AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing

Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu,
Beijing National Research Center for Information Science and Technology (BNRist)

https://www.usenix.org/conference/atc22/presentation/li-junru

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing

Junru Li Youyou Lu* Qing Wang Jiazhen Lin Zhe Yang Jiwu Shu

Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

High-performance transaction processing needs to schedule numerous requests from the network. However, such request scheduling comes with costs of complex information gathering and considerable computation. We observe that emerging SmartNICs pose opportunities for transaction scheduling with low overhead. In this paper, we propose AlNiCo, which leverages SmartNICs to intelligently schedule incoming transaction requests to CPU cores, minimizing inter-transaction contention with low latency. AlNiCo describes the contention according to system states in a way that SmartNICs can efficiently process, and co-designs hardware and software to enable flexible and adaptive scheduling. We implement AlNiCo using FPGA-equipped Innova-2 SmartNICs, and our evaluation shows that AlNiCo boosts the throughput by 1.30× ~ 2.68× and reduces the latency by up to 48.8%.

1 Introduction

Transaction processing is a critical building block for many applications such as e-commerce and the stock exchange. Over the last few years, two trends in transaction processing stand out. First, network bandwidth has improved significantly, enabling a surge of transaction requests from the network. Second, modern servers are equipped with numerous CPU cores, providing abundant computing capacity for transaction processing [1–4]. These two trends together pose a crucial problem: *how to schedule each individual transaction request to the most appropriate CPU core?* Using a sophisticated scheduler for transaction processing can mitigate lots of inter-transaction contention (i.e., two transactions concurrently access the same record, and at least one performs a write), reducing transaction aborts/blocking and thus boosting system throughput.

There are two setbacks in realizing efficient transaction scheduling. First, it is hard to gather necessary information for scheduling in consideration of cost-efficiency. This is because the transaction system is intricate: each request contains a write/read-set with multiple records, and different records have different degrees of hotness [5–7] and affinities. The hotness and affinity are dynamic: for example, in the live selling platform of Kuaishou, product popularity changes over time due to the behavioral uncertainty of users. Second, transaction scheduling incurs considerable computation overhead. For an incoming request, to calculate which CPU core it executes on causes the least contention, the scheduler has to consume computation cycles that are proportional to the number of CPU cores and the request's write/read-set size.

We observe that exploiting emerging SmartNICs [8, 9] can enable efficient transaction scheduling for two reasons. First, every transaction request/response flows through NICs, so they are the natural place to gather information about scheduling. Second, transaction scheduling requires considerable computation, and SmartNICs are equipped with specialized programmable hardware (e.g., FPGAs), which is adept in fine-grained parallelism. Thus, SmartNICs can serve numerous concurrent scheduling tasks with low latency.

It is non-trivial to design transaction scheduling using SmartNICs. First, we need to describe inter-transaction contention in a hardware-friendly manner so as to squeeze out the parallelism capabilities of the specialized hardware in SmartNICs. Second, it is impossible for SmartNICs to perform all scheduling tasks since CPU-side transaction software owns important information about transaction execution, e.g., the abort rate, which is indispensable for scheduling. Thus, to enable effective transaction scheduling, SmartNICs should cooperate with upper-level transaction software.

We propose AlNiCo, a transaction processing system with on-NIC request scheduling. At the core of AlNiCo is a SmartNIC-accelerated contention-aware scheduler; it schedules incoming transaction requests to different CPU cores intelligently, minimizing contention between transactions with low latency.

To describe the inter-transaction contention in a hardware-friendly manner, we first classify system runtime states into

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

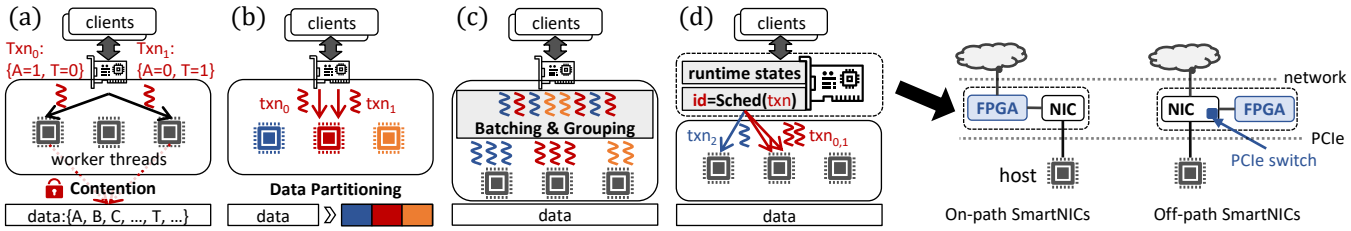

Figure 1: **The architectures of networked transaction systems (a-d).** *(a): A system without scheduling. (b): A system using the static data partitioning method. (c): A system using the batching-based scheduling method. (d): AlNiCo.*

three types: 1) *request state*, i.e., the set of records accessed by a request and the associated access modes (read and write); 2) *worker state*, i.e., the set of requests being executed by the worker thread on each CPU core; 3) *global state*, i.e., workload characteristics such as hotspots. Then, we encode the above three types of states into compact vectors and translate contention calculation into fast vector computation on SmartNICs. For an incoming request, to calculate its contention against different CPU cores, the scheduler compares the request state and each worker's worker state, with the global state as the weight. We optimize the calculation process via fine-grained parallelism, ensuring low scheduling latency.

To enable adaptive scheduling for time-varying applications, AlNiCo adopts a feedback mechanism from the upper-level transaction software to the SmartNICs. Specifically, software in AlNiCo periodically samples the global state as the scheduling guideline and updates the on-NIC scheduler by flexible hardware/software interfaces. In this way, AlNiCo can handle dynamic workloads in which hotspots change over time. In addition, via the generalized feedback mechanism, AlNiCo can support various concurrency control protocols.

We implement AlNiCo with Mellanox Innova-2 [8], a SmartNIC equipped with an FPGA. The evaluation shows that AlNiCo reduces the abort rate by contention-aware transaction scheduling. AlNiCo boosts the throughput by 1.30× ~ 2.68× than the original system with various concurrency control protocols. In addition, compared with existing static data partitioning methods, AlNiCo can handle time-varying and skewed workloads; compared with batching-based scheduling methods, AlNiCo schedules requests in real-time and renders latency two orders of magnitude lower.

In summary, we make the following contributions:

- A SmartNIC-enabled contention-aware transaction scheduler that offers low overhead by exploiting the acceleration provided by the FPGA.
- AlNiCo, a transaction processing system based on the scheduler, supporting various concurrency control protocols and time-varying workloads.
- An AlNiCo prototype on real hardware, exhibiting higher throughput, lower latency, and less abort rate.

2 Background and Motivation

2.1 Contention-aware Scheduling

In an in-memory single-server transaction system, as shown in Figure 1 (a), the NIC receives transaction requests and dispatches them to worker threads. Each transaction request often includes multiple write/read operations to the in-memory records. Worker threads execute transactions using various concurrency control protocols to guarantee transaction isolation. However, concurrent transactions are more likely to be aborted or blocked when there is contention (i.e., two transactions concurrently access the same record, and at least one performs a write). Aborts result in costly retries, and blocking may be cascaded, both of which degrade performance [3]. Therefore, the transaction system needs to minimize contention among concurrent transactions [10–22]. Designing a contention-aware scheduler that schedules conflicting transactions to the same worker threads is meaningful. However, there is an accuracy-overhead trade-off in designing it.

