In-Network Support for Transaction Triaging

Theo Jepsen
Stanford University
USA

Alberto Lerner
University of Fribourg
Switzerland

Fernando Pedone
Università della Svizzera italiana
Switzerland

Robert Soulé
Yale University
USA

Philippe Cudré-Mauroux
University of Fribourg
Switzerland

ABSTRACT

We introduce Transaction Triaging, a set of techniques that manipulate streams of transaction requests and responses while they travel to and from a database server. Compared to normal transaction streams, the triaged ones execute faster once they reach the database. The triaging algorithms do not interfere with the transaction execution nor require adherence to any particular concurrency control method, making them easy to port across database systems.

Transaction Triaging leverages recent programmable networking hardware that can perform computations on in-flight data. We evaluate our techniques on an in-memory database system using an actual programmable hardware network switch. Our experimental results show that triaging brings enough performance gains to compensate for almost all networking overheads. In high-overhead network stacks such as UDP/IP, we see throughput improvements from 2.05× to 7.95×. In an RDMA stack, the gains range from 1.08× to 1.90× without introducing significant latency.

PVLDB Reference Format:
Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-Network Support for Transaction Triaging. PVLDB, 14(9): 1626-1639, 2021.
doi:10.14778/3461535.3461551

1 INTRODUCTION

Improving transaction processing performance has long been a critical concern for database systems [24]. There are many techniques for simultaneously handling sets of concurrent transactions [4] and for executing them efficiently [61]. These techniques focus on the transactions after they reach the database server. However, a portion of a transaction's lifetime is spent on networking: sending requests to the database server and shipping the results back. For in-memory databases, where transactions are processed without I/O delays, we show that such networking overhead can represent up to 70% of the user-perceived response time for typical workloads.

The overhead occurs because the network has been agnostic about how a database processes transaction requests and responses. For instance, the network interrupts the database whenever a new transaction arrives. As the transaction rate goes up, handling frequent interrrupts becomes inefficient. Moreover, an interrupt stops one random core running the database, and not necessarily the core that should process that particular transaction. Moving the transaction to the desired core wastes one expensive context switch.

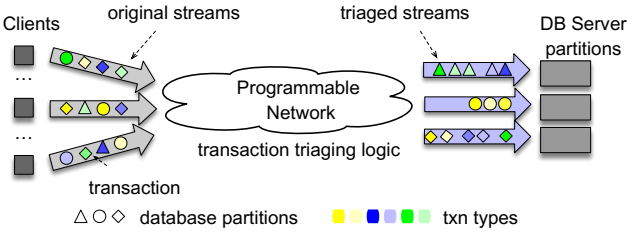


Figure 1: Transaction Triaging uses a programmable network to rearrange transaction streams mid-flight. Triaged streams incur less networking overhead and can be executed with fewer server resources.

The inefficiencies do not stop at transaction delivery. There is also overhead when the database responds to a transaction. As each response is directed to a different client, the database will prepare transaction responses and deliver them to the network one at a time. These inefficiencies add up, especially for small transactions such as the ones found in OLTP workloads.

Technologies such as RDMA [30, 31] can reduce the overhead, but not all networks have RDMA-capable hardware, especially on the client-end of the transactions. Moreover, we show that the overhead does not simply disappear by using RDMA. Recent advances notwithstanding, many still consider the networking overhead to be an unavoidable "tax" on transaction processing.

The problem has never been a lack of networking power. The switches performing the data transfers can parse and make routing decisions for a handful of billions of packets per second. The problem is that such computing power has not been harnessed for application use—e.g., to perform some task on its behalf—because switches could not simply be re-programmed. Their use of closed hardware made any modifications on how a switch works internally require a new fabrication cycle.

Recently, however, a new generation of commercial switching silicon became programmable [11, 74, 80]. They turn a switch's data plane (the component that forwards packets) into a programmable computer, albeit with a very restrictive computing model. Developers quickly realized they could run application logic inside the network beyond strictly networking protocols, in what is called In-Network Computing (INC) [63]. INC has already benefited databases in areas such as query execution [7, 41, 73], replication [42, 85], and caching [34], to name a few applications.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461551

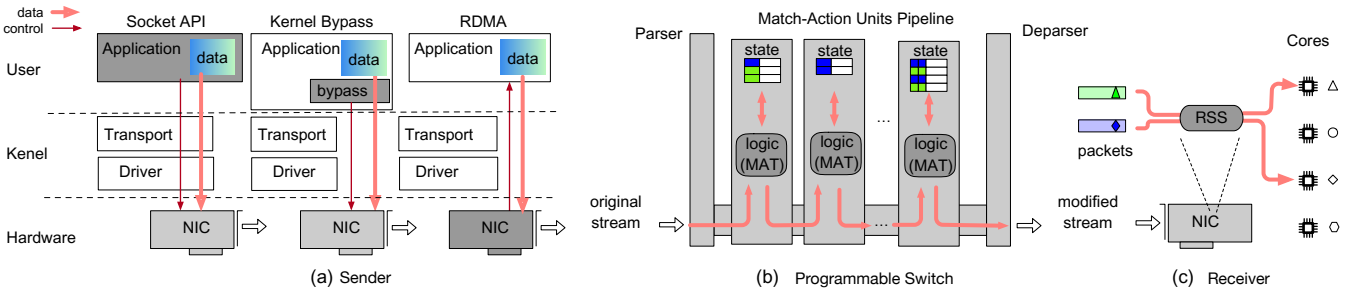


Figure 2: Main components of our end-to-end pipeline comprising (a) different types of network API; (b) PISA data plane; and (c) Receive-Side Scaling (RSS) NIC offload. The dark grey areas denote the critical components the pipeline leverages.

In this paper, we show that INC can also address the network overhead that transaction processing incurs. We propose several techniques that teach a networking device to recognize transaction requests and response data, and to reorganize them in a way to foster database performance, as Figure 1 depicts. Our techniques do not require any changes to existing networking protocols and are designed to be orthogonal to them.

We call these techniques Transaction Triaging (TT). One of the techniques can batch (coalesce) many transaction request packets into one. The net effect is that the database is interrupted only once for the entire batch, amortizing the overhead across many transactions. This TT technique can also batch responses. The server can coalesce responses to distinct transactions, and send them together as a batch. The network then recognizes these response sets, breaks them into individual responses, and delivers them to each corresponding destination.

Another technique selects which core to interrupt for the (now batched) transaction packet. Data partitioning plays an important role here. If a batch contains only transactions for the same partition, the core handling that partition is best suited to handle the interrupt. Therefore, TT techniques are also concerned about which transactions to batch together, and even the order in which they should be delivered. We discuss these and other techniques and show that, together, they can carefully craft the transaction stream to and from the database.

We validate our techniques using a high-performance in-memory database [78] and an actual programmable switch. The results show that TT can improve workloads such as TPC-C [13] and YCSB [12], both on an IP-based stack and an RDMA-enabled one. For instance, in one experiment with a regular IP-based stack, we run a TPC-C workload and achieve 182 Ktps. This number goes up to 373 Ktps, a 2.05× improvement, by applying TT techniques. To put things in perspective, if we preload the transactions on the server, i.e., artificially eliminating the cost of networking, we achieve 386 Ktps. This means that TT techniques leverage the network's computing power and compensate for 97% of the networking overhead. For RDMA-enabled networks, which are already low-overhead, triaging can increase the transaction rate by up to 1.9× by lowering the CPU work necessary to handle requests at very high rates.

Creating TT techniques is challenging in at least two ways. It requires identifying effective and portable transaction manipulations that influence the server's performance. It also requires encoding the techniques as algorithms that programmable networks can support. As mentioned above, the switch imposes a very peculiar computing model, called *feed-forward* [67]. For instance, the model does not support some regular constructs such as deep branches or iterations; some algorithms do not easily translate into this model.

In summary, this paper makes the following contributions:

- It establishes Transaction Triaging as a new source of performance improvement in transaction processing.
- It presents a set of effective transaction triaging techniques that leverage programmable networks.
- It offers feed-forward formulations of these algorithms.
- It discusses how to incorporate TT into existing systems.
- It presents experimental results showing that TT brings significant improvements across diverse scenarios.

The rest of this paper is structured as follows. Section 2 provides background on the networking concepts we use. Section 3 brings an overview of in-network transaction triaging opportunities. Section 4 presents forward-logic formulations for our TT techniques. Section 5 discusses how to generalize algorithms to benefit arbitrary workloads. Section 6 details the changes that are required from an existing system to benefit from TT. Section 7 reports our experimental findings. Section 8 discusses the state of the art in our context, while Section 9 concludes the paper.

