC accesses. The LLC and each individual Locking Buffer are accessed in parallel—there is no serialization of Locking Buffer access. Accessing the LLC involves

accessing the large tag array, performing tag comparison, going through the multiplexer, and detecting the hit. In parallel, we hash the address and access a BF.

C. Other Hardware Primitives

HADES uses two more hardware primitives that operate on BFs: one detects membership of an address in a BF; the other quickly finds the lines in the directory/LLC that have been written by a given transaction ID. Detecting membership of an address in a BF is a well-known operation that involves hashing the address and then checking if the resulting set bits are in the BF [8].

Identifying the set of lines in the LLC that have been written by a given transaction is needed in three operations. The first one is a transaction squash: all the LLC lines tagged with the transaction's WrTX_ID are identified and invalidated. The second operation occurs at the end of a transaction commit: all the LLC lines tagged with the transaction's WrTX_ID have their WrTX_ID cleared because they become non-speculative.

The third operation occurs when a local transaction i commits, and needs to check for conflicts against remote transactions {j, k, ...} on local data. Transactions {j, k, ...} have their RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in the NIC of the local node (Module (4a)). To check for conflicts, one needs to first collect the set of local cache line addresses tagged with i's WrTX_ID. Then, these addresses are checked for membership in RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$.

Note that the opposite case, where a committing remote transaction j needs to check for conflicts against a local transaction i on local data is easier: we already have a list of local line addresses written by j: they are included in the *Intend-to-commit* message received from the node on which j runs (Table II).

To collect the set of lines written in the LLC by a transaction, we organize the write BF in the following way. We logically divide it into two sections: *WrBF1* and *WrBF2*. *WrBF1* is filled by hashing addresses using a conventional hash function (e.g., CRC [52], [68]); *WrBF2* is filled by taking the LLC index bits of addresses and applying modulo WrBF2 size. As a result, each bit of WrBF2 corresponds to a few sets in the LLC (e.g., 4 or 8) and, if the bit is set, it tells that WrBF2 has an address that maps to such sets. An example is shown in Figure 8. In the figure, the WrBF2 has 4 bits, and the LLC has 8 sets. Hence, if bit 2 of WrBF2 is set, it means that WrBF2 has at least a line that maps to sets 2 or 6.

With this BF design, address insertion in the BF and membership detection work as usual. However, this design allows fast, parallel detection of the LLC lines tagged with a given WrTX_ID. As shown in Figure 8, each set bit in WrBF2 enables a group of LLC sets. The enabled sets compare an input thread ID (TID) with all the WrTX_ID tags of all their ways. (The figure uses TID rather than WrTX_ID). Once the matches are found in parallel, retrieving the address tags does not consume excessive time, since the number of matches is typically modest—around 5 for our workloads. These addresses are the ones written by the TID transaction.

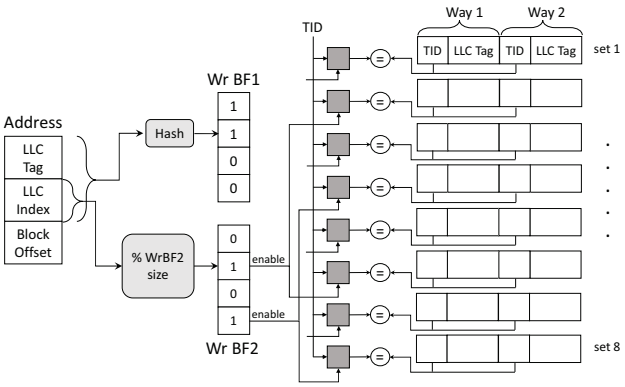

Fig. 8: New write BF design to quickly identify the lines in the LLC written by a given transaction. The figure uses TID next to the tag rather than WrTX_ID for simplicity.

D. HADES-H: A Hybrid Protocol

To help integrate HADES more easily into current hardware, we also propose a simplified design called HADES-Hybrid (HADES-H). It minimizes the hardware changes to the processor and uses software for some of the protocol operations.

The idea is to support the local operations in software, like in *SW-Impl* (Section III), and the remote operations in hardware, like in HADES. To support remote operations, the NIC has the same hardware as in HADES. Since the local operations are in software, the processor does not have local Bloom filters or special tags in caches or directories. Of the hardware structures in Figure 5, HADES-H only has Modules (4a) and (4b); the rest of the modules are eliminated. However, for efficiency, HADES-H retains the hardware primitive that partially locks the directory/LLC (Section V-B). With this design, nearly all the changes are concentrated in the NIC.

During transaction execution, remote reads and writes are tracked in the NIC hardware at cache line granularity like in HADES. However, local reads and writes are tracked and recorded in software at record granularity in Read and Write sets like in *SW-Impl* (Section III). For this reason, data records are augmented as in Figure 1. Moreover, conflicts on local data are not detected eagerly by comparing directory tags. Instead, they are detected in software as in *SW-Impl*, during the validation phase before the commit: the software re-reads from local memory all the local records that exist in the transaction's Read and Write sets and checks that the versions have not changed due to an intervening write. Note that the Write set is also checked because local operations are executed at record granularity. We call this process the *Local Validation*.