Accuracy-overhead trade-off. A high-accuracy scheduler can assist the majority of requests in selecting an appropriate (i.e., with less contention) worker thread to execute on. To achieve such accuracy, the scheduler incurs much computation overhead. Unlike single-key get/put requests in key-value stores, transaction requests are more sophisticated, containing multiple operations (reads, writes, and range queries) to multiple records. The packet steering method [23, 24] based on RSS [25] or Flow Director [26], which embeds a single key in the packet header as the request dispatching information, is no longer effective for transaction requests. Furthermore, the records have different degrees of hotness (i.e., a small set of records are accessed frequently) and affinities (i.e., two records are often accessed together). The hotness and affinity are also constantly changing. As a result, contention-aware scheduling is time-consuming and induces high overhead for high accuracy.

2.2 Existing Scheduling Methods

We revisit two transaction scheduling methods and show their choices in the accuracy-overhead trade-off.

Static data partitioning method. As shown in Figure 1 (b), recent studies [10–12] maintain a data partition scheme, i.e., a mapping from records to worker threads. Clients know the partition scheme and send transactions directly to worker threads without scheduling. Then, each worker thread executes the transactions that access records in a single partition sequentially. Further, Jepsen et al. [15, 16] use a programmable switch to triage transactions belonging to different data partitions before sending them to the database server.

This method is effective for partitionable workloads where transactions tend to access records in a single partition because it has no scheduling overhead under such workloads. However, this method sacrifices the accuracy for two types of workloads: ❶ workloads that do not have a good partition scheme and ❷ workloads whose record affinities and degrees of hotness are constantly changing. In the first type of workloads, many transactions are cross-partition. These transactions require synchronization between worker threads to access remote partitions. In the second type of workloads, the partition scheme needs to be updated to follow workload changes. Recent studies all use offline computation on workload traces to generate the partition scheme; therefore, this method can not react to the changes in workloads in time and introduces load imbalance when hotspots change.

Batching-based scheduling method. As shown in Figure 1 (c), in this method, worker threads collect a batch of transactions and divide the batch into `n` groups (`n` is the worker thread count), intending to minimize contention between groups. After that, each worker thread executes a transaction group with almost no contention. Specifically, recent studies on this method [18–21] use a *graph partition* algorithm that treats transactions as nodes, contention between them as edges, and transaction groups as sub-graphs.

This method is adaptive and can react to changes in workloads. Moreover, it avoids load imbalance by guaranteeing even grouping (i.e., high accuracy). However, it introduces high latency for batching (i.e., high overhead). For example, in a state-of-the-art system, Strife [21], the batch size is 10K, which adds about 5*ms* latency for requests. This is insufferable for *in-memory* transaction systems [22] which usually have microsecond-level latency.

2.3 Scheduling with SmartNICs

We observe that exploiting emerging SmartNICs (as shown in Figure 1 (d)) is promising to break the accuracy-overhead trade-off due to the following two advantages.

First, every transaction request/response flows through the NIC, so the NIC is the natural place to implement a scheduler, which can route packets to any worker thread and keep track of running/queuing transactions on each worker thread. Then, with this piece of information, NICs have an opportunity to make accurate and adaptive scheduling decisions. Recent studies [23,27–29] leverage the NIC as a scheduler to dispatch key-value requests, addressing the load imbalance or head-of-line blocking problems. However, they do not take transaction semantics into consideration.

Second, FPGA-equipped SmartNICs can perform lots of computation to generate accurate scheduling decisions with low latency. FPGA-equipped SmartNICs allow users to customize network processing logic. Also, the FPGA modules can reduce scheduling overhead with hardware acceleration instead of amortizing it by batching transactions. Recent studies [30–36] also show that packet manipulation processes with data/pipeline parallelism (i.e., encryption and serialization) are suitable to be offloaded to the FPGA modules on SmartNICs. Such offloading can accelerate networking and relieve the host-side CPU/memory burdens.

FPGA-equipped SmartNICs can be divided into two categories, on-path and off-path, as shown on the right of Figure 1. They vary in the connection architecture between the FPGA and other NIC components. In an *on-path* SmartNIC, the FPGA is located between network ports and the NIC ASIC. In this type of SmartNICs, the FPGA modules need to process all link-layer network traffic, complicating the FPGA logic. Contrarily, in an *off-path* SmartNIC, the NIC ASIC is the same as that of a standard NIC, and packets are routed to either the host or the FPGA by an on-board PCIe switch. We use an off-path SmartNIC, *Mellanox Innova-2* [8], so that we can focus on the scheduling logic while leaving sophisticated network functions (e.g., reliable delivery and ordering) to the full-fledged NIC ASIC.

Mellanox Innova-2 [8] has a ConnectX-5 NIC ASIC and a Xilinx XCKU15P FPGA. It has two 25Gbps ports and uses PCIe 3.0 ×8 to connect the NIC ASIC, the FPGA, and the CPU. The CPU communicates with the FPGA via MMIO (memory-mapped IO) or the FPGA's DMA engine. The ConnectX-5 NIC ASIC supports RDMA (remote direct memory access), through which remote clients can read/write the server's memory while bypassing the server's CPUs.

Challenges of transaction scheduling with SmartNICs. It is non-trivial to design a transaction scheduler using SmartNICs due to the following challenges: 1) The FPGA is adept in fixed and simple execution flows. We need to describe complicated inter-transaction contention in a hardware-friendly manner to squeeze out the FPGA's parallelism capabilities. 2) Worker threads have important information about transaction execution, e.g., abort rate, which is indispensable for scheduling. Thus, to enable effective transaction scheduling, worker threads should gather this information as feedback to adjust the scheduler. 3) Transaction systems use various concurrency control protocols and face various workloads; however, the FPGA redesign is costly. It is difficult for an application-specific FPGA-based scheduler to support various transaction systems [37, 38]. To be generalized for all transaction systems, the scheduler design (e.g., request format, scheduling algorithm, and feedback interfaces) should not encode any application-specific characteristics.

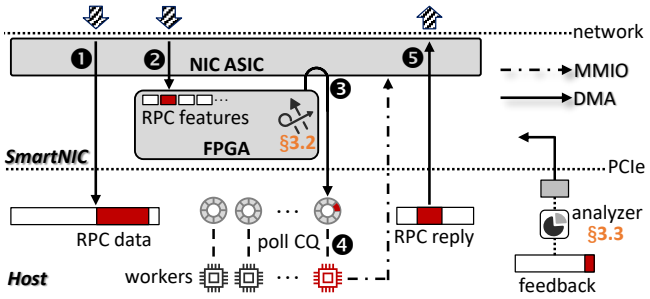

Figure 2: **A scheduling-enabled RPC with SmartNICs.**

3 AlNiCo

3.1 Overview

To reduce the inter-transaction contention and break the accuracy-overhead trade-off, we propose AlNiCo, a SmartNIC-accelerated contention-aware request scheduler. AlNiCo uses hardware acceleration of the FPGA for low overhead and provides generalized software feedback interfaces for high accuracy.

Transaction systems typically provide a *stored procedure* interface, where each type of transaction is compiled into a procedure, and clients issue transactions via remote procedure calls (RPCs). To achieve request scheduling, SmartNICs need to parse the procedure parameters and then detect inter-transaction contention. However, the parameters are application-specific, including keys that the transaction will access, the access modes (read/write), the values that the transaction will write, and so on. The format of the parameters also varies across applications.

Therefore, in order to support diverse applications without encoding application-specific properties, we include a fixed-form header in each request. Clients need to convert the parameters into the fixed-form header. We call this the ***request feature vector***. Scheduling also needs the information on running/queuing transactions in worker threads and workload characteristics. AlNiCo encodes them in a hardware-friendly manner and leverages the data and pipeline parallelisms of the FPGA to make scheduling decisions (§3.2).