2 BACKGROUND

In this section, we introduce the components upon which Transaction Triaging stands. We take advantage of the flexibility of several networking devices, including programmable hardware switches. We describe each component of our solution in turn.

Alternative Network APIs. Sockets have been the most common interface with the network [70]. They rely on the OS for: (a) copying data in and out of the kernel via blocking systems calls; and (b) executing network protocol logic. These tasks consume a significant amounts of CPU and scale poorly with respect to networking speed [21]. As servers became multi-core machines, the need for alternative interfaces became apparent [25, 55].

Kernel-bypass methods offer such alternatives. A bypass model that gained wide acceptance is called the Data Plane Development Kit (DPDK) [16]. It gives an application direct access to a network card , making it responsible for handling all the networking protocol logic. In return, the application can transfer data without incurring intermediate copies or issuing system calls.

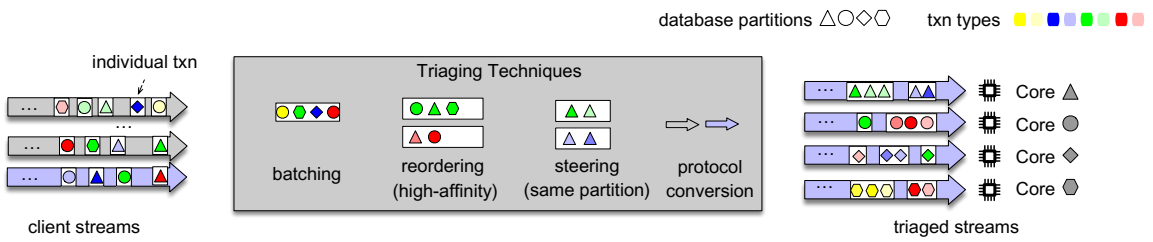


Figure 3: Conceptual view of our triaging techniques. Transactions with the same shape target the same partition. Transactions with a similar color execute in sequence will likely improve performance. The transactions arrive randomly at the switch and are triaged to generate batches of high-affinity, reordered transactions, delivered to a chosen core on the server using the fastest protocol available.

Another bypass method is called Remote Direct Memory Access (RDMA). It is a different protocol stack than TCP/IP altogether although one variant can run on an Ethernet network [30]. RDMA replaces the socket interface with an API called Infiniband "verbs" [17]. The verbs allow an application to point to the data it wishes to transmit. The NIC's hardware peforms all the work necessary to move data in and out of the network.

We compare the three APIs in Figure 2a, showing that a different component can take responsibility for the transmission in each of the stacks. Our techniques work with all three types of APIs.

Programmable networks. This term loosely refers to the flexibility that networking equipment offers to adapt to different scenarios. For instance, programmability can refer to the ability to configure and operate network equipment via Software-Defined Networks (SDN) [20, 40]. Simply put, SDN is an architecture in which switches are divided into two layers: a *control plane* that defines the networking policies (e.g., routing tables), and a *data plane* that forwards packets accordingly. The control plane talks to the data plane via standardized protocols [47].

Network programmability can also refer to data planes that can execute software [5]. Programmable data planes (PDP) appear in a new generation of chips (ASICs) for switches [11, 74, 80] and on FPGA-based NICs [1, 86]. They each implement slightly different programming models that try to balance expressive power vs. execution speed. The Protocol Independent Switch Architecture (PISA) [67] is arguably the most widely adopted PDP model. PISA provides a generic high-speed packet processing pipeline based on a sequence of stages called *Match-Action Units* (MAUs), as Figure 2b shows. Each stage processes one packet at a time by executing one or more *Match-Action Tables* (MATs) in parallel. We describe MATs in more detail in Section 4, but it suffices for now to note that packets flow in only one direction in the pipeline of MATs, giving the programming model its *feed-forward* characteristic.

A PISA device can be programmed to accumulate state related to packets (e.g., counters, histograms, or even payload portions). There are, however, numerous limitations [23]. For instance, variables (application state) are placed in different MATs and can only be accessed by instructions that run in that MAT–like in a *shared-nothing* machine. The instructions themselves are limited in terms of the number of cycles they may take. There are several hardware implementations of PISA pipelines. The switch we use in this work is based on Reconfigurable Match-Action Table (RMT) [9].

NIC protocol offload. The packet handoff between the network and an application is a sophisticated process mediated by the NIC and the OS. Modern NICs can perform several operations in hardware on behalf of the OS [48]. Of particular interest is a mechanism called Receive-Side Scaling (RSS) [46]. When a multi-core server receives a packet, the NIC generates an interrupt against a random core. In a fast network, this task can easily overwhelm one (or more) cores. RSS addresses the problem by distributing the interrupts randomly across the cores, as Figure 2c depicts. The NIC decides which core to interrupt based on a hash over some of the packet's fields. In Section 3 we describe how to couple a programmable data plane with a NIC to perform *semantic* RSS. This form of RSS can deliver the packet to the core that is more likely to process it.

3 TRANSACTION TRIAGING

Triaging aims to produce streams of transactions carefully designed to improve server efficiency. We identify four basic triaging techniques: *batching*, *re-ordering*, *steering*, and *protocol conversion*. Figure 3 illustrates how the individual techniques interact.

The batching technique bundles several single-packet transaction requests into a larger network packet. Batching amortizes the packet-receiving overhead across multiple transaction requests.

Transaction re-ordering manipulates the sequence in which the transactions arrive at the server. It attempts to place similar transactions close to one another using an *affinity* metric. For example, transactions with similar access patterns may reuse instructions or data caches [3, 77].

The steering technique influences how a multi-core database interacts with the network. The network may elect a single core to receive an incoming packet or use RSS to load balance the related overhead across the cores. In-memory databases often choose to partition the data horizontally and map partitions to cores (e.g., partitioning TPC-C by `warehouseID`) [53, 71, 72]. RSS is bound to deliver transactions to a "wrong" partition if it does not take that mapping into consideration. Our steering technique allows RSS to use partitioning information to guide the NIC's RSS algorithm.

Lastly, protocol conversion translates between networking protocols so as to use the most efficient network stack available between the switch and the server. Quite often, a top-of-rack switch has a fast connection with the servers in that rack. We assume this is the case and allow the switch to use techniques such as RDMA in the last hop of the communication, even if the client-to-switch portion of that stream cannot benefit from RDMA.

Combining the four techniques produces the following result: our pipeline generates streams that carry multi-transaction packets, each containing transactions for a given database partition/core, while all transactions within a batch have a high degree of affinity. The batch is delivered directly to the core most involved with the execution, and the delivery utilizes low overhead protocols. The set of triaging techniques we present is portable across database systems, complement each other, and can be expressed together in the programming model we use. As we discuss next, conformance to the model is key to leveraging in-network computing.

4 IN-NETWORK ALGORITHMS

The TT techniques described in the previous section can be implemented on an `x86` CPU using standard data structures and programming languages. However, our goal is to execute such logic on high-speed network devices. We must create algorithms that respect the logical constraints of the computing model and the physical limitations of the current generation of such equipment.

There are at least two design challenges that need to be addressed in today's networking environment. First, we must divide our algorithms' logic into a control plane component—which does not need to see every individual packet—and a data plane component—which does. An example of a control plane task is the initialization and maintenance of TT's *control tables*. We talk about them in detail in Section 6. The core of the TT logic runs on the data plane, manipulating packets while they are in transit.

The second design challenge is to express the data plane component in the *feed-forward* style the switch imposes. This model enforces strict restrictions designed to prevent any pipeline stalls in the data plane. Each stage can perform a limited number of steps, and all logic must fit into the switch's fixed number of stages. By design, there are no loops because network devices are designed to process packets in a single pass. However, iteration can be achieved by re-circulating a packet through the pipeline, at the cost of reducing the throughput of the device.

We introduce a simple example of a feed-forward logic next and describe our algorithms in detail afterward.

Switch Programs. A data plane program running on the switch consists of a sequence of MATs that each packet traverses. Each MAT encodes a lookup on a hash-table-like structure and a corresponding action (side-effect) in case of a match. The action may alter the packet, its metadata, or the match-action table itself. Figure 4 shows how a sequence of MATs modifies a packet carrying a transaction. This program is a preamble to our batching algorithm, which would proceed to insert the transaction the packet is carrying into the designated batch (queue).