At commit time, the operations of Table II are followed with some modifications. When a local transaction i attempts to commit, the software passes the addresses of all the local records read or written by i to the local NIC. The NIC uses them to build the equivalent of a LocalReadBF$_i$ and LocalWriteBF$_i$ for i. Then, these BFs are placed in one of the Locking Buffers of the node to partially lock the directory.

If locking is unsuccessful, i is squashed. Otherwise, the commit proceeds as in HADES: the NIC checks LocalWriteBF$_i$ against the NIC's BFs to identify L–R conflicts, and then initiates the remote commit process in all other nodes involved in the transaction.

The remote commit process in a remote Node y starts like in HADES. However, after the hardware succeeds in partially locking y's directory for i, and has checked for conflicts between i and other remote transactions in y, it cannot check for conflicts between i and local transactions in y. The reason is that local transactions do not have BFs. Hence, y will return an *Ack* to i without checking for conflicts with local transactions. When Core i receives all the *Ack*s, it invokes software to perform the *Local Validation* of transaction i. If the local validation fails, i is squashed; otherwise, i merges its local updates to local memory or LLC, and terminates the commit transaction as in HADES.

When local transactions in nodes such as y attempt to commit, they will perform their *Local Validation*. If they had a conflict with transaction i, they will discover it at that time and will squash themselves.

Overall, HADES-H eliminates most of the processor hardware at the cost of adding software overhead.

VI. Additional Considerations

Protocol Deadlock and Livelock Issues. The HADES protocol does not deadlock or livelock. On data conflicts, transactions are squashed and restarted. When two local transactions conflict on local data, the second conflicting access eagerly triggers a squash of the transaction that issued it. When at least one of the conflicting accesses is remote, the conflict is resolved lazily at commit time. The first transaction to commit detects the conflict and squashes the other. When two transactions conflict in multiple nodes on remote data, it is possible that both transactions get squashed. Also, recall that a transaction also gets squashed when it attempts to partially lock a directory and fails because there is already a partial lock that conflicts with it.

A transaction may be repeatedly squashed. To avoid livelock, HADES uses the same strategy as FaRM. Specifically, when a transaction is squashed more than a certain number of times, it stops using an OCC protocol. Instead, it first locks all data that it will need (i.e., it gets all permissions) and then executes the transactional code.

Filter Bits in the Private Caches. Recall that the private caches (L1 and L2) have filter bits (Module ① in Figure 5) to avoid the extra traffic to the LLC on read or write requests to addresses that were previously read or written by the transaction. On a context switch, these bits are cleared, to guarantee that accesses to these cache lines by the incoming thread will first go to the LLC for conflict detection. This will correctly identify conflicts between transactions executing on the same core. If the core supports SMT, then the filter bits are augmented with a TX_ID so that accesses from different transactions can be disambiguated.

Supporting Context Switches. We envision that, on a context switch, the running transaction is not typically squashed. The filter bits in the private caches are cleared, but the WrTX_ID tags in the LLC and the BFs of the outgoing thread are kept in place, without saving them. The OS gives the incoming thread the BFs that it was using when it was preempted during a transaction. When threads start a transaction, they get a new pair of BFs. If a core runs out of BFs, no new transaction can start until another transaction completes or gets squashed.

Hardware Modifications and Scalability. Transactional-based key-value stores and databases are key workloads. To obtain performance beyond FaRM-like software protocols, hardware changes such as those of HADES and HADES-H are needed. As shown in Figure 5, HADES requires hardware changes in the processor (Modules ①, ②, and ③) and in the NIC (Modules ④a and ④b). HADES-H eliminates all the processor hardware except the mechanism to partially lock the directory. However, it has software overheads.

HADES relies heavily on BFs, which are area- and energy-efficient structures to quickly check for membership. They are ideal for conflict detection in transactions. The BFs and the cache tags are automatically set in hardware when the processor issues reads/writes inside a transaction.

To compute the hardware needed by HADES, assume N nodes, C cores per node, m multiplexed transactions per core, and an average number of remote nodes accessed per transaction equal to D. Then, each node needs $m \times C$ pairs of BFs (read and write), each LLC line needs $\log_2(m \times C)$ bits for WrTX_ID, and each NIC needs $m \times C \times D$ pairs of BFs in Module ④a plus $m \times C$ entries in the structures in ④b.

Based on the parameters of Section VII, a pair of core BFs take 0.7KB, a pair of NIC BFs in Module ④a take 0.25KB, and the entries for a single TX_ID in Module ④b take less than 100B. One cluster evaluated in Section VIII has N=5, C=5, and m=2. In this case, the storage needed per node is: 7.0KB for 10 pairs of core BFs, 4 bits in the LLC tags, and 11.0KB in the NIC (for 40 pairs of BFs and 10 TX_IDs in Module ④b).