We design a **scheduling-enabled RPC** with SmartNICs as the communication mechanism between clients and the server, allowing the FPGA to receive requests and schedule them. Figure 2 describes its workflow. The server maintains a data buffer on the host memory and a feature vector buffer on the FPGA. To issue a transaction, the client sends the data (❶) and the feature vector (❷) to their respective buffers via two one-sided RDMA writes posted together. AlNiCo's scheduler (on the FPGA) polls the feature vector buffers to determine the arrival of new requests. The scheduler then selects the most appropriate worker thread based on the request's feature vector, the runtime states of worker threads, and the workload characteristics. After making the scheduling decision, the scheduler notifies the selected worker thread via writing its receive completion queue (CQ) (❸). The CQ entry includes only the address of request data but not the feature vector. Since RDMA enforces ordered writes, after reading a new CQ entry (❹), the worker thread can get the completed data from the RPC data buffers. The worker thread then executes the transaction and replies to the client via a normal RDMA write (❺).

To make the scheduler adaptive to workload changes, AlNiCo employs a software feedback mechanism (§3.3). It allows the software to use information about transaction execution (e.g., records that cause transactions to be aborted) to guide the scheduler. The feedback interface is generalized for various concurrency control protocols and different workloads. The FPGA fetches the feedback periodically via DMA reads outside the critical path.

3.2 Accelerated Scheduling on The Hardware

To describe the inter-transaction contention and schedule transactions in a hardware-friendly manner, we identify three types of states for scheduling: *request state* (§3.2.1), *worker state* (§3.2.2), and *global state* (§3.2.3). We then encode the states into compact *vectors* and translate the scheduling algorithm into the fast *vector computation*, which is well-suited for the FPGA (§3.2.4).

3.2.1 Request state

AlNiCo uses the request state to describe the resources required by a transaction. Clients embed the complex parameters of a transaction request into a request feature vector. Specifically, each feature vector ($\vec{f_{req}}$) has `L` elements[1]. We use a mapping function to divide all records into `L` *groups*. Each element in the vector represents whether the transaction will access the corresponding group of records. The mapping function is the same in all clients. To generate the feature vector, clients enumerate the keys[2] in the request parameters and set the corresponding elements. Each element can be encoded into either 1-bit or 2-bit element. A 1-bit element describes two access modes: no operation and access (read or write). A 2-bit element describes three access modes: no operation, read, and write, having better expressibility but consuming more space. The scheduler can use the feature vectors of requests to estimate whether contention exists between them. For example, if two requests set the same element in their feature vectors, these two requests might have contention.

The mapping function for generating feature vectors. The mapping function is used to select a group for a given key. Clients use a hash function to calculate a hash value (`v`) for each key and use `v % L` as the group number. Due to hash collisions, clients might map two different keys to the same group.

[1] All vectors have a length of `L` in this paper.
[2] The primary keys of records.

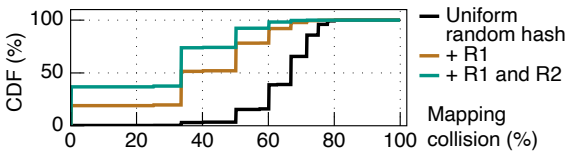

Figure 3: **Mapping collisions of two New-order txns.**

To reduce the negative impact of hash collisions, AlNiCo imposes two requirements (**R**) on the mapping function. First, transaction systems manage records with different semantics in different tables (e.g., the CUSTOMER table and the ORDER_LINE table in TPC-C) and access records through ⟨*table_id*, *key*⟩. Therefore, clients should map the keys of different tables into different groups (**R1**). For example, if an application has two tables, the mapping function maps the keys of one table to $\frac{L}{2}$ groups and maps the keys in the other table to the other $\frac{L}{2}$ groups. Second, because different tables can have different sizes (i.e., numbers of records), the number of groups of a table should be proportional to its size (**R2**). We test the CDF of the mapping collisions between New-order transactions in TPC-C, as shown in Figure 3. The mapping function with these two requirements reduces the mapping collisions compared with the simple uniform random hash. Note that when the table count is large, all tables should instead share the feature space since there are not enough elements in the feature vector.

Discussion. There are two corner cases worth discussing when describing the request state.

How to handle data growth. In our current design, the feature vector length `L` and the key-to-feature mapping function are static at runtime. When the amount of data grows, AlNiCo does not need to increase `L` correspondingly in most cases. This is because the determinant of mapping collisions is the amount of hot data instead of the data in the entire system; only a small amount of data is accessed frequently. However, if changing `L` or the mapping function is a must (e.g., when the hotspot size changes significantly), AlNiCo can reload the on-NIC scheduler with the new configuration (which takes 4.18*ms* on Innova-2 in our evaluation). In that case, clients also need to update the new configuration.

How to describe pre-unknown keys. Clients specify most of the keys of the records that a transaction will access in the request's parameters. However, some keys are pre-unknown and can only be determined through transaction execution. To describe the required resources for these transactions more accurately, AlNiCo employs the following two speculative optimizations. First, for transactions containing range queries, clients randomly generate the keys in the ranges and set the corresponding elements of the request feature vector. Second, for transactions that access records via non-primary keys, the client maintains a secondary-index cache, which maps each non-primary key to a set of primary keys, to determine the records that those transactions will access.

	no op	read	write	read & write
request feature vector $\vec{f_{req}}$	00	10	01	01
worker feature vector $\vec{f_w}$	00	01	11	11

Table 1: **Encoding of read and write modes.**

3.2.2 Worker state

AlNiCo uses the worker state to describe the resources that are being or will be accessed by worker threads. Each worker thread has different running/queuing transactions. Therefore, a state of the `i`-th worker thread is the union of the feature vectors of its running/queuing transactions. We call it the **worker feature vector** ($\vec{f_{w_i}}$ for the `i`-th worker). This vector has the same format as the request feature vector. Further, to allow worker threads to steer requests actively, AlNiCo introduces the other worker state, called the **worker steering vector**. Each worker thread maintains a steering vector with 1-bit elements ($\vec{s_{w_i}}$ for the `i`-th worker). Worker threads can set their steering vectors via the software feedback interface, which is described in detail in §3.3.2.

How to describe the contention. AlNiCo uses the bitwise `AND` (&) operation to describe the contention between a new request and the running/queuing requests in a worker. When the element in the feature vector is 1-bit, AlNiCo encodes request features and worker features in the same way, where `1` represents access (read or write) and `0` represents no operation. The `AND` operation between request features and worker features estimates all concurrent accesses to the same group. When the element is 2-bit, AlNiCo encodes the access modes in request features and worker features in different ways, as shown in Table 1. The result of `AND` operation between the request features and the worker features describes read/write and write/write accesses to the same group as contention.

How to maintain worker feature vectors. The scheduler maintains a feature vector queue (FVQ) for each worker. The FVQ stores the feature vectors of the worker's running/queuing requests ($\vec{f_{req}}$), in the same order as the requests in the worker's CQ. The new worker feature vector is recalculated on adding/deleting an $\vec{f_{req}}$ to/from the worker's FVQ. Specifically, for each worker, the scheduler keeps two counter vectors. A counter vector counts the number of read requests on different groups, and the other vector counts the write requests. When a request is added/deleted, the scheduler updates the counter vectors and then converts them to the aforementioned worker feature vector with the FPGA's parallelism.

When a request completes, the scheduler needs to remove its feature vector from the FVQ. One naive method is to let the worker thread notify the scheduler after executing the request. However, it requires an additional MMIO operation, which may slow down the system. AlNiCo designs a lazy updating mechanism. Specifically, the scheduler will check the completed requests only when the scheduler pushes a new request to a worker's CQ. This can be done efficiently through the FPGA's DMA without invoking the CPU.