Algorithm Notation. We use pseudo-code that captures common feed-forward restrictions without using the syntax of any particular programming language. The data plane logic is separated into a sequence of stages, indicated by the **Stage:** keyword, where each stage corresponds to one match-action unit in the pipeline. The logic at each stage is triggered when a request packet (to the server) or response one (from the server) is received, which is indicated by the **upon** ... **do** code block. We use the notation **with** *var* **as lookup** *key* **in table** *id* to indicate that we perform a lookup operation on the table named *id* with a specified *key*. The result

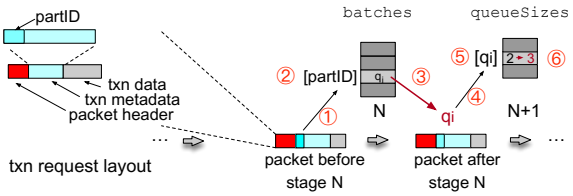


Figure 4: Processing a packet on the switch. (1-2) The program uses a numeric field, `partID`, as an index into the `batches` MAT. That MAT keeps a queue ID, q_i, per entry. (3) The selected queue ID is added to the packet's metadata. (4-5) The program then starts a new match-action cycle by using q_i as an index into the `queueSizes` MAT. (6) The selected entry's counter gets incremented from 2 to 3.

Algorithm 1: The steps are numbered as in Figure 4

```
   Stage: N
   Table: batches
1  upon request pkt, metadata m do
2      with row as lookup (pkt.partID) in table batches
3          m.qid ← row.qid
   Stage: N+1
   Table: queueSizes
4  upon request pkt, metadata m do
5      with row as lookup (m.qid) in table queueSizes
6          row.qsize ← row.qsize + 1
```

of a lookup is bound to the variable name *var*, in the lexical scope indicated by the indented text.

As an example, Algorithm 1 encodes the logic depicted by Figure 4. Note how the pseudo-code captures the two tables' placement: the `batches` table resides in stage `N`, and the `queueSizes` in `N+1`. `N` does not need to start at 0. This means that a particular algorithm may be placed in the middle of the pipeline, either because of a dependency (a value produced by a previous algorithm) or because of resource allocation constraints (the previous stages were occupied). Note also that all the stages execute in parallel in a pipelined fashion. Stage `N` could be operating on a packet while stage `N+1` could be handling the packet that the former just processed. The actions in all the stages are designed to take the same time, and a compiler for feed-forward languages verifies this.

We present more detailed algorithms for each of our four TT techniques using this feed-forward notation. The notation allows us to easily translate them into `P4`, arguably the most adopted programming language for in-network computing applications [8]. The notation also translates naturally to other similar languages such as Broadcom's NPL [51], Huawei's POF [69], or Xilinx's PX [10]. Moreover, we note that by using pseudo-code, the algorithms here could be adapted to run on *smart* network cards, e.g., via `P4`→FPGA program synthesis [29]. However, targeting switches guarantees that our algorithms can run on the fastest devices available. Usually, faster switches appear before faster NICs, e.g., 400 Gbps switches are available at the time of this writing, but the fastest NIC only reaches 200 Gbps.

Describing each TT technique separately in feed-forward logic is only part of our solution. We also show that the four techniques can act as an ensemble and can all be executed simultaneously in the current generation of PDPs. Figure 5 shows the layout of a data plane running our techniques together.

4.1 Batching

Conceptually, the logic for creating batches is straightforward. The main idea is to combine transactions from multiple request packets into a single batched packet. However, to implement this in the network, we must address several non-trivial issues. First, we must manage several queues (batches) in a pipeline architecture. Second, we must determine which transactions to place within the same batch. Third, we must keep track of where transaction requests came from, in order to forward the corresponding individual response when splitting the batched response packets.

Algorithm 2 gives an outline of our feed-forward implementation of batching. At a high level, as requests arrive, the transactions are put in one of many possible queues. When a queue is full, it is drained, and the transactions are combined into a single request. There are three main steps in the algorithm: (i) choosing the queue (batch) for a transaction; (ii) updating the state (size) of the chosen queue; and (iii) either enqueueing a transaction or draining the queue when it is full. When the queue is drained, the transactions in the queue are appended to the current packet.

In stage 1, the algorithm picks a queue for the transaction by performing a lookup on table `batches` (line 2). The table contains a mapping from the transaction's partition (`pkt.partID`) into a queue ID (`qid`); this ensures that each batch contains transactions for the same partition and is steered to the appropriate database thread. We discuss in Section 5 how the table `batches` can be generated, potentially considering more information than simply the transaction's partition, and in Section 6, how the table can be updated to reflect dynamic changes in partitioning.

In stage 2, the algorithm loads the current queue size (`m.qsize`) for the chosen queue (line 5). The queue size is then incremented and stored. If the queue size has reached the batch size, then it wraps around to 0.

The packet then passes through the stages 3 to 3 + `BATCH SIZE`, each of which holds one index entry of all queues. The head of the queue is in stage 3, and the tail moves down the pipeline as transactions are enqueued. Depending on whether the queue is full or not, the stages perform different actions. If the queue is not full, the transaction in the packet is stored when the packet reaches the stage corresponding to the tail of the queue (line 14). If the queue is full, the transaction stored in each stage is loaded into the packet (line 16); once the packet reaches the end of the pipeline, it will contain all the transactions from the queue, and the batch is sent. Figure 6 (2-4) depicts the phases a packet goes through while being processed by Algorithm 2.

To guarantee that batches do not stay incomplete indefinitely, the control plane injects special, per-queue timeout requests. Such packets trigger a batch-send in case the batch has not increased for a given time. For simplicity, we omit this mechanism in Algorithm 2.

Splitting. After executing the transactions, the database responds with a batched response packet. This packet contains the result for transactions submitted by different clients. The switch is responsible for splitting this packet into multiple packets, each addressed to the originating client.

Our response splitting algorithm relies on making copies of the packet as it iterates over the transactions within it, as shown in Algorithm 3. Programmable switches usually provide very efficient mechanisms to support such packet copy operations. In stage 1, the algorithm checks whether the packet contains multiple transactions (line 2). If there are indeed multiple transactions, the packet is split into two packets: a copy which contains all but the first transaction; and the original packet, with only the first transaction. The original packet continues to the next stage, while the copy is sent back to the beginning of the pipeline (line 6). Note that recirculating a packet always makes it start again from Stage 0.

Algorithm 2: Batching Transactions

```
   Stage: 1
   Table: batches                          ▷ Maps transactions to queue IDs
1  upon request pkt, metadata m do
2      with row as lookup (pkt.partID) in table batches
3          m.qid ← row.qid
   Stage: 2
   Table: queueSizes                       ▷ Stores the current size of each queue
4  upon request pkt, metadata m do
5      with row as lookup (m.qid) in table queueSizes
6          m.qsize ← row.qsize + 1
7          if m.qsize = BATCH_SIZE then                    ▷ Queue is full
8              row.qsize ← 0
9          else
10             row.qsize ← m.qsize
   Stage: N: 3 ... 3+BATCH_SIZE
   Table: slotN                            ▷ Stores txn at position N-3 in the queue
11 upon request pkt, metadata m do
12     with row as lookup (m.qid) in table slotN
13         if N−2 = m.qsize then                           ▷ Tail of queue
14             row.txn ← get txn from pkt
15         else if m.qsize = BATCH_SIZE then               ▷ Queue full
16             append row.txn to pkt
```

Algorithm 3: Splitting Batched Transaction Responses

```
   Stage: 1
1  upon response pkt do
2      if pkt contains multiple txns then
3          pkt′ ← copy pkt
4          remove first txn from pkt′
5          truncatate all but first txn from pkt
6          send pkt′ to stage 0 ▷ recirculation; must always send to 0
   Stage: 2
   Table: clientAddrs                      ▷ Stores each client's address
7  upon response pkt do
8      with row as lookup (pkt.clientID) in table clientAddrs
9          pkt.dstAddr ← row.addr
```

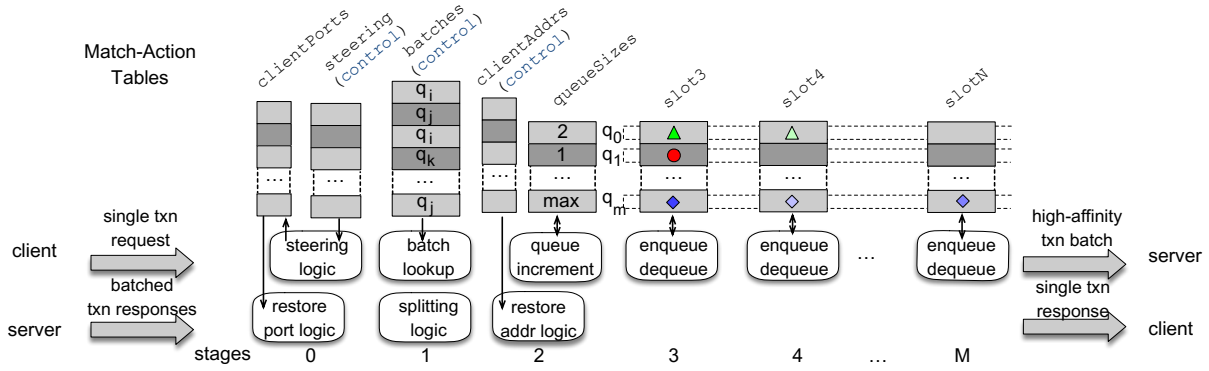


Figure 5: Placement of TT-related match-action tables across the switch's stages. Note that a stage can host more than one MAT, e.g., stages 0 and 2, and that some tables are not updated as a diret result of processing packets, i.e., the control tables.