The HADES hardware is scalable. A larger system can be the one considered in a FaRM paper [61], where N=90, C=16, and m=2. In this case, if we assume that $D = 5$, the storage needed per node is: 22.4KB for 32 pairs of core BFs, 5 bits in the LLC tags, and 43.1KB in the NIC (for 160 pairs of BFs and 32 TX_IDs in Module ④b). Note that an NVIDIA NIC currently has up to 4MB of memory and can incorporate even more [55].

If the transaction concurrency exhausts the local BFs, we can gracefully degrade to HADES-H during intervals, since HADES-H needs no local BFs.

VII. Evaluation Methodology

Modeled Architecture. We model a default cluster architecture of N=5 nodes, C=5 cores per node, and m=2 multiplexed transactions per core. Later, in a section on scalability, we model larger machines of up to 200 cores. Each node has 64 GBs of memory and a NIC that supports RDMA. The

architecture parameters are shown in Table III. Each core is out-of-order, has private L1 and L2 caches, and has a shared LLC. In the table, *Find LLC Tags* refers to finding all the lines in the LLC that are tagged with a given WrTX_ID (Section V-C). The area and power of the structures are computed with CACTI 7.0 [5] at 22nm.

TABLE III: Architectural parameters used for evaluation.

Cluster Architecture Parameters	
Nodes (N) & cores/node (C)	Default: N=5 nodes, C=5 cores per node, and m=2 multiplexed transactions per core. For scalability analysis, we try: N=10 and C=5; N=5 and C=10; N=8 and C=25.
Core	out of order, 6 issue, 2GHz
Ld-St queue; ROB	92 entries; 192 entries
L1 cache	64KB, 8-way, 2 cycles round trip latency (RT)
L2 cache	512KB, 8-way, 12 cycles RT
LLC cache	4MB/core, 16-way, 40 cycles RT
Core Bloom filters	Read: 1024 bits; Write: 512 bits with CRC hashing plus 4096 bits with cache indexing hashing. Each pair of Rd+Wr BF: 0.7KB of storage. Rd BF: 12.8/12.7pJ per rd/wr; 1.7mW leakage. Wr BF: 12.8/13.1pJ per rd/wr; 1.9mW leakage.
Find LLC Tags	80 - 120 cycles typical
CRC hash function	2-cycle latency; Area: $1.9 * 10^{-3} mm^2$; Dyn. energy: $0.98pJ$; Leak. power: $0.1mW$
Network Parameters	
Network latency	$2\mu s$ RT NIC-to-NIC RDMA latency
Network Bandwidth	200Gb/s
NIC Bloom filters	Read: 1024 bits; Write: 1024 bits. Each pair of Rd+Wr BF: 0.25KB of storage. BF: 12.8/12.7pJ per rd/wr; 1.7mW leakage.
Other NIC hardware	Structures per TX_ID: 90B of storage
Per-Node Main-Memory Parameters	
DRAM	64GB, 4 Channels, 8 Banks, 100ns read/write RT
Freq; Bus width	1GHz DDR; 64 bits per channel

We use RDMA for low-latency data transfers between nodes without involving the remote processor, and augment it with the operations required by HADES. These operations include support for: (1) remote read and write operations that need to update the NIC hardware structures, (2) Intend-to-commit, Ack, and Validation messages of our protocol, and (3) squashing transactions on a conflict. We model a high-end NIC with a bandwidth of 200Gb/s [46], and up to 400 Queue Pairs [69] for scheduling messages. The round-trip latency of a message between two NICs is 2μs [3], [33], [46], [58]. We model the latency of adding elements to the BFs, checking for conflicts, and using BFs to partially lock the directory.

Modeling Approach. We use the SST simulator [56], Pin [44], and the DRAMSim2 memory simulator [57]. With Pin, we collect instruction traces for a given number of cores processing read and write client requests. Traces have no timing information. Then, we take these traces and feed them to the same number of cores of our distributed architecture. Timing is dynamically determined by the simulator. The simulator models all the protocol messages required for the execution, validation, and commit phases of transactions. In the case of transaction conflicts, when a transaction is squashed, we restart the transaction from its first instruction and follow the same instruction path. Records are statically distributed across all the nodes in a uniform manner.

Configurations and Applications. We compare three configurations: *Baseline* (the optimized implementation of the software-only approach for distributed transactions [12], [21], [71] that we called *SW-Impl* in Section III), HADES, and HADES-H (which uses software for local operations and hardware for remote operations). We use three distributed transactional applications and four key-value stores. The transactional applications are *TPC-C* [66], *TATP* [62], and *Smallbank* [4], [63]. TPC-C is an OLTP benchmark that simulates an order-processing application. We fill the TPC-C warehouses with 10M items. TPC-C is write intensive and has many record accesses per transaction at a fine granularity. TATP is an OLTP benchmark that simulates a telecommunication database with 1M subscribers. It has 80% read and 20% write requests, and a small number of requests per transaction. Smallbank [4], [63] is a write-intensive OLTP benchmark (46% write requests) that simulates bank account transactions on 5M accounts.