State of	Name	Format	Maintainer
request	feature: $\vec{f_{req}}$	vector: 2-bit × L	client
worker	feature: $\vec{f_w}$	vector: 2-bit × L	scheduler
	steering: $\vec{s_w}$	vector: 1-bit × L	worker thread
global	weight: $\vec{W}$	vector: 8-bit × L	
	worker-set table	type → worker set	

Table 2: **The states used for making scheduling decisions.**

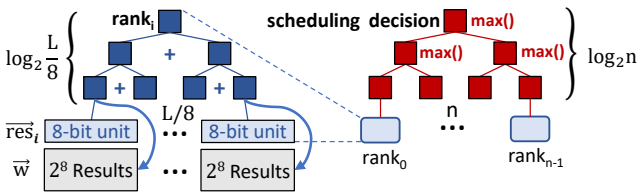

Figure 4: **Hardware acceleration for scheduling.**

3.2.3 Global state

AlNiCo uses the global state to describe workload characteristics. Real-world applications have skewed access patterns, where some records are accessed frequently (i.e., hotspots). These hot records are the main source of contention. Therefore, in AlNiCo, one of the global states is the **weight vector**. Each element in the vector has 8 bits and is the weight of a group. The element actually represents the total sum hotness of the keys in a group. To make our scheduler adaptive to hotspot changes, the weight vector is dynamically updated by the software, and the feedback interface is detailed in §3.3.1.

The other global state is the **worker-set table**, which stores a set of worker threads for each request type. The scheduler selects a worker only from each request type's worker set. This is used to avoid the head-of-line blocking for long-running requests, which is described in detail in §3.3.3.

3.2.4 Making scheduling decisions

As summarized in Table 2, states are formatted in vectors with length L, except for the global worker-set table. Therefore, the scheduler can use fast vector computation to make scheduling decisions. For each new request, the scheduler performs the following three steps.

Step#1. The scheduler searches the worker-set table to get the set of workers for this type of transaction.

Step#2. For each worker, the scheduler calculates a **contention rank** ($rank_{w_i}$ for the i-th worker) between the new request and the running/queuing requests in the worker. The contention rank is calculated using the following formula:

$$rank_{w_i} = \left(\texttt{sign}\,(\vec{f_{req}}\ \texttt{AND}\ \vec{f_{w_i}})\ \texttt{AND}\ \vec{s_{w_i}}\right)\ \texttt{DOT}\ \vec{W}$$

Here we assume elements in steering vectors ($\vec{s_{w_i}}$) are 1. When each element in feature vector is 2-bit, AlNiCo needs the sign function to transform the result of ($\vec{f_{req}}$ AND $\vec{f_{w_i}}$) to a vector with 1-bit elements for later calculations. If the input is greater than 0, the result of the sign function is 1, otherwise it is 0. This result $\vec{res_i}$[3] represents the same groups accessed by the new request ($\vec{f_{req}}$) and the i-th worker's running/queuing requests ($\vec{f_{w_i}}$). The contention rank of the i-th worker is the weighted sum ($\vec{W}$) of $\vec{res_i}$.

[3]We denote the results of sign ($\vec{f_{req}}$ AND $\vec{f_{w_i}}$) AND $\vec{s_{w_i}}$ as $\vec{res_i}$.

Step#3. The scheduler selects the worker with the *highest* contention rank among n workers[4] to route the request.

Hardware acceleration. We leverage the FPGA to accelerate the scheduling calculation, including the AND (&) and DOT PRODUCT (·) to compute the contention rank of each worker, and the MAX to select a worker with the highest rank.

(1) We leverage the FPGA data parallelism to calculate all bits simultaneously during the AND operation.

(2) Because each element in $\vec{res_i}$ is 1-bit, the DOT PRODUCT operation between $\vec{res_i}$ and $\vec{W}$ is actually a summation operation, which adds a weight value into the sum only if the corresponding bit in $\vec{res_i}$ is 1. We optimize the summation via a binary tree reduction algorithm. As illustrated in Figure 4, the reduction algorithm executes the + operation in parallel using many computation units. Each unit adds two values in the L inputs, and then the $\frac{L}{2}$ results become new inputs. It repeats this process until there is only one result left. This algorithm only needs to execute $\log_2 L$ times but uses $\frac{L}{2}$ computation units. To save the computation units, we trade memory consumption. Because the weight vector is periodically updated, it is constant for a period. The FPGA stores the results of the DOT PRODUCT between different $\vec{res_i}$ and the constant $\vec{W}$ in advance. However, $\vec{res_i}$ has 2^L cases, and it is impractical to store all results. Therefore, we divide $\vec{res_i}$ into $\frac{L}{8}$ segments. Each segment includes 8 bits and has $2^8 = 256$ results. There are $\frac{L}{8} \times 2^8$ results in total, small enough to be stored in the FPGA's memory. The FPGA still uses the reduction algorithm to compute the sum of these segments' results. Storing segment result trades the memory for reducing the number of computation units from $\frac{L}{2}$ to $\frac{L}{16}$.

(3) The MAX operation is to find the highest value among n values. As shown in Figure 4, the FPGA uses the MAX reduction algorithm, which requires only $\log_2 n$ steps.

These hardware optimizations are leveraged to reduce the overhead of making accurate scheduling decisions.

3.3 Adaptive Feedback from The Software

To make the scheduler adaptive to the changes in workload characteristics, AlNiCo allows the software to set the following three states: the weight vector $\vec{W}$ (§3.3.1), the workers' steering vectors $\vec{s_w}$ (§3.3.2), and the worker-set table (§3.3.3). AlNiCo provides generalized feedback interfaces to worker threads and uses a lightweight *analyzer thread* to update the

[4]We denote n as the worker count in the following paper.

```
1  void hotness_feedback(i: worker id, key){
2    hotness[i][hash(key)]++;
3  }
4
5  void affinity_feedback(group_A, group_B){
6    set_affinity(group_A, group_B)
7  }
8
9  void worker_set_feedback(txn_type, worker_set);
10
11 void update_weight(E: epoch id){ // Sec.3.3.1
12   TP(E) = ∑_{i=0}^{n-1} tp_i
13   h⃗_all = ∑_{i=0}^{n-1} h⃗_i × tp_i / TP(E)
14   if ( E == 0 || TP(E) < (1-c) × TP(E-1) )
15     update W⃗ based on h⃗_all  // normalization
16   next_epoch(): clear tp_i and h⃗_i
17 }
18
19 void update_steering(W⃗){ // Sec.3.3.2
20   g_groups = [] // the indexes of guideline groups
21   for weight in sorted W⃗ { // from the highest
22     new_group = weight.index;
23     if no_affinity(new_group, g_groups)
24       g_groups.append(new_group);
25     if (g_groups.size() == n) break;
26   }
27   update steering vectors based on g_groups
28 }
```

Listing 1: **The feedback interfaces and algorithms.**

scheduler's states based on the feedback. Listing 1 shows the feedback interfaces and the algorithms for translating the feedback into the states needed by the scheduler.

3.3.1 Hotness feedback

We define the keys that *cause contention frequently* as the hotspots in the transaction system, instead of the keys that are accessed frequently. The hotspots keep changing over time. AlNiCo requires the software to identify the hotspots in real-time and update the weight vector correspondingly.

Hotness feedback interface. We provide a hotness feedback interface as shown in Lines 1-3. Each worker increases the hotness of a key in its exclusive hotness vector ($\vec{h_i}$) without any coordination with other workers. Different concurrency control protocols invoke the interface in different situations, and we divide them into two cases based on the way they address the contention. First, with OCC and the 2PL that avoids deadlock by wait-free algorithms, transactions might get aborted and re-executed. The worker increases the hotness of the *record that causes the abort*. Second, with the 2PL that holds locks sequentially to avoid deadlock, transactions might be blocked while trying to acquire a lock. Every time a transaction *retries acquiring the lock of a record*, the worker increases the corresponding hotness.

How to update the weight vector. AlNiCo employs an epoch-based updating approach. The analyzer thread collects these hotness vectors at the end of each epoch. Further, each worker measures its throughput (tp_i). Then, as shown in Line 13, the analyzer calculates the weighted average (based on tp_i) of workers' hotness vectors as the global hotness vector ($\vec{h_{all}}$) and uses it to update the weight vector (Line 15).