Stage 2 receives the original packet from stage 1. In this stage, the packet must be addressed to the client that originally sent the transaction. The client's address is looked-up in the `clientAddrs` table (line 8), which contains a mapping of clientIDs to addresses. This mapping can be established during the initialization of each client's connection. For simplicity, we omit such logic and assume that the `clientAddrs` table is static.

Note that this scheme also works for multi-packet responses. We expect each packet to carry metadata about the client so the process above can be applied.

4.2 Reordering

As described above, Algorithm 2 picks a queue for each transaction based solely on the transaction's target (or main) partition. This constitutes a coarse-grained mechanism to reorder transactions. The ordering can be further manipulated in several ways. For example, we can also classify transactions by their type (e.g., NewOrder or Payment in TPC-C), in addition to their target partition. The table `batches` in Algorithm 2, line 2 would then map a packet's `partID, txnType` (instead of just `partID`) into a `qid`, effectively enforcing an *affinity policy*: if certain transaction types should be delivered together to the server, the `batches` table would map them to the same `qid`. Conversely, transaction types that interfere with one another could be assigned to different queues. The number of queues is limited only by the memory available on the switch, but programmable switches can have memory for thousands of such queues (as the batch sizes are usually small).

Having a finer control over the transaction ordering within the same batch is also possible. With slightly different logic in Stages `N` (lines 11 to 16), Algorithm 2 could perform an insertion sort and place a transaction anywhere within the batch. Lines 13 to 14 would instead swap a transaction in the first position when the incoming transaction's priority is higher than the existing one. If a transaction gets displaced this way, it becomes the current transaction—and the insertion process repeats, starting at the stage the displacement occurred. When draining, the algorithm preserves the order in which the transactions appear on a queue, therefore inducing the same order onto the server, upon the receipt of a new batch.

In summary, the algorithms described here are flexible with respect to the order in which transactions are placed within a batch. The layout and contents of the table `batches` and the matches performed by Algorithm 2 can be further tailored to specific cases. We discuss some suitable alternatives in more detail in Section 5.

4.3 Steering

When the server's NIC receives a packet, its RSS algorithm computes a hash over five of the header fields—the source/destination addresses and ports, and the IP protocol—and uses the hash to select a CPU core to interrupt. RSS simply load balances packets across cores (e.g., with N cores, RSS selects core number $hash\%N$).

To steer a packet to a specific database core, we influence the NIC's RSS algorithm by changing the values of some of the five header fields. Note that we cannot change the source and destination address, as that would interfere with routing in the network. Nor can we change the destination port, because the database server is already expecting packets on certain ports. This leaves a single candidate to modify: the UDP source port.

Finding the UDP source port for steering a transaction can be done offline if the client addresses are known upfront or on-the-fly if each new client's address is sent to the switch during the connection handshake between the client and the database server. Finding the port that tips the hash results to the right core is done via an exhaustive search. The search space is limited because the number of CPUs on a modern server is low. For brevity, we omit a formal algorithm description and present the following intuition instead. The algorithm tries UDP port numbers incrementally until the hashing one of them with the other four RSS fields induces the NIC to send the packet to the desired partition/core [39]. This algorithm produces the `steering` table, which maps a client address `srcAddr` and a `partID` to a `srcPort` (for the RSS) and `dstPort` (where the partition/core is). This table can either be used at the client or directly in a network switch.

Algorithm 4 shows how the switch uses the `steering` table to steer a transaction packet to a specific database thread. The logic can fit in a single stage of the switch pipeline. Upon receiving a transaction request packet, the switch first stores the packet's original source port. It does so by looking up the client (identified by `pkt.clientID`) in the `clientPorts` table and updating the row with the port (line 2). The switch loads the ports associated with the partition the transaction wishes to access (lines 4–5). It looks up the

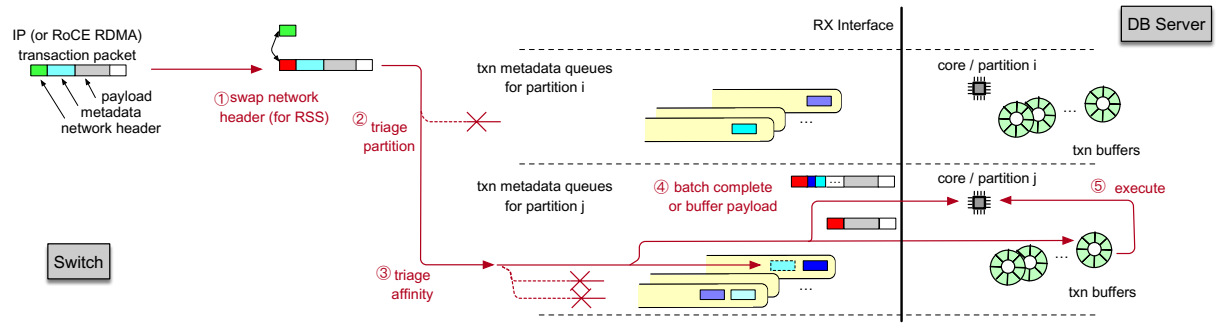


Figure 6: Packet manipulation inside the switch. (1) The network header is adjusted so that RSS can deliver it to the right database partition; (2/3) the transaction metadata determines the desired partition/queue (in yellow) to insert the packet; (4) the payload is sent to a server buffer (in green) if the batch is not full, otherwise the full metadata batch plus the transaction packet is delivered to a designated core; (5) the core accesses the buffered batch and executes the transactions.

Algorithm 4: Steering Transactions and Responses

```
   Stage: 0
   Table: steering          ▷ Stores ports that steer client requests
   Table: clientPorts       ▷ Stores the original srcPort for clients
1  upon request pkt do
2      with row as lookup (pkt.clientID) in table clientPorts
3          row.port ← pkt.srcPort
4      with row as lookup (pkt.srcAddr, pkt.partID) in table
         steering
5          pkt.srcPort ← row.srcPort
6  upon response pkt do
7      with row as lookup (pkt.clientID) in table clientPorts
8          pkt.dstPort ← row.port
```

client's address (`pkt.srcAddr`) and the partition (`pkt.partID`). It then substitutes the source port (`srcPort`) in the packet, effectively causing the packet to go to that destination instead. Figure 6 (1) illustrates this substitution.

Upon receiving a response (line 6), the switch must restore the `pkt.srcPort` from the original request packet. Otherwise, the client will receive a packet for an unknown port (the client is not aware of the translation). The switch loads the previously stored source port, `pkt.srcPort`, which is now the destination port of the packet, `pkt.dstPort` (line 8).

4.4 Protocol Conversion

This technique has both a strategic and a practical purpose. The strategic one is to take advantage of RDMA as a low-overhead protocol between the switch and the database server. We can do so even if RDMA is not available for part of the client-server interconnect. The practical purpose is related to the current generation of programmable switches. They can buffer a handful of packet fields: the transaction's ID, type, and originating client. However, buffering the entire transaction payload would require full packet manipulation involving an area of the switch that is not (yet) fully programmable: the traffic manager. There have been attempts to address this issue [65, 67], but we took an alternative approach instead. To overcome what we believe is a temporary limitation, we buffer the transaction directly on the database server via one-side RDMA initiated by the switch.

Figure 6 (4) provides an overview of the two channels we use to send transactions to the server. When the switch receives a transaction, it assigns the transaction to a queue as described above, as well as a memory address in a ring buffer on the server. It stores the transaction metadata, along with the memory address in the queue in switch memory. It then transforms the transaction packet into an RDMA WRITE request and forwards it to the server (Figure 6 (4), bottom packet). The server's NIC receives the WRITE and stores the transaction in the ring buffer at the address specified by the switch. Note that this does not involve the server's CPU.

When a batch is full, the switch pops all the transactions from the queue forming a single packet that includes the memory address of the server's ring buffer. This batched packet reaches the server through an RDMA SEND operation. The database process receives the batched packet, reading the transactions' full details from the ring buffer (Figure 6 (4), top packet).

Our protocol conversion technique has another advantage. It minimizes the changes to the networking subsystem of a database that wishes to integrate Transaction Triaging. The transaction metadata (steered, batched, and reordered) reach the switch via the normal client connection, as we discuss in Section 6. The only necessary change is for the system to read the transaction payload from the designated queues.