The key-value stores are *HashTable (HT)*, *Map*, *B-Tree* [26] and *B+Tree* [7]. We evaluate them with Yahoo! Cloud Serving Benchmark (YCSB) [15] running write-intensive workload-A (*wA*) (50% writes, 50% reads) and read-intensive workload-B (*wB*) (5% writes, 95% reads), using a zipfian distribution. We fill the key-value stores with 4M keys. Like prior work [17], [19], [23], we select transactions to be 5 client requests.

In our experiments, we warm up the architectural state by running 1B instructions before simulating 25B instructions.

VIII. Evaluation

We first assess HADES' gains in throughput and latency reduction, and then characterize the HADES structures, perform a sensitivity analysis, and consider HADES' scalability.

A. Improving Transaction Throughput

Figure 9 shows the transaction throughput in committed transactions per second of our applications with Baseline, HADES-H, and HADES, normalized to Baseline. We see that both HADES and HADES-H substantially boost the throughput over the state-of-the-art software-based Baseline. On average, HADES-H and HADES attain 2.3× and 2.7× higher throughput, respectively.

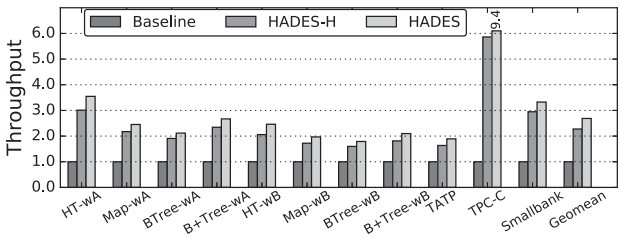

Fig. 9: Transaction throughput normalized to Baseline.

HADES delivers very high throughput for TPC-C. The reason is that a typical TPC-C transaction issues many small requests (about 13.5), while executing relatively few instructions. As a result, the software overheads of these transactions

in Baseline are high. In particular, managing the Read and Write sets, ensuring read atomicity, and reading the whole record in an access adds significant performance overheads.

For the key-value stores running with YCSB, HADES achieves higher gains for the write-intensive *wA* than for the read-intensive *wB*. This is because, in Baseline, writes have the overheads of fetching the record before writing and of updating the record version. In constrast, read-only transactions in Baseline do not need to lock any records. Such transactions validate the read set for conflicts by re-reading the version during Validation; if no conflict is detected, the transaction commits. This saves a network round-trip to lock remote records, and the execution time to lock local records.

We observe the same behavior for the read-intensive TATP and write-intensive Smallbank workloads.

B. Reducing Transaction Latency

Figure 10 shows the mean latency of the transactions for the different applications in each of the configurations, normalized to Baseline. We break down the transaction latency into the Execution, Validation, and Commit phases. HADES-H and HADES only have Execution and Validation categories. Compared to Baseline, HADES-H and HADES reduce the mean transaction latency by 54% and 60%, respectively.

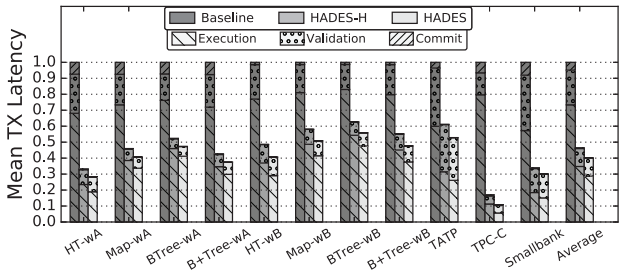

Fig. 10: Mean transaction latency normalized to Baseline.

In Baseline, Execution accounts for most of the latency. HADES mitigates much of the overheads during this phase. HADES does not manage Read/Write sets, check the atomicity of read operations, or operate at record granularity. As a result, HADES avoids many redundant reads and writes in Baseline, and all the checking and bookkeeping overheads shown in the first, third, and fourth rows of Table I.

Validation is the second highest contributor to the Baseline latency. The processor performs conflict detection by re-reading the record versions. Moreover, the processor serializes the locking of the written records with the re-reading of the record versions for reads. In contrast, HADES spends less time in these operations because the BFs perform fast conflict detection. In addition, after a node receives the "Intend-to-commit" message, the node processes both writes and reads at the same time.

Baseline spends time in Commit to update record versions, apply the updates, and unlock records. In contrast HADES spends little time in these operations. Indeed, HADES lacks record versions. Also, HADES offloads multiple operations to the NIC and other hardware: (i) sending the updates to the remote nodes at commit, (ii) making the local updates in the LLC non-speculative, and (iii) unlocking the directory and clearing the BFs.

Figure 11 shows the 95th percentile tail latency of the transactions for the different applications in each of the configurations, normalized to Baseline. We see that the tail latency follows the same relative trends as the mean latency.