When to update the weight vector. AlNiCo updates the weight vector only when the real hotspots change. When the system is cold-started (e.g., in epoch 0), requests are randomly scheduled, and therefore $\vec{h_{all}}$ can reflect the real hotspots. However, when the system is stable, due to AlNiCo's effective scheduling mechanism, those hot keys detected right after the cold start no longer cause contention, and their degrees of hotness is low in the new $\vec{h_{all}}$. In this case, $\vec{W}$ should remain unchanged to keep the scheduler effective.

The software uses throughput decreases or abort rate increases as the signal of the hotspot changes. Line 14 shows how to detect changes in hotspots. In this algorithm, we use the throughput (TP) decreases as the signal. If the throughput decreases significantly, i.e., TP drops by more than a factor of c, AlNiCo will store $\vec{h_{all}}$ in a state buffer that can be read by the NIC via DMA.

When the workload is read-intensive, workers increase the hotness every time a record is read. The analyzer simply ignores the workload change signal and always updates the weight vector at the end of epochs. In this way, AlNiCo helps worker threads to better leverage cache locality.

3.3.2 Affinity feedback

A contention-aware scheduler not only schedules conflicting transactions to the same workers but also schedules transactions without contention to different workers to make all workers busy. To this end, AlNiCo introduces the steering vectors ($\vec{s_w}$) to guide the scheduler. As shown in the formula in §3.2.4, only if the element in a worker's steering vector ($\vec{s_{w_i}}$) is 1, the corresponding element in its feature vector ($\vec{f_{w_i}}$) is valid. The key idea is to select n (i.e., worker count) **guideline groups** that have the *highest weights* and *do not have affinities with each other*. AlNiCo assigns different guideline groups to different workers to achieve the aforementioned goal. Each worker's steering vector consists of 1-s for its assigned guideline group and all non-guideline groups by default. For other guideline groups, the corresponding element in the steering vector is 0. In this way, the worker will steer transactions accessing its guideline group. Therefore, in a running system, the feature vector of a worker represents the groups that have affinities with the worker's guideline group.

Affinity feedback interface. To find the n guideline groups, AlNiCo needs the affinity characteristics of workloads. We provide the affinity feedback interface (Lines 5-7) that allows users to offer the affinity hint for different workloads.

How to update the steering vectors. As shown in Lines 19-28, the analyzer thread first sorts all groups according to their weights and enumerates them from the group with the highest weight. Then, the analyzer thread checks whether the group has an affinity with the already found guideline groups. If there is no affinity between them, the group will be labeled as a new guideline group.

3.3.3 Reserving workers for long-running transactions

In real-world workloads, some transactions take a long time to finish (e.g., analytical transactions) and block the transactions assigned to the same worker (i.e., head-of-line blocking). For example, a Delivery transaction accesses about 10 times more keys than a New-order transaction. We reserve some workers to handle these long-running transactions, which is similar to the size-aware request scheduling mechanism [39, 40]. We provide an interface to set the worker set for different types of transactions. Our design only sets the worker-set table on the system startup for workloads that have long-running transactions.

4 Implementation

We implement AlNiCo on an Innova-2 SmartNIC [8] and use STO [41], a state-of-the-art transaction processing framework, as the backend transaction processing module.

The scheduler on the FPGA. We implement two fundamental drivers for SmartNICs: a PeerDirect driver [42,43] to build peer-to-peer communication between the NIC ASIC and the FPGA, and a DMA driver [44] to connect the FPGA and the host. Innova-2 holds a Xilinx KU15P FPGA. In our implementation, the feature vector length L is 512, and we encode read-/write modes to the same bit to save feature space. The clock frequency is 250MHz. With these configurations, AlNiCo consumes 54 cycles for each request to make the scheduling decision. AlNiCo updates global states every 1024 requests (every 22ms) in the background, consuming 1184 cycles. Therefore, we set the epoch size for gathering the feedback in software to 20ms. The implementation consumes 159K (30.48%) LUTs, 157K (15.10%) FFs, and 678.5 (68.95%) BRAMs. We evaluate the computation cycles and resource usage with different feature vector lengths in §5.6.

The transaction processing module on the host. We choose STO as our backend transaction processing module because it implements various CC protocols in the same framework and is high-performance. To use STO, we locate the contention handling function in the framework and then add feedback interface codes to these functions. The parameters of this feedback interface are the records that triggered the contention handling function. On the client side, we do not modify the original request format of the stored procedure, and we only add the field for the feature vector in the request header. We use the following four concurrency control (CC) protocols in the STO framework:

- OCC: it is based on Silo [45], which avoids allocating global timestamps to improve multi-core scalability.
- TicToc [46]: it is an OCC variant that assigns commit timestamps dynamically according to read/write set.
- MVCC: it is based on Cicada [47], which optimizes the management of the timestamps and multi-version values.
- 2PL [48, 49]: it uses NO_WAIT to avoid deadlock; when a transaction fails to acquire a lock, it immediately aborts.

All four concurrency control protocols run at the serializable isolation level. The table indexes use Masstree [50], which supports range queries.

5 Evaluation

We evaluate AlNiCo under various workloads and seek to answer the following five questions:

1. How does AlNiCo perform compared with existing transaction scheduling methods (§5.2)?
2. How does AlNiCo react to dynamic workloads (§5.3)?
3. Why are SmartNICs necessary for the contention-aware request scheduling (§5.4) ?
4. How compatible is AlNiCo with various concurrency control protocols (§5.5) ?
5. What are the overhead and the limitation incurred by the on-NIC scheduler in AlNiCo (§5.6)?

5.1 Experimental Setup

Experimental environment. We run all experiments on three machines, one as the server and two as clients.

Hardware settings. Each machine is equipped with two 12-core Xeon E5-2650 v4 2.20GHz CPU sockets, PCIe 3.0 interfaces, and memory of 128GB. The database server is equipped with a dual-port 25Gbps Innova-2 SmartNIC. Each client is equipped with one 100Gbps Mellanox ConnectX-5 NIC. They are connected by a 100Gbps Mellanox switch.

Software settings. Each client thread issues up to 4 transaction requests simultaneously (i.e., the queue depth is 4). We adjust the number of clients and the number of asynchronous requests per client to change the test pressure. The server uses 20 cores for worker threads, 2 cores for long-running transactions in TPC-C workloads, and one core for the analyzer thread and performance measurement. For a fair comparison, the competitors use the RDMA-based RPC with optimizations from previous RDMA systems [51–58], and only use the normal ASIC part of the Innova-2 NIC. We evaluate the performance of this RPC against our scheduling-enable RPC in §5.6.

Workloads. We use the following benchmarks:

TPC-C [6] simulates the activity of a wholesale supplier with five types of transactions. We use the full-mix TPC-C. AlNiCo reserves two cores for the long-running delivery transactions.

YCSB-T is a transactional extension of YCSB, which is a popular KV store benchmark [59]. It has 20 tables as many as the worker threads, and the key space is 100M. Each record contains an 8-byte key and a 384-byte value [39, 60]. Each transaction accesses 16 records in a single table, and the keys are generated according to the Zipf distribution (θ=0.99).