5 TRANSACTION AFFINITY

The reordering algorithm described in Section 4.2 can effectively program the switch to carry out the following mapping:

$$partID[, affinity] \rightarrow queueID[, priority]$$

Given a transaction's target partition and optionally an *affinity* value, the mapping determines the queue to which the transaction should be batched and optionally where to position the transaction within the batch. A DBA that incorporates TT in the database can use this scheme to express many different policies.

The simplest policy is to use the transaction type as the affinity criterion and not use priorities. In practice, the table `batches` in Algorithm 2 line 2 would be keyed by `partID` and `txnType`, instead of just the former. This places transactions in the same batch in the

order in which they arrive at the switch if their partition and type are the same.

The number of queues required corresponds then to the product of the number of different transaction types by the number of partitions in which the database is divided. For instance, running TPC-C (5 transaction types) on a 12-core machine would call for 60 queues on the switch. We show how TPC-C can benefit from this technique in Section 7.

In some other scenarios, however, we expect much more elaborate transaction sets and clustering techniques. One way to triage transactions is through micro-benchmarking them offline and identifying opportunities. For instance, STREX [3] and ADDICT [77] are techniques based on static analysis of query plans. They find transaction types that share instruction patterns and can thus benefit from instruction cache reuse if executed together (on the same core, within a short amount of time). The affinity concept used here is *topological similarity*: transactions that access the same tables/indices and share a common portion of the query plan get the same affinity number.

Another technique to obtain a mapping could be exhaustive evaluation. A starting point is the combination of the number of transaction types per size of batch (the size of a batch is an implementation detail that depends on the flavor of the queue used on the switch and structural properties of the switch). This number can be aggressively pruned by reducing the number of different transaction types in a batch. We envision a calibration tool that performs such tests automatically.

6 INTEGRATION WITH EXISTING SYSTEMS

Integrating TT into an existing database system requires a few changes to the database's control and data paths. On the control path, the database must perform three additional tasks, as Figure 7 depicts. At initialization, it has to fill the `batches` table with the partition and affinity TT policies it wishes the switch to apply. It has to inform the switch about clients via the `steering` and `clientAddrs` tables, either in bulk or on a per-connection basis. Lastly, the database can change the TT policies during runtime by changing the above tables.

The changes to the data path are mainly related to the networking subsystem. These changes involve creating a staging area for transaction requests, as Figure 6 depicted, and adapting to the new packet formats. The impact falls mainly in three areas.

Transaction buffer. The server continues to read transaction request packets as before, although the latter will only contain the transactions' metadata. To receive the transactions' contents, the server must create a transaction buffer. The switch is responsible for delivering transaction payloads into that area. The server informs the switch of the location of this area at initialization time. When the server receives a metadata batch to execute, it fetches the corresponding payloads from the buffer.

Reliability. The switch performs triaging on transactions carried by UDP/IP or RDMA UD. These protocols are easier to manipulate in part because they lack reliability. To account for packet loss, we assume that a client would re-send a transaction request if it does not receive a response within a certain time. The client would also

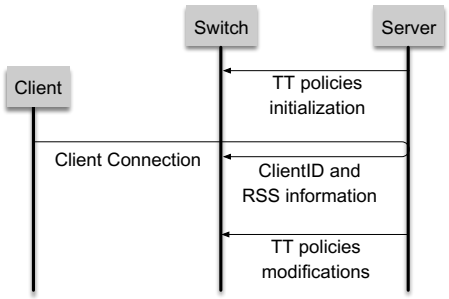


Figure 7: Control interactions: the database server initializes the `batches` table at startup, the `ClientAddrs` and `steering` ones at initialization or at every client connection. The database can change batching policies at any time.

periodically inform the server of the most recent response it has received.

The server maintains a transaction response cache. If it received a transaction request whose response is in the cache, it re-sends those results rather than re-execute the transaction. The cache is cleared using the clients' acknowledgement messages or after a timeout. This scheme assumes that both clients and transactions can be uniquely identified.

Packet format. We assume that network packets have a standard transaction metadata header that allows the switch to manipulate arbitrary transactions uniformly. The header carries metadata information about the client and the transaction. An example of metadata is the partition to which a transaction is directed. Another example is the affinity criteria, if any, used for TT. As we mentioned before, a common affinity criteria is the transaction type. The client and the server may negotiate some parameters at connection time, such as the database partitioning criteria. We assume that a transaction is an instance of a *stored procedure* initiated by an OLTP application [36]. Moreover, the application can fill all the transaction's input parameters at the same time, when issuing the transaction's execution request. The database client's library is responsible for filling the transaction metadata fields.

7 EVALUATION

To evaluate Transaction Triaging, we carried out five sets of experiments. The first set establishes a baseline comparison by quantifying the overhead attributed to the network (Section 7.1). The second set evaluates the impact of each optimization in isolation under various system parameters (Sections 7.2, 7.3, and 7.5). The third set evaluates the optimizations using different network transport layers (Section 7.4). The fourth set compares the performance of the TT techniques under different workloads (Section 7.6). Lastly, we evaluate the benefits of running TT in network hardware, compared to a software implementation on the server (Section 7.7).

Experimental Setup and Environment. As a representative in-memory transactional database, we used Silo [78]. Silo is open-source and capable of executing hundreds of thousands to millions of transactions per second. For this reason, Silo is often used as a benchmark for new concurrency control algorithms [19, 50, 82].

We extended Silo with a network component for each of the protocol stacks we use, as the open-source version does not support networking. We refer to this extended version of Silo as `NetSilo`.

We ran all experiments on servers connected to a 32-port 100 Gbps programmable switch based on the Tofino ASIC [74]. We programmed the switch with our TT techniques using the P4 language. One server acted as the database server, the others as multiple clients. The servers had dual-socket Intel Xeon E5-2603v3 CPUs @ 1.6GHz with a total of 12 cores and 16GB of 1600MHz DDR4 memory (32GB for the database server). Each server had both an Intel 82599ES 10 Gb/s (DPDK compatible) and a Mellanox ConnectX-5 100 Gb/s (RDMA) Ethernet controller.

7.1 Networking Overhead

Transmitting and handling transaction requests accounts for a portion of the perceived transaction response time. As we mentioned, this overhead can be substantial. To quantify this impact, we pregenerated entire workloads made of synthetic transactions and placed them in memory on our Silo server (`Silo`). By reading the transaction requests straight from memory, we eliminate the work of generating and sending them to the server. For each workload, we use a single transaction type that is parameterized to read a certain number of rows from the database. We tested workloads that access either 10, 100, or 1000 rows per transaction. We then compared the time to execute the workload locally versus sending the same workload through remote clients (`NetSilo`). We attribute the difference in performance to network-related overhead. For this experiment, we used a UDP/IP stack.

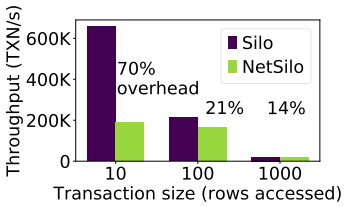
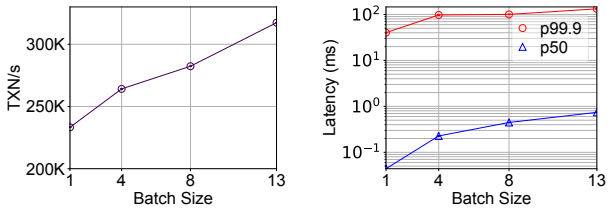


Figure 8: Overhead of network communication on in-memory database.

Figure 8 shows the throughput of the locally- and remotely-generated workload executions. With networking, executing relatively small transactions adds about 70% overhead compared to executing the transaction locally. As the transaction size increases, the overhead decreases, which suggests that the absolute per-transaction overhead is constant. To put things into perspective, when running a local vs. a remote TPC-C workload, we obtained a 53% overhead, as we discuss in Section 7.4. The percentage was much higher for YCSB. These numbers indicate that OLTP workloads tend to present high networking overhead.

7.2 Batching Experiments

To test the idea that the batch size affects throughput, we configured the switch to send increasingly larger batches. Our implementation holds as many as 12 transactions' metadata on the switch and unloads them at the 13^{th} packet. Figure 9a shows the throughput for increasing batch sizes with the TPC-C benchmark.



(a) **(b)**

Figure 9: (a) Throughput and (b) latency due to batching.

Batching can improve the throughput by as much as 36%. This is expected; as the batch size increases, the networking overhead is amortized across a larger number of transactions. There are diminishing benefits for larger batch sizes. As the batch size increases, the overhead of packet processing becomes smaller, relative to transaction execution.