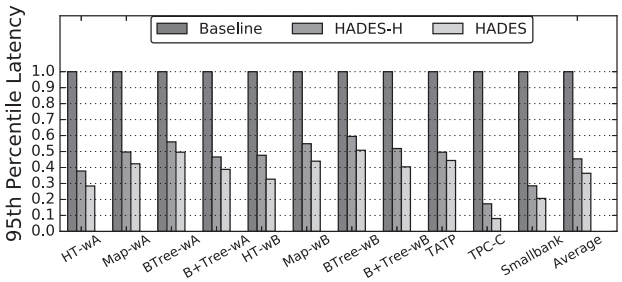

Fig. 11: Tail (95th %) latency normalized to Baseline.

C. Characterizing HADES

We characterize two aspects of the HADES hardware. The first one is how frequently do transactions get squashed due to evictions of their modified lines from the LLC. Recall that, because HADES uses Bloom filters, evicting a non-modified line does not cause a squash. For this experiment, we run the applications forcing every request in the transactions to target data in the local node. This setting maximizes the pressure we put on the LLC. In addition, we modify the cache replacement policy to avoid evicting from the cache a line modified by an active transaction if there are lines in the same set that are not speculatively modified. With this setup, we find that, on average, only 0.1% of the executed transactions need to be squashed because of LLC evictions. The percentage of these squashes is the highest in TPC-C, where 0.7% of the transactions are squashed. Note that this is a small percentage, and the impact is negligible when transactions are also accessing remote nodes. Consequently, we conclude that squashes due to cache line evictions are insignificant for our workloads.

The second experiment characterizes false positive conflicts in HADES' Bloom filters. We find that, of all the conflict detection operations in HADES-H and HADES, 0.02% and 0.04% of them, respectively, result in false positive conflicts. These rates are small in part because individual transactions read and write from different nodes and, as a result, use multiple Bloom filters—each of which is lightly used. Specifically, a transaction at most reads 76 cache lines and writes 40 cache lines in our applications, and these lines are spread across the nodes of the system.

To further assess the effectiveness of the Bloom filters, we consider the worst-case scenario where all the requests of a transaction target a single node. This leads to an average false

positive rate of about 2% for a 1-Kbit Bloom filter. We also perform a sensitivity analysis of the false-positive rate of the Bloom filters we used for our evaluation as a function of the number of cache line addresses inserted in the filter. The results are shown in Table IV.

TABLE IV: Sensitivity of the false positive rate of the filter (%) to the number of cache lines inserted in the filter.

Bloom Filter Size	10 lines	20 lines	50 lines	100 lines
1Kbit	0.04%	0.138%	0.877%	3.26%
512bit+4Kbit	0.003%	0.022%	0.093%	0.439%

From the data in this section, we conclude that our Bloom filter designs are an efficient solution for conflict detection. Their false positive rate is very small.

D. Sensitivity Analysis

We perform two sensitivity analyses. First, we examine the sensitivity of HADES and HADES-H to different network latencies. Figure 12a presents the throughput of the different configurations for different round-trip network latencies (1μs, 2μs, and 3μs). The figure shows the throughput averaged across all the applications and normalized to Baseline with a 2μs network. We see that HADES increases its relative speedup as the network latency decreases. This is because the software overheads of Baseline become a more serious bottleneck as the network latency decreases. Our design eliminates these software overheads. Hence, faster networks favor HADES and HADES-H over Baseline even more.

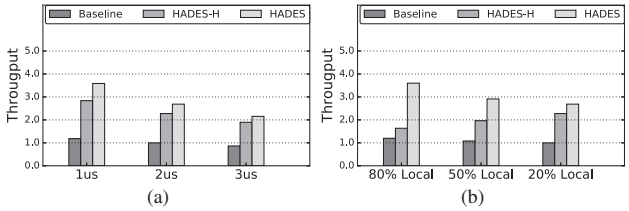

Fig. 12: Throughput for: (a) different network latencies normalized to the 2μs Baseline and (b) different fraction of local requests normalized to 20% local Baseline.

Next, we perform a sensitivity analysis on the fraction of requests in a transaction that target the local node. Figure 12b shows the throughput of the different configurations for different fractions of local requests (80%, 50%, and 20%). The throughput is averaged across all the applications and normalized to Baseline with 20% local requests—which is close to the configuration we used in all the previous experiments. We see that, as the fraction of local requests increases, HADES achieves relatively higher speedups. However, the relative speedups of HADES-H decrease rapidly as we increase the fraction of local requests. This is because HADES-H uses a software-based approach for local operations, which introduces sizable software overheads.

Overall, we conclude that HADES is the best solution across various scenarios, and HADES-H performs relatively better when remote accesses are frequent.

E. Scalability Analysis

To quantitatively assess HADES' scalability, we consider three larger machines (Table III): 10 nodes with 5 cores per node; 5 nodes with 10 cores per node; and 8 nodes with 25 cores per node. In the last two machines, we run multiple applications at a time, to model a space-shared environment.

For the machine with N=10 nodes with C=5 cores per node, Figure 13 shows the throughput for different workloads. Comparing Figure 13 to Figure 9, we see that HADES' speedups over Baseline are similar.