YCSB-HOT is a dynamic workload based on YCSB-T. Different from YCSB-T, it has 100 tables, and 20 tables are hot at one time. The hot tables are changed every two seconds to

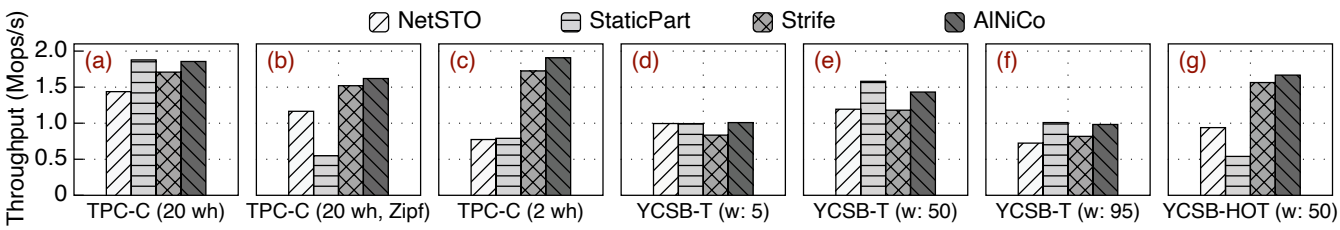

Figure 5: **Throughput.** *The wh in TPC-C is the number of warehouses, (a): the low-contention workload, (b): the skewed workload under Zipf θ=0.99, (c): the high-contention workload. The w in YCSB is the percentage of write operations.*

simulate the changes in time-varying applications. Keys are generated according to the Zipf distribution (θ=1.2).

Competitors. We compare AlNiCo with five systems: a baseline system without scheduling, two systems using existing scheduling methods, and two CPU-based versions of AlNiCo.

NetSTO is a baseline system in which clients randomly select a worker to send transaction requests without scheduling.

StaticPart is a system using the static data partitioning method. For TPC-C workloads, StaticPart partitions data based on the warehouse ID [12]. For YCSB-T and YCSB-HOT workloads, StaticPart partitions data based on the ID of tables.

Strife [21] is a system using the batching-based method. The batch size is 10K, and the batch waiting time is 5ms, the same configuration as in Strife's paper. To support batching, the queue depth in clients is 1K.

AlNiCo-CPU-2 is a CPU-based version of AlNiCo, which reserves two dedicated threads to execute the scheduling logic. Each scheduler thread connects to half of the clients and communicates with the workers through separated message queues. The worker/global states are shared by the scheduler threads and updated with atomic operations.

AlNiCo-CPU-N is the other CPU-based version of AlNiCo. It co-locates the worker logic and scheduler logic in each thread, where they multiplex the CPU resource. All threads share the worker/global states. The client connections are evenly distributed among threads. Each thread can make scheduling decisions and delegate requests to others as a scheduler.

5.2 Overall Performance

We first evaluate the peak throughput of NetSTO, StaticPart, Strife, and AlNiCo with various workloads in Figure 5. Then we evaluate the latency of different types of transactions under TPC-C by varying request pressure in Figure 6.

Throughput under TPC-C. We evaluate TPC-C under different levels of contention by varying the number of warehouses. We set the warehouse count to 20 and 2 to simulate low-contention and high-contention scenarios. We also introduce a skewed TPC-C, which has 20 warehouses. In this skewed TPC-C, clients select the warehouse according to the Zipf distribution with θ = 0.99. In StaticPart, when the warehouse count is equal to the worker count (20), each worker manages an exclusive warehouse, and when the warehouse count is 2, every 10 workers manage a warehouse. For a fair comparison, StaticPart, Strife, and AlNiCo all reserve 2 worker threads for long-running transactions. Figure 5 (a)-(c) show the throughput of TPC-C under these configurations, and we have the following two observations.

First, in the low-contention workloads (Figure 5 (a)), StaticPart has the highest throughput, outperforming NetSTO, Strife, and AlNiCo by 1.30×, 1.11×, and 1.06×, respectively. This is because only 10% of the transactions are cross-warehouse transactions, and the warehouse count is the same as the worker count, which is a good case for the static data partition method. The transaction grouping algorithm in Strife and the feedback mechanism in AlNiCo cost the CPU resources that are originally used for transaction execution. In the skewed workloads (Figure 5 (b)), the throughput of StaticPart is only 47% of NetSTO. This is because clients access hot warehouses while the data partition in StaticPart is static, which results in load imbalance.

Second, in the high-contention workloads (Figure 5 (c)), 1) AlNiCo improves throughput by 2.46× compared with NetSTO. This is because the contention-aware scheduling reduces the contention between the running transactions and saves the CPU resources to execute more transactions. 2) The throughput of AlNiCo is 2.41× higher than StaticPart. This is because the workload is not partitionable and there is still a lot of contention between concurrent transactions.

Throughput under YCSB-T. Figure 5 (d)-(f) show the throughput under YCSB-T with different read/write ratios. We have the following two new observations.

First, all scheduling methods do not have benefits under the read-intensive workload (Figure 5 (d)). This is because 1) there is less contention for read-intensive transactions; 2) the throughput of this workload is bound by the bandwidth of the NIC (i.e., 50Gbps); 1Mops/s throughput in this workload requires about 45.6Gbps outbound bandwidth to transmit data.

Second, the highest throughput of write-intensive workloads (Figure 5 (f)) is bound by the NIC in AlNiCo. However, NetSTO can not use the full bandwidth because this workload causes more contention (especially the write-write contention) than the read-intensive one.

Latency under TPC-C. We evaluate the latency distribution for New-order transactions and Delivery transactions (long-running) with varying throughput. Figure 6 shows the median

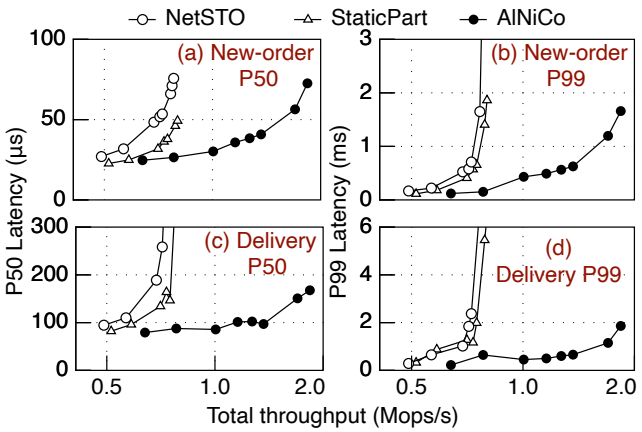

Figure 6: **Latency of New-order and Delivery transactions under TPC-C (2 warehouses).**

(P50) and 99th percentile (P99) latency, and we have the following two observations.

First, as shown in Figure 6, when the throughput is low, the median latency of New-order in AlNiCo (24.1μs) is higher than that in NetSTO (23.6μs). This is because the requests in AlNiCo have extra latency for scheduling logic and two extra PCIe communication latency for the off-path SmartNICs. As the throughput increases, the New-order median latency in AlNiCo is lower than NetSTO because the reserving worker threads for long-running transactions prevent them from blocking other normal transactions.

Second, in NetSTO, before the throughput reaches the peak, the latency increases faster than in AlNiCo. This is because the contention blocks transactions or causes them to retry, increasing the latency of the running transactions and blocking subsequent requests. Under a similar throughput of AlNiCo (0.78Mops/s) and NetSTO (0.71Mops/s), AlNiCo reduces the median latency of New-order from 51.8μs to 26.5μs.

In addition, median and P99 latency in Strife exceed 7ms, larger than the batch waiting time (5ms). In Strife, the median latency of New-order is between 7.00ms and 19.92ms, and the P99 latency is between 12.54ms and 25.18ms, which are not plotted in the figure to avoid obscuring other results. This is because the end-to-end latency includes the network stack latency, the batch time, the transaction grouping time, and the transaction execution time. AlNiCo does not sacrifice the request latency: at the peak throughput, the P99 tail latency of the New-order transaction is no more than 1.17ms, and the median latency is only 72.7μs.

5.3 Dynamic Workloads

We evaluate AlNiCo's ability to adapt to dynamic workloads by changing the hot tables in YCSB-HOT. We run YCSB-HOT with a 50/50 read/write ratio.

Figure 7 shows the throughput over time, from which we have the following three observations.