Creating batches on the switch requires queuing transactions as they arrive. As expected, batching increases the average latency, as Figure 9b shows for the 50^{th} and 99.9^{th} percentiles. At the 50^{th} percentile, batching increases the average latency from 0.044 ms without batching, to 0.741 ms with the full batch size. The effect is more attenuated at the 99.9^{th}, from 40 ms to 133 ms. We assume in the experiments that batches do not need to time out. In practice, the switch control plane would send regular packets into the switch to unload partial batches that reach a given latency threshold.

7.3 Steering Experiments

To evaluate how steering contributes to performance, we run `NetSilo` in three modes: with RSS disabled, with standard RSS, and with our semantic RSS. In each mode, a different core (or set thereof) receives an interrupt from the NIC to indicate that a new packet arrived. Without RSS, a single core is interrupted. With RSS, the cores are selected via hashing some fields of the packet's IP headers. Semantic RSS delivers each packet to the primary database partition to which the transaction refers.

Figure 10 shows the effect of varying the number of cores on transaction throughput. The results indicate that RSS becomes necessary when 6 or more cores are used. At 12 cores, there is a substantial improvement in terms of throughput, from 132 Ktps without RSS to 182 Ktps with normal RSS. The difference reflects how using several cores to spread the interrupt load is more efficient. The figure also shows that using semantic RSS can further improve the throughput to 201 Ktps.

Curiously, semantic RSS does not help until the core count increases, as Figure 10 shows. The reason is that Silo's performance scales almost linearly with the number of CPUs. For instance, at 2 CPUs, Silo can process roughly half the transaction rate it can at 4 CPUs, and so on. The experiment shows that at low transaction rates, the CPU cores are not busy enough that they cannot handle the level of interrupts generated by the network. As we increase the number of cores allocated to Silo, the transaction rate increases, and so does the network traffic. Beginning at 6 cores, the interrupt level generated by that traffic starts seeing benefits from RSS techniques.

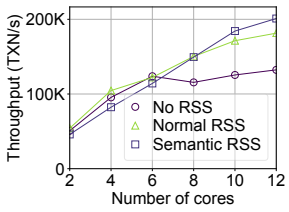
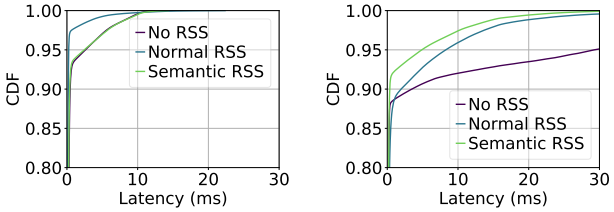


Figure 10: RSS throughput increases with the number of cores.

With respect to latency, we see similar results. In Figure 11a we show that with 6 cores, using normal RSS instead of semantic RSS may be more advantageous. Without RSS, there is some increase in latency due to context switches. In Figure 11b we show that semantic RSS benefits become more pronounced with 12 cores. At higher transaction rates, saving context switches leaves more resources available to actual transaction execution.



(a) (b)

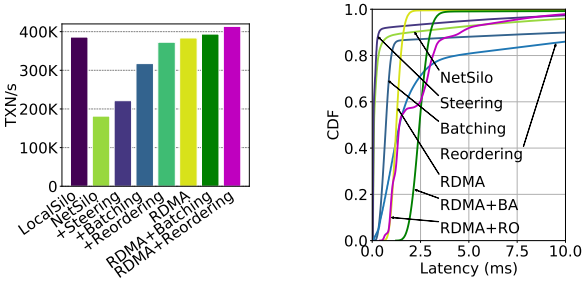
Figure 11: RSS latency CDF (a) for 6 cores, and (b) for 12 cores.

7.4 Comparing UDP/IP and RDMA stacks

To understand the techniques' compound benefits, we run the TPC-C workload adding one TT technique at a time. As discussed before, our techniques are based on a UDP stack in order to allow the switch to manipulate the number and order of the network packets. However, we are also interested in evaluating whether a low-overhead stack such as RDMA could benefit from TT.

Figure 12a compares the throughput improvements on both network stacks. We use `LocalSilo`, i.e., the non-networked, pre-generated workload loaded into the server's memory, as a baseline. It achieves 386 Ktps. We then start adding TT techniques, one at a time, to both network stacks. The simple UDP stack (`NetSilo`) reaches 182 Ktps. Incidentally, this represents a networking overhead of 53% when compared with the baseline, `LocalSilo`. By the time we are running all TT techniques, the throughput increases to 373 Ktps, which represents a 2.05× improvement over `NetSilo`–and 97% of `LocalSilo`. Hence, our triaging techniques almost entirely compensate for the network overhead on a UDP/IP stack.

We implemented a different networking module for `NetSilo` that uses two-sided RDMA SEND over Unreliable Datagrams (UD). We allocate one QueuePair (QP) per core. This setting best approximates the implementation decisions we took for the UDP/IP stack. We test the RDMA stack, which reaches 383 Ktps, as Figure 12a shows. The numbers reflect how efficient that stack already is; the networking overhead amounts to less than 1%. Because RDMA is



(a) (b)

Figure 12: TPC-C throughput (a) and latency (b) at 80% of the maximum throughput.

so efficient, Silo is the performance bottleneck in this scenario as opposed to the network.

We add triaging techniques to the RDMA stack, one by one as before. Note that RDMA QueuePairs already provide the equivalent mechanism of steering, so we skip that technique. The final throughput is 414 Ktps. To put things into perspective, our techniques bring the performance of a UDP stack quite close to an RDMA one. Nevertheless, they improve RDMA in this scenario by 8% only. In other scenarios, transaction reordering can bring much better improvements. We present one such scenario in Section 7.5, but we first discuss the latency implications of our TT techniques.

Figure 12b shows the latency CDF curves for each of the scenarios above. Steering improves on the `NetSilo` baseline because it reduces the overhead of delivering the transactions. All the other techniques impact latency to some extent. This is expected, as we buffer transactions on the switch. As discussed before, reordering is the technique that changes latency the most. The reason is that depending on the affinity criterion, some transaction types may sit in incomplete batches longer–with a high variance, depending on their type. This is, once again, the reason why some latency curves have a wider distribution.

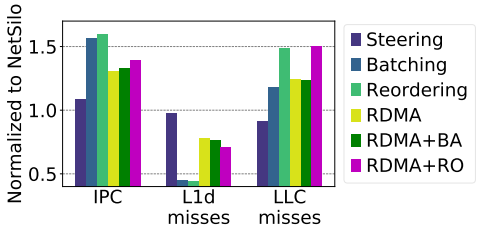


Figure 13: TPC-C CPU micro-architecture analysis.

Figure 13 shows micro-architecture measurements for the scenarios above. Each technique improves the instructions per clock (IPC) value of the previous technique and lowers the rate of L1 cache misses. This is expected: with fewer context switches (e.g., due to steering) it is more likely for the instruction and data caches to have "hot" instructions/data. Context switches would have flushed these caches. We see, however, an increase in LLC misses. This is not uncommon: batching/reordering of transactions makes it so that more data is touched per unit of time, hence yields more last layer cache misses [66].

7.5 Reordering Experiments

To evaluate the effect of transaction reordering, we used the standard TPC-C workload but partitioned the data differently: we assigned multiple cores to each warehouse. This increases contention because conflicting transactions that would otherwise execute serially in one core can now run concurrently in several cores.

Figure 14(left) shows the transaction throughput with an increasing number of cores per warehouse. We contrast the RDMA baseline execution with two TT techniques: batching (BA) and reordering (RO). With batching, we simply group transactions at random. With reordering, we use `txnType` as the affinity criterion, i.e., batches are made of a single transaction type.

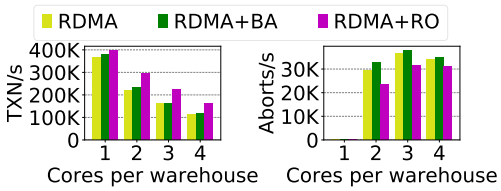


Figure 14: TPC-C workload (left) under varying degrees of contention and (right) number of aborts in each scenario.