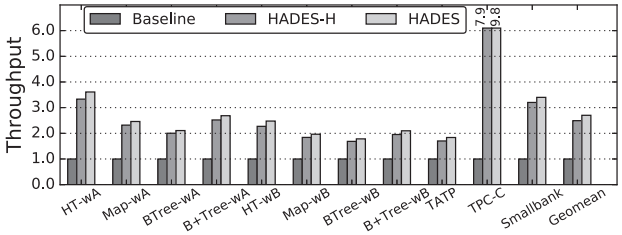

Fig. 13: Throughput normalized to Baseline for N=10, C=5.

In a second experiment, we model N=5 nodes with C=10 cores each. In each node, one workload uses 5 cores and another the other 5 cores. Figure 14 shows the throughput for different mixes of two workloads. Comparing Figure 14 to Figure 9, it can be seen that the resulting mix obtains a throughput that is approximately the average of the two separate workloads. The workloads have relatively small interference because the LLC is fairly large and even same-application threads do not share many lines.

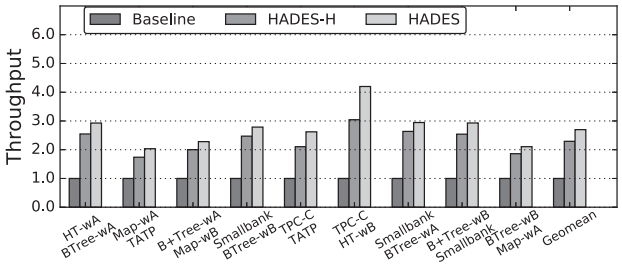

Fig. 14: Throughput of mixes of two workloads normalized to Baseline for N=5 nodes with C=10 cores each.

Finally, we model a cluster with N=8 nodes of C=25 cores each, for a total of 200 cores. We run experiments with different mixes of four workloads from the usual set. Table V shows the mixes used. Figure 15 shows the resulting throughput for each of the mixes and each of the configurations normalized to Baseline. While there is variations across the mixes, we see that HADES delivers the highest throughput. On average across mixes, HADES and HADES-H deliver

2.9× and 2.1× higher throughput, respectively, than Baseline. Overall, we conclude that HADES scales to large machines.

TABLE V: Mixes of workloads used in Figure 15.

mix1	HT-wA, BTree-wA, Map-wA, TATP
mix2	Map-wA, TATP, B+Tree-wB, Map-wB
mix3	B+Tree-wA, Map-wB, Smallbank, BTree-wB
mix4	Smallbank, BTree-wB, TPC-C, TATP
mix5	TPC-C, HT-wB, Smallbank, BTree-wA
mix6	B+Tree-wB, Smallbank, TPC-C, TATP
mix7	TPC-C, TATP, BTree-wB, Map-wA
mix8	BTree-wB, Map-wA, HT-wA, BTree-wA

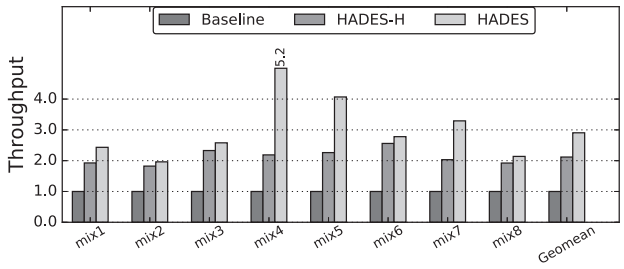

Fig. 15: Throughput of mixes of four workloads normalized to Baseline for N=8 nodes with C=25 cores each.

IX. Related Work

Hardware Transactional Memory (HTM). HADES builds on and uses ideas from the abundant HTM literature [30], [31]. Specifically, HADES uses Bloom filters for conflict detection, which have been used in HTM designs (e.g., [10], [59], [75]). Garzaran et al. [25] provided a taxonomy of the different organizations of speculative buffers, which has inspired HADES' buffering of speculative state. HADES buffers the speculative state in the LLC rather than in the L1; past HTM work has also stored the speculative state in the LLC (e.g., the work by Joshi et al. [32]) or even allowed the speculative state to overflow into main memory (e.g., Kiln [78] or DudeTM [43]). Many of these works extend the cache tags with additional state, like HADES—e.g., Kiln [78] extends the tags for the use of NVM.

The main novelty of HADES over past HTM work is that HADES is the first design for hardware-only transactions in a distributed system. HADES is also the first to utilize the NIC for remote conflict detection. Further, HADES allows RDMA operations within a hardware transaction without aborting, with the use of Bloom filters in the cores and NIC. Two related designs are DrTM [72] and DrTM+R [12]. DrTM first locks all the remote records that a transaction will use and fetches them locally. Then, it uses HTM locally to execute the transaction atomically. Instead, HADES uses OCC to execute the transaction with remote data and needs no a-priori knowledge of the records accessed in the transaction. DrTM+R extends DrTM to use OCC in software. Its mechanism for conflict detection is like FaRM. In addition, inside the distributed software transaction, it uses HTM to guarantee the atomicity of local reads and writes. Instead, HADES executes both local and remote operations in a hardware transaction.