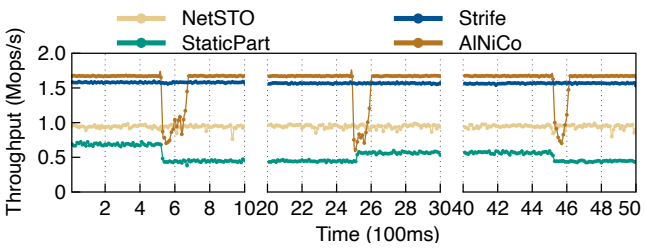

Figure 7: **YCSB-HOT throughput over time.** *Hot tables change every 2s; measuring the throughput every 10ms.*

First, AlNiCo takes about 150ms to adapt to the changes of hot tables. At the beginning of changes, the throughput drops to the lowest point, even worse than NetSTO. This is because the weight vector fails to reflect the conflicting hotspots in the new hot tables. The steering vectors based on the historical hotspots can not guide the scheduling. Before adjusting the scheduler to fit the new workloads, AlNiCo suffers performance jitters. This is because the feedback in AlNiCo reflects the workload characteristics over time, and it takes a while to make the characteristics of old hot tables fade away. Second, Strife reacts more quickly to dynamic workloads without experiencing throughput degradation. This is because the results of transaction grouping in Strife are based on the information in a batch and do not rely on the historical information of the workloads. However, the median/P99 latency (at the peak throughput) of AlNiCo and Strife are 72.6us/1.66ms and 12.21ms/15.23ms, respectively. This is because Strife introduces extra latency due to batching. Third, StaticPart performs worse than NetSTO and varies with the workload changes. This is because the hot tables cause the load imbalance problem in this static data partition method.

In summary, according to §5.2 and §5.3, the static data partition methods are the best for the partitionable workloads, but they can not handle skewed or dynamic workloads. The batching-based scheduling methods can handle various workloads, but they introduce orders of magnitude higher latency for requests. Thanks to SmartNICs, AlNiCo can handle skewed or dynamic workloads with low latency.

5.4 Comparison with CPU-based AlNiCo

We demonstrate the necessity of SmartNIC-accelerated design for contention-aware scheduling by comparing it with two versions of CPU-based AlNiCo.

Figure 8 shows the throughput of CPU-based AlNiCo with varying worker thread count. The workload is TPC-C with 2 warehouses. We have the following three observations.

First, AlNiCo-CPU-2 brings an improvement in throughput when the worker thread count is small. However, with more worker threads, the performance decreases. This is because two scheduler threads have limited computing resources, and the scheduling complexity increases linearly with the number of worker threads. As a result, the CPU can not accelerate the scheduling computation.

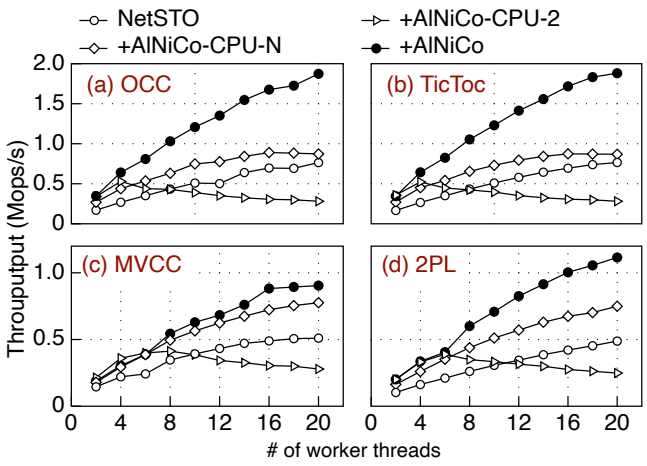

Figure 8: **TPC-C (2 warehouses) throughput with four different concurrency control protocols.**

Second, AlNiCo-CPU-N is not as good as AlNiCo-CPU-2 when the worker thread count is small because the scheduling consumes the workers' resources. It improves the throughput compared with NetSTO because its scheduling overhead is less than the overhead for transaction aborts/blocking. Moreover, its throughput is scalable with the thread count because the resources for the scheduler increase linearly with the thread count.

Third, the improvement of the AlNiCo-CPU-N version is small. The throughput of the scheduler itself is not the bottleneck in AlNiCo-CPU-N, but it can not enjoy the acceleration of the FPGA. The scheduling logic costs lots of CPU resources. AlNiCo speeds up individual request scheduling by fine-grained parallel computation, and the computation of multiple requests is pipelined. These provide higher performance with less CPU resource cost.

5.5 Generality of AlNiCo

We demonstrate the generality of AlNiCo by changing the concurrency control protocols in STO. Figure 8 shows the throughput with varying worker thread count and the four CC protocols described in §4. The workload is TPC-C with 2 warehouses. Note that, since the 2PL implementation in STO has performance abnormalities under full-mix TPC-C, we only evaluate the New-order transactions for 2PL (Figure 8 (d)). Comparing AlNiCo with NetSTO, we have two observations:

First, AlNiCo brings performance improvement for these 4 CC protocols. With 20 workers, AlNiCo improves the throughput by 2.45× (OCC), 1.77× (TicToc), 2.45× (MVCC), and 2.28× (2PL), respectively. This is because AlNiCo provides a generalized hotness feedback interface and affinity feedback interface to generate the worker states and global states. The feedback mechanism in AlNiCo is generalized for various concurrency control protocols.

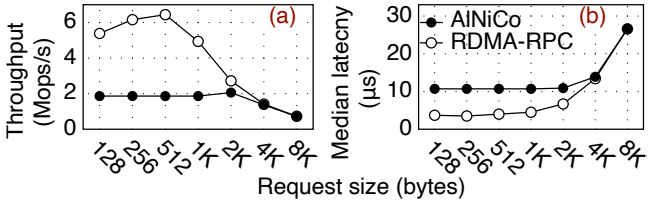

Figure 9: **RPC throughput (a) and median latency (b) with varying request sizes.**

		L=128	L=256	L=512	L=1K	L=4K
Accuracy rate	TPC-C	22.7%	36.7%	59.8%	85.5%	99.9%
	YCSB-T	5.2%	6.5%	9.9%	28.4%	84.4%
	YCSB-HOT	17.6%	20.7%	29.4%	42.7%	79.8%
Computation cycles		37	37	54	101	N/A
BRAM/LUT/FF(%)		6/2/9	10/3/11	16/5/16	30/10/26	N/A

Table 3: **The trade-off of feature vector length L.**

Second, 2PL performance is worse than other protocols. This is because 2PL needs to write shared memory to acquire the read lock, while the weakness of OCC's high rollback overhead is negligible in the in-memory system. MVCC has the extra overhead of maintaining version information, so it has lower overall throughput than OCC and TicToc.

5.6 Overhead and Limitation

We evaluate the overhead of AlNiCo and discuss its limitation.
Overhead of clients. The client specifies the request feature vector when serializing a transaction into a network message. It takes about 23ns for each key in the transaction parameters.
Overhead of SmartNICs. Figure 9 shows the performance of RDMA-RPC and AlNiCo under a micro-benchmark, where the server sends an 8-byte reply to clients immediately as soon as receiving a request. When the request is less than 2KB, the IOPS and latency are limited by the on-NIC scheduler. This is because AlNiCo needs extra bandwidth to send the feature vector, and the off-path SmartNIC introduces additional latency.
Overhead of server feedback. The evaluation shows that AlNiCo uses only 1.2% of the CPU resources for the software feedback since the FPGA completes most computations for scheduling. However, the CPU-based AlNiCo (AlNiCo-CPU-N version) takes 27.7% of the CPU resources (i.e., scheduling overhead) to make scheduling decisions, which overshadows the scheduling benefits. This illustrates the need for using SmartNICs to reduce scheduling overhead.
The trade-off of feature vector length L. Table 3 shows the trade-off for choosing the feature vector length L, where the accuracy rate means *the proportion of keys detected as the contention that are truly contention*. We have two observations. First, a larger feature vector length L improves the accuracy rate because it reduces mapping collisions in features. Second, the scheduler IP core in AlNiCo requires more computation cycles and FPGA resources, and they increase linearly with

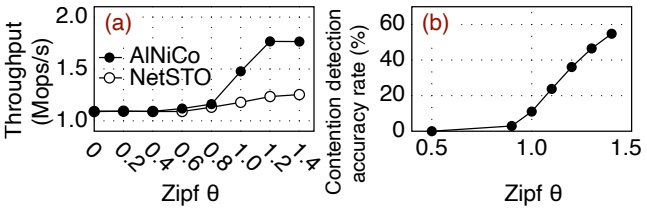

Figure 10: **YCSB-T (W:50) throughput (a) and contention detection accuracy rate (b) with varying θ.**

the feature vector length L. The theoretical maximum of L is limited by BRAM resources that are used to receive request feature vectors and store scheduler runtime states. We can sacrifice computation optimization or use on-NIC DDR4 (8GB in Innova-2) as storage resources to support a larger L.