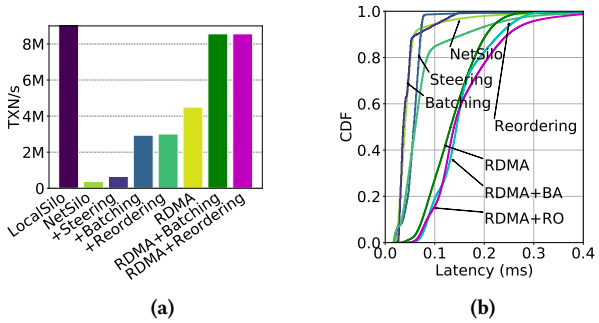
With one core per warehouse, there is minimal contention, so the biggest improvement in performance comes from batching. At four cores per warehouse, the contention is much higher. The baseline RDMA throughput is reduced to 115 Ktps with four cores from 364 Ktps when using just one core. Reordering reduces contention by separating the reads from the writes: while one core is performing read-only transaction types on a snapshot, another core can perform write-heavy transaction types uncontended. This results in a throughput of 162 Ktps with 4 cores, a 1.4× improvement over the baseline RDMA scenario. We observe that this improvement is consistent when increasing the contention level.

The main source of speedup from reordering based on `txnType` comes from reducing contention. This is evident from the abort rate shown in Figure 14(right). With a single core, the abort rate is close to zero. With two cores per warehouse, reordering reduces the abort rate from 30K to 23K aborts/s. This reduces the number of abort responses that have to be sent back to the client, thus increasing the "goodput." Batching, on the other hand, slightly increases the abort rate, because it delivers transactions to the database at an overall higher throughput.

7.6 Comparing TPC-C and YCSB

To evaluate if the benefits of TT extend to different workloads, we repeated the experiments from Section 7.4, this time using the YSCB "A" benchmark with a 80/20 R/W transaction mix. YCSB transactions are notably smaller than TPC-C's in that they read fewer rows and make fewer changes, on average.

Figure 15a shows the throughput obtained by applying our triaging techniques to a YCSB workload. As before, we cumulatively apply one technique at a time. As expected, YSCB transactions are heavily impacted by networking overhead. In our experiments, the throughput of `LocalSilo` is 17.5 Mtps, whereas the UDP/IP `NetSilo` delivers 377 Ktps, a 98% networking overhead. Although the combined techniques over UDP/IP do not match the throughput



(a) (b)

Figure 15: YSCB throughput (a) and latency (b) at 80% of the maximum throughput.

of `LocalSilo`, batching and reordering have the most substantial speedup: they deliver 3 Mtps, an increase in throughput of 7.95× compared to `NetSilo`. This is a markedly better improvement compared to TPC-C benefits. The reason is that the more the networking overhead slows down transaction rates, the more opportunities to recover this overhead when applying TT techniques.

The baseline RDMA further reduces networking overhead, delivering 4.5 Mtps. By using our batching technique, we increase the RDMA throughput by 1.9x to 8.56 Mtps. Since the YCSB has small transactions with little contention, the reordering technique provides little improvement over batching; the main benefit to this workload comes from reducing the time spent handling network requests. Furthermore, we surpass the RDMA rates at a negligible latency cost, as we discuss next.

Figure 15b shows the latency CDF for the various techniques. As before, the impact on the UDP/IP stack is small. We see in this scenario that, as before, our techniques add more latency with the RDMA stack. This reflects how very small transactions can be more sensitive to latency. Note, however, that the x-axis in Figure 15b is measured in tens of milliseconds. With a mere 0.1 ms of additional latency, TT can almost double the RDMA transaction rate.

7.7 Software-Based TT

The question may arise as to whether network hardware is really necessary for executing TT techniques. To evaluate this question, we implemented TT on the RDMA-enabled `NetSilo` server by having it perform reordering using `txnType` as the affinity criterion. We then compared this server-based TT to the baseline RDMA-enabled `Netsilo` and the switch-based TT. Figure 16 shows the throughput and latency of the three scenarios.

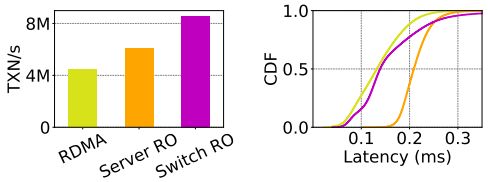


Figure 16: YCSB throughput (a) and latency (b) of different TT/reordering (RO) implementation sites.

Reordering transactions on the server increases the throughput compared to the baseline RDMA from 4.5 to 6.06Mtps, an improvement of 1.35×. Server-based TT optimizes the execution of the transactions to some extent, but it simply cannot alleviate the server from the network processing. Switch based TT delivers 8.56Mtps, an improvement of 1.41× over the server-based TT implementation. Although they have similar tail latencies, the switch-based TT has a lower mean latency.

Applying TT techniques in software *after* the network forgoes several opportunities for optimization. The transaction requests are not batched until after they arrive at the server, with all the networking overhead that this entails. Furthermore, the server-based TT cannot benefit from batching responses. Since the standard network does not provide the functionality for splitting response packets, the server is forced to send transaction responses individually to their respective clients.

8 RELATED WORK

We divide the related work into three categories. First, we consider other transaction management techniques that do not execute in-nework. Second, we examine work that leverages in-network computing for applications other than transaction management. Lastly, we compare systems that also benefit from the low-overhead networking that RDMA technology introduced.

Transaction Management Techniques. Partitioning databases is a common way to optimize transaction management and achieve scalability in multi-core systems [37, 38, 45, 71, 78]. These systems schedule transactions immediately upon arrival and can benefit from having the network deliver transactions to a core respecting the database partitioning.

Steering techniques that go beyond RSS have been proposed in the context of OS support for low-latency transactions [35, 52, 57, 58]. One project even leverages a programmable NIC [60]. These techniques try re-assigning cores to applications when some cores find themselves with more work than others. Our techniques also move computations onto specific cores, but we do so without requiring any changes to the OS.

Transaction batching is a widespread technique that databases use in different execution stages: during transaction execution [53], logging of transactions (group commit) [22, 26], and at replication time [56]. We perform batching in-network, which eliminates the cost of doing so on the database server.

Transaction reordering is also a known technique. Many systems seek to minimize concurrency conflicts in such a way [15, 53, 72]. The schedulers in those systems try to select the next transaction to execute that would cause minimal interference to ongoing transactions. These techniques are complementary to ours. Reordering has also appeared in the context of maximizing resource sharing during execution [3, 77] or replication [54]. We have shown similar improvements, although by resorting to in-network mechanisms instead of consuming server resources that could otherwise be dedicated to processing additional transactions.

In-Network Computing Applications. Some work has shown that engaging the network can also be useful in transaction execution [32, 42] or replication scalability [85]. They try to identify or eliminate conflicts that can cause transaction aborts. Naturally, these techniques are dependent on the kind of concurrency control a given database uses. Our techniques are complementary to those.

Other database subsystems—such as query execution—have also leveraged INC capabilities. For instance, a number of relational operators can be processed in-network, such as selection, join, and aggregation [28, 41, 73], as well as other data analysis operators such as map-reduce [7, 63].

Other applications of in-network computing include aggregation in machine learning contexts [64], caching [34, 76], consensus [14, 43], and some limited version of data streaming applications [2, 33, 68]. We should note that streaming computations [49] and in-network computing bear only some superficial resemblance. The differences in programming models cause algorithms formulated for one model not to translate naturally—or at all—into the second.

RDMA and fast networking. Before the advent of fast networking, databases were designed to make as minimal and optimal use of communication as possible. RDMA networks ultimately revert this trend, as they drastically lowered transmission costs. Moreover, it is no longer sensible to tradeoff CPU cycles to save networking time. CPU performance (in terms of clock speed) is believed to have plateaued [27], whereas network performance is still improving at a fast pace. Network cards that operate at 200 Gbps and switches with 400 Gbps ports are already commercial off-the-shelf equipment. Moreover, the 800 Gbps Ethernet standard has already been ratified, and the 1.6 Tbps one is already being discussed [18].

Many database subsystems have been redesigned to leverage fast networking and kernel bypass in light of these changes. There are works that redesign distributed query execution [6, 44, 59, 62], distributed transaction processing [81, 83], and replication [79, 84].

Our work falls into this category but it leverages network programmability on top of low-overhead, high-speed networking. Moreover, we are in a unique position because we formulate our algorithms in the feed-forward style. This allows us to keep benefiting from faster-speed equipment when it becomes available without changing our algorithms, e.g., 400G programmable switches [75].

9 CONCLUSION

In this paper, we introduced Transaction Triaging, a set of techniques that can be executed in-network and shape the streams of transactions before their delivery to a database server. We showed that performing Transaction Triaging can reverse the network overhead by providing server performance improvements.

As programmable networks become off-the-shelf technology, we see our techniques as one more step towards revisiting the traditional separation of concerns between networking and database systems, allowing a new generation of systems to emerge.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments, and Paolo Costa, Rana Hussein, André Ryser, Yanfang Le, and Daniel Cason for discussions on early drafts of this paper. This work was partly supported by DARPA Contract No. HR001120C0107, the European Research Council (ERC) under the Horizon 2020 Program (grant agreements 683253/Graphint) and the Swiss National Science Foundation (SNSF), projects #175717 and #166132.