Software Optimizations for Distributed Transactions. To optimize the performance of distributed transactions, the systems community has developed many software-based systems [18], [21], [22], [33], [73], [74]. An influential design is FaRM [21] and its extension [22], which utilize RDMA primitives to accelerate remote data accesses. Later, Opacity [61] advanced the FaRM implementation by enabling strict serializability for all transactions using global timestamps. However, all these schemes are software-based and are limited by the data access protocols provided by the existing network and memory devices. They can suffer significant overheads.

Distributed Transactional Protocols. Researchers have re-examined distributed transactional protocols by exploring advanced features [12], [33], [35], [50], [64], [70], [71], [77]. DrTM [72] and DrTM+R [12] are discussed above. DrTM+H [71] uses both one-sided and two-sided RDMA operations. FaSST [33] replaces one-sided RDMA with fast RPCs using two-sided unreliable datagrams, based on the observation that packet drops happen extremely rarely in modern RDMA networks. PRISM [9] proposes four new RDMA primitives for distributed systems without modifying the underlying hardware. Different from these works, HADES offloads many of the transactional operations to hardware, minimizing software overhead. HADES also develops three new RDMA operations for efficient distributed transaction execution. Also, unlike these works, HADES focuses on accelerating the protocol of distributed transactions with SmartNICs.

Network Support for Distributed Systems. Some proposed designs exploit the compute capability of network devices to accelerate conventional host-based distributed systems and services [20], [54], such as key-value stores [40], RPC [39], remote storage accesses [36], [41], network functions [53], [65], and distributed file systems [37]. Xenic [60] takes advantage of the SmartNIC to reduce the data lookup overhead for distributed software transactions. Finally, although not related to transactions, SABRes [16] proposes a hardware engine that monitors local coherence traffic to guarantee that remote read operations are atomic.

X. Conclusion

This paper presented *HADES*, a new distributed transactional protocol that uses new Bloom filter-based hardware and SmartNIC support to provide fast distributed transactions. We also proposed a cheaper, hybrid hardware-software implementation of HADES called HADES-H. Compared to a state-of-the-art software-only distributed transactional system, HADES and HADES-H increase the average throughput of distributed transactional workloads by 2.7× and 2.3×, respectively.

Acknowledgments

This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute.

REFERENCES

[1] "Apache Cassandra." https://cassandra.apache.org, 2015, (last accessed on 11/20/2021).

[2] "MySQL." https://mysql.com, 2015, (last accessed on 11/20/2021).

[3] 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.

[4] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The Cost of Serializability on Platforms That Use Snapshot Isolation," in *2008 IEEE 24th International Conference on Data Engineering*, 2008, pp. 576–585.

[5] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3085572

[6] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017. [Online]. Available: https://doi.org/10.1145/3015146

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

[8] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[9] 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

[10] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 227–238.

[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, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[12] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM," in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds. ACM, 2016, pp. 26:1–26:17. [Online]. Available: https://doi.org/10.1145/2901318.2901349

[13] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the Killer Microsecond," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 2018, p. 627–640. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00057

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1277–1288, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454167

[15] 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: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[16] A. Daglis, D. Ustiugov, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot, "SABRes: Atomic Object Reads for in-Memory Rack-Scale Computing," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[17] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration," *Proc. VLDB Endow.*, vol. 4, no. 8, p. 494–505, May 2011. [Online]. Available: https://doi.org/10.14778/2002974.2002977

[18] 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," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 205–220, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294281

[19] A. Dey, A. Fekete, R. Nambiar, and U. Röhm, "YCSB+T: Benchmarking web-scale transactional databases," in *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014, pp. 223–230.

[20] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler, "A RISC-V in-network accelerator for flexible high-performance low-power packet processing," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*, 2021.

[21] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414.

[22] A. Dragojević, 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*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 54–70. [Online]. Available: https://doi.org/10.1145/2815400.2815425

[23] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 301–312. [Online]. Available: https://doi.org/10.1145/1989323.1989356

[24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, S. Banerjee and S. Seshan, Eds. USENIX Association, 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[25] M. Garzaran, M. Prvulovic, J. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering memory state for thread-level speculation in multiprocessors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 191–202.

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

[27] J. Gray, "The Transaction Concept: Virtues and Limitations (Invited Paper)," in *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 1981, pp. 144–154.

[28] P. Grun, "Introduction to infiniband for end users," White Paper, InfiniBand Trade Association, 2010. [Online]. Available: http://www.mellanox.com/pdf/whitepapers/Intro_to_IB_for_End_Users.pdf

[29] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, 1983. [Online]. Available: https://doi.org/10.1145/289.291

[30] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*. Morgan & Claypool, 2010.

[31] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *International Symposium on Computer Architecture (ISCA)*, May 1993.