Limitation. Based on all experiments in this section, we discuss the following two limitations. 1) AlNiCo can not improve the performance of uniform workloads. Figure 10 shows the throughput and the accuracy rate of contention detection with different θ under YCSB-T(W:50%). We observe that only when θ is higher than 0.8, the feature vectors can reflect the contention between requests. This is because, with the more skewed access pattern, the mapping collisions are fewer, and the weights of groups are more distinguishing. 2) AlNiCo can not improve the performance of the workloads whose throughput is limited by the NIC bandwidth. We focus on workloads whose throughput is limited by contention because AlNiCo consumes additional bandwidth for feature vectors.

6 Related Work

In-network scheduling. There have been intensive evolution efforts in application layer network scheduling, focusing on core affinity, load balance, head-of-line blocking, and in-network transaction coordination.

Core affinity. RSS [25] and FlowDirector [26] dispatch packets based on hashing header fields. MICA [23] uses RSS to assign single-key KV requests to cores based on the key hash partitioning for object-level core affinity. RSS++ [61] achieves dynamic load balance by RSS indirection and supports stateful flow migration by optimizing state transfers among cores. Different from them, AlNiCo focuses on contention-aware scheduling for transaction requests.

Load balance. Recent studies [62–64] achieve μs-scale SLOs through dynamic core scheduling or request scheduling. Humphries et al. [27] offload Shinjuku [64] to SmartNICs. RPCValet [28] and R2P2 [65] dispatch stateless RPC by emulating the theoretically optimal single-queue scheduling policy on NICs and programmable switches, respectively. RackSched [66] is a rack-level service scheduler with two-layer (i.e., inter/intra-server) scheduling.

Head-of-line blocking. In Minos [39] and DARC [40], KV requests for records of different sizes go to different cores to avoid blocking small requests by long-running requests.

In-network transaction coordination. Recent work offloads the transaction coordination to programmable switches [67–69] and client-side SmartNICs [70] to reduce the network overhead of distributed transactions. AlNiCo focuses on scheduling single-machine transactions to different CPU cores.

Transaction scheduling. Recent work for transaction processing can be categorized into inter-transaction scheduling and intra-transaction scheduling. AlNiCo focuses on inter-transaction scheduling, which schedules each entire transaction to the most appropriate CPU core.

Inter-transaction scheduling. The common principle of batching-based scheduling methods is to make each group almost conflict-free, and they differ in the approaches to residual conflicts. Calvin [71], LADS [19], and QueCC [18] keep track of the dependencies between transaction groups and wait for the completion of dependent transactions. Ding et al. [20] present a method that retries conflicting transactions at a higher priority in the next batch. Further, Jepsen et al. [15, 16] use a programmable switch to triage transactions belonging to different static data partitions before sending them to the database server.

Intra-transaction scheduling. A series of studies use strategies including tracking transaction dependencies [72], exploring the operation commutability [73], reading values from the write buffer [20], and releasing the lock in advance [74] to schedule each read/write operation. Polyjuice [75] uses machine learning models to specify different execution policies for each operation. Moreover, some studies focus on the order of locks; QURO [76] allows transactions that are more likely to block the system to hold locks, while Chiller [77] changes the lock order to reduce the locking time of hot records. Their intra-transaction scheduling is complementary to AlNiCo.

7 Conclusion

This paper presents AlNiCo, a transaction system that leverages SmartNICs to intelligently schedule incoming transaction requests to CPU cores, minimizing contention with low latency. AlNiCo describes the contention in a hardware-friendly manner so that specialized hardware can efficiently make scheduling decisions, and co-designs hardware and software to enable flexible and adaptive scheduling. Evaluation with real hardware (Innova-2) shows that AlNiCo reduces the contention between the running transactions and significantly improves performance.

Acknowledgements

We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback, which greatly improved this paper. We also thank Jian Gao, Minhui Xie, Jing Wang, Wenhao Lv, Xiaojian Liao,and Jian Chen for their suggestions. This work is funded by the National Natural Science Foundation of China (Grant No.62022051, 61832011), and Kuaishou.

References

[1] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[2] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[3] Mohammad Sadoghi and Spyros Blanas. Transaction processing on modern hardware. *Synthesis Lectures on Data Management*, 14(2):1–138, 2019.

[4] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Plor: General transactions with predictable, low tail latency. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD-/PODS '22, page 19–33, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[6] Standard Specification. TPC BENCHMARK^{TM} C. 1994.

[7] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming aborts in all phases for distributed In-Memory transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 217–232, Santa Clara, CA, February 2022. USENIX Association.

[8] Mellanox. Innova^{TM}-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf, 2020.

[9] Cisco. Cisco Nexus SmartNIC. https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html, 2021.

[10] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[11] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment*, 3(ARTICLE), 2010.

[12] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.

[13] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[14] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 430–441, New York, NY, USA, 2013. Association for Computing Machinery.

[15] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. 2021.

[16] Theo Jepsen. *Building blocks for leveraging in-network computing*. PhD thesis, Università della Svizzera italiana, 2020.

[17] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *EDBT*, pages 73–84, 2020.

[18] Thamir M Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*, pages 13–25, 2018.

[19] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2635–2650, 2016.

[20] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169–182, 2018.

[21] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[22] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment*, 13(5):629–642, 2020.

[23] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[24] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. Flexnic: Rethinking network dma. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 7, USA, 2015. USENIX Association.

[25] Intel. Receive-Side Scaling (RSS). http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf, 2007.

[26] Intel. Ethernet Flow Director. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf, 2014.

[27] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.

[28] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[29] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.

[30] Intel. Infrastructure Processing Units (IPUs). https://www.intel.com/content/www/us/en/products/network-io/smartnic.html, 2021.

[31] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, 2020.

[32] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[33] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[34] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[35] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.

[36] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous nic offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 18–35, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Zsolt István. Let's add transactions to fpga-based key-value stores! In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Zhaoshi Li, Leibo Liu, Yangdong Deng, Jiawei Wang, Zhiwei Liu, Shouyi Yin, and Shaojun Wei. Fpga-accelerated optimistic concurrency control for transactional memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 911–923, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, 2019.

[40] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing

tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Yihe Huang, Nathaniel Herman, William Qian, Jeevana Priya Inala, Eddie Kohler, Lillian Tsai, Barbara Liskov, and Liuba Shrira. STO: Software Transactional Objects. https://github.com/readablesystems/sto/, 2021.

[42] Mellanox. How To Implement PeerDirect Client using MLNX_OFED. PeerDirect, 2018.

[43] PCI-SIG. PCI-Express Specification. https://www.pcisig.com/specifications/pciexpress/, [n. d.].

[44] Wojciech M Zabołotny. Dma implementations