REFERENCES

[1] Alveo [n.d.]. ALVEO Adaptable Accelerator Cards for Data Center Workloads. https://www.xilinx.com/content/xilinx/en/products/boards-and-kits/alveo.html.

[2] Arvind Arasu, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 480–491.

[3] Islam Atta, Pinar Tözün, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. 2013. STREX: Boosting Instruction Cache Reuse in OLTP Workloads through Stratified Transaction Execution. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, 273–284. https://doi.org/10.1145/2485922.2485946

[4] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc.

[5] Roberto Bifulco and Gábor Rétvári. 2018. A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 1–7. https://doi.org/10.1109/HPSR.2018.8850761

[6] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[7] Marcel Blöcher, Tobias Ziegler, Carsten Binnig, and Patrick Eugster. 2018. Boosting Scalable Data Analytics with Modern Programmable Networks. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) *(DAMON '18)*. Association for Computing Machinery. https://doi.org/10.1145/3211922.3211923

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

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4, 99–110.

[10] Gordon Brebner and Weirong Jiang. 2014. High-Speed Packet Processing using Reconfigurable Computing. *IEEE Micro* 34, 1 (Jan. 2014), 8–18.

[11] Broadcom Trident 4 [n.d.]. Broadcom Trident 4. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA). ACM, 143–154.

[13] Transaction Processing Performance Council. 2010. TPC-C Benchmark Revision 5.11.0. http://www.tpc.org/tpcc/.

[14] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Santa Clara, California) *(SOSR '15)*. Association for Computing Machinery. https://doi.org/10.1145/2774993.2774999

[15] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 169–182. https://doi.org/10.14778/3282495.3282502

[16] DPDK [n.d.]. Data Plane Developemnt Kit. https://dpdk.org/.

[17] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2017. RDMA Reads: To Use or Not to Use? *IEEE Data Eng. Bull.* 40, 1 (2017), 3–14.

[18] Ethernet [n.d.]. Ethernet Technology Consortium – 800G Specificatin. https://ethernettechnologyconsortium.org/wp-content/uploads/2020/03/800G-Specification_r1.0.pdf.

[19] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201.

[20] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98. https://doi.org/10.1145/2602204.2602219

[21] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev R. Patwardhan, and Greg J. Regnier. 2003. TCP Performance Re-Visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '03)*. IEEE Computer Society, 70–79.

[22] Dieter Gawlick and David Kinkade. 1985. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Eng. Bull.* 8, 2 (1985), 3–10.

[23] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the Stateless Quo of Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*.

[24] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc.

[25] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, 135–148.

[26] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. 1989. Group commit timers and high volume transaction systems. In *High Performance Transaction Systems*, Dieter Gawlick, Mark Haynie, and Andreas Reuter (Eds.). Springer Berlin Heidelberg, 301–329.

[27] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[28] Jaco Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. 2019. High-Performance In-Network Data Processing. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'19)*.

[29] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. 1–9.

[30] Infiniband Architecture Specification Annex A16 [n.d.]. Infiniband Architecture Specification–Annex A16: RoCE. https://www.infinibandta.org/ibta-specifications-download/.

[31] Infiniband Architecture Specifications [n.d.]. Infiniband Architecture Specification. https://www.infinibandta.org/ibta-specifications-download/.

[32] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. 2018. Infinite Resources for Optimistic Concurrency Control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing* (Budapest, Hungary) *(NetCompute '18)*. Association for Computing Machinery, 26–32. https://doi.org/10.1145/3229591.3229597

[33] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the fast lane: A line-rate linear road. In *Proceedings of the 4th ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR'18)*.

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

[35] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[36] Robert Kallman, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[37] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[38] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, 1675–1687. https://doi.org/10.1145/2882903.2882905

[39] Hugo Krawczyk. 1994. LFSR-based Hashing and Authentication. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '94)*. Springer-Verlag, 129–139.

[40] Diego Kreutz, Fernando M. V. Ramos, Paulo E. Veríssimo, Christian E. Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (Jan 2015), 14–76. https://doi.org/10.1109/JPROC.2014.2371999

[41] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *9th Biennial Conference on Innovative Data Systems Research* (Asilomar, California) *(CIDR '19)*.

[42] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, 104–120. https://doi.org/10.1145/3132747.3132751

[43] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[44] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2019. Design and Evaluation of an RDMA-Aware Data Shuffling Operator for Parallel Database Systems. *ACM Trans. Database Syst.* 44, 4 (Dec. 2019). https://doi.org/10.1145/3360900

[45] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (July 2019), 1316–1329. https://doi.org/10.14778/3342263.3342270

[46] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. 2006. Receive Side Coalescing for Accelerating TCP/IP Processing. In *Proceedings of the 13th International Conference on High Performance Computing* (Bangalore, India) *(HiPC'06)*. Springer-Verlag, 289–300. https://doi.org/10.1007/11945918_31

[47] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. 38, 2 (March 2008), 69–74. http://doi.acm.org/10.1145/1355734.1355746

[48] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Lihue, Hawaii) *(HOTOS'03)*. USENIX Association, 5.

[49] S. Muthukrishnan. 2005. *Data Streams: Algorithms and Applications*. Now Publishers Inc. https://ieeexplore.ieee.org/document/8186985

[50] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 511–524. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/narula

[51] Network Programming Language [n.d.]. Network Programming Language. https://nplang.org/.

[52] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[53] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[54] Fernando Pedone, Rachid Guerraoui, and André Schiper. 2003. The Database State Machine Approach. *Distributed Parallel Databases* 14, 1 (2003), 71–98. https://doi.org/10.1023/A:1022887812188

[55] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, 337–350. https://doi.org/10.1145/2168836.2168870

[56] Frank M. Pittelli and Hector Garcia-Molina. 1989. Reliable Scheduling in a TMR Database System. *ACM Trans. Comput. Syst.* 7, 1 (Jan. 1989), 25–60. https://doi.org/10.1145/58564.59294

[57] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, 325–341. https://doi.org/10.1145/3132747.3132780

[58] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[59] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 228–239. https://doi.org/10.14778/2856318.2856319

[60] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019* (Beijing, China) *(APNet '19)*. Association for Computing Machinery, 71–77. https://doi.org/10.1145/3343180.3343184

[61] Mohammad Sadoghi, Spyros Blanas, and H. V. Jagadish. 2019. *Transaction Processing on Modern Hardware*. Morgan & Claypool Publishers.

[62] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking Distributed Query Execution on High-Speed Networks. *IEEE Data Engineering Bulletin* 40, 1 (2017), 27–37.

[63] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (Palo Alto, CA, USA) *(HotNets '17)*. Association for Computing Machinery, 150–156. https://doi.org/10.1145/3152434.3152461

[64] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[65] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 685–699. https://www.usenix.org/conference/nsdi20/presentation/sharma

[66] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-Architectural Analysis of In-Memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, 387–402. https://doi.org/10.1145/2882903.2882916

[67] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

[68] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. ACM, 164–176. https://doi.org/10.1145/3050220.3063772

[69] Haoyu Song. 2013. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. 127–132.

[70] W. Richard Stevens and Thomas Narten. 1990. Unix Network Programming. *SIGCOMM Comput. Commun. Rev.* 20, 2 (April 1990), 8–9. https://doi.org/10.1145/378570.378600

[71] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, Austria, September 23-27, 2007*. ACM, 1150–1160.

[72] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/2213836.2213838

[73] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2019. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos* (Beijing, China) *(SIGCOMM Posters and Demos '19)*. Association for Computing Machinery, 72–74. https://doi.org/10.1145/3342280.3342311

[74] Tofino [n.d.]. Barefoot Tofino. https://www.barefootnetworks.com/technology/.

[75] Tofino 2 [n.d.]. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[76] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. 2018. LaKe: The Power of In-Network Computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2018.8641696

[77] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. 2014. ADDICT: Advanced Instruction Chasing for Transactions. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1893–1904. https://doi.org/10.14778/2733085.2733095

[78] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania). ACM, 18–32.

[79] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 406–419. https://doi.org/10.1145/3186728.3164137

[80] xpliant [n.d.]. XPliant Ethernet Switch Product Family. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[81] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, 1571–1586. https://doi.org/10.1145/3183713.3196890

[82] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, 1629–1642. https://doi.org/10.1145/2882903.2882935

[83] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[84] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1637–1650. https://doi.org/10.14778/3342263.3342639

[85] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 376–389. https://doi.org/10.14778/3368289.3368301

[86] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (Sep. 2014), 32–41. https://doi.org/10.1109/MM.2014.61