[32] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: durable hardware transactional memory," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. IEEE Computer Society, 2018, pp. 452–465. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00045

[33] A. Kalia, M. Kaminsky, and D. 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)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 185–201. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[34] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A Fast,

Fault-Tolerant and Linearizable Replication Protocol," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 201–217. [Online]. Available: https://doi.org/10.1145/3373376.3378496

[35] A. Katsarakis, Y. Ma, Z. Tan, A. Bainbridge, M. Balkwill, A. Dragojevic, B. Grot, B. Radunovic, and Y. Zhang, "Zeus: Locality-Aware Distributed Transactions," in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*, ser. EuroSys '21, Online Event, United Kingdom, 2021.

[36] D. Kim, A. 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'18)*, Budapest, Hungary, 2018.

[37] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, Virtual Event, Germany, 2021.

[38] 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

[39] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Virtual, USA, 2021.

[40] 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 (SOSP'17)*, Shanghai, China, 2017.

[41] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. K. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[42] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A High-Performance Programmable NIC for Multi-tenant Networks," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 243–259. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/lin

[43] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building Durable Transactions with Decoupling for Persistent Memory," 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. 329–343. [Online]. Available: https://doi.org/10.1145/3037697.3037714

[44] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

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

[46] Mellanox Technologies, "White paper: Introducing 200G HDR InfiniBand Solutions," Tech. Rep., 2019. [Online]. Available: https://www.mellanox.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf

[47] Mellanox Technologies, "ConnectX-6 VPI Card: 200Gb/s InfiniBand & Ethernet Adapter Card," 2020. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf

[48] Mellanox Technologies, "InfiniBand EDR 100Gb/s Routers," Tech. Rep., 2020. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-edr-ib-router.pdf

[49] Mellanox Technologies, "NVIDIA Mellanox BlueField SmartNIC for Ethernet High Performance Ethernet Network Adapter Cards," 2020. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-bluefield-smart-nic.pdf

[50] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting More Concurrency from Distributed Transactions," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014.

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

[52] W. W. Peterson and D. T. Brown, "Cyclic Codes for Error Detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[53] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A Programming System for NIC-Accelerated Network Applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, USA, 2018.

[54] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous NIC Offloads," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Virtual, USA, 2021.

[55] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The Benefits of General-Purpose on-NIC Memory," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2022, p. 1130–1147. [Online]. Available: https://doi.org/10.1145/3503222.3507711

[56] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, Mar. 2011.

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

[58] A. Ruhela, S. Xu, K. V. Manian, H. Subramoni, and D. K. Panda, "Analyzing and Understanding the Impact of Interconnect Performance on HPC, Big Data, and Deep Learning Applications: A Case Study with InfiniBand EDR and HDR," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 869–878.

[59] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 123–133.

[60] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: SmartNIC-Accelerated Distributed Transactions," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, Virtual Event, Germany, 2021.

[61] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro, "Fast General Distributed Transactions with Opacity," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 433–448. [Online]. Available: https://doi.org/10.1145/3299869.3300069

[62] Simon Neuvonen and Antoni Wolski and Markku Manner and Viho Raatikka, "Telecom Application Transaction Processing Benchmark," http://tatpbenchmark.sourceforge.net/, 2011.

[63] The H-STORE Team. Smallbank Benchmark., https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2013.

[64] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast Distributed Transactions for Partitioned Database Systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA, 2012.

[65] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[66] TPC-C. TPC benchmark C., http://www.tpc.org/tpcc/, 2018.

[67] W. Vogels, "All Things Distributed," https://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html, 2010.

[68] M. Walma, "Pipelined Cyclic Redundancy Check (CRC) Calculation," in *Proceedings of the 16th International Conference on Computer Communications and Networks, IEEE ICCCN 2007, Turtle Bay Resort, Honolulu, Hawaii, USA, August 13-16, 2007*. IEEE, 2007, pp. 365–370. [Online]. Available: https://doi.org/10.1109/ICCCN.2007.4317846

[69] Z. Wang, X. Wang, Z. Qian, B. Ye, and S. Lu, "RDMAvisor: Toward Deploying Scalable and Simple RDMA as a Service in Datacenters," *CoRR*, vol. abs/1802.01870, 2018. [Online]. Available: http://arxiv.org/abs/1802.01870

[70] X. Wei, R. Chen, H. Chen, Z. Wang, Z. Gong, and B. Zang, "Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. USENIX Association, Apr. 2021.

[71] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!" in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 233–251. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/wei

[72] 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'15)*, Monterey, California, 2015.

[73] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining ACID and BASE in a Distributed Database," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014.

[74] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang, "High-Performance ACID via Modular Concurrency Control," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, California, 2015.

[75] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 261–272.

[76] R. Zambre, M. Grodowitz, A. Chandramowlishwaran, and P. Shamis, "Breaking Band: A Breakdown of High-Performance Communication," in *Proceedings of the International Conference on Parallel Processing (ICPP'19)*, Kyoto, Japan, 2019.

[77] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building Consistent Transactions with Inconsistent Replication," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, ser. SOSP '15, Monterey, California, 2015.

[78] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the Performance Gap between Systems with and without Persistence Support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 421–432. [Online]. Available: https://doi.org/10.1145/2540708.2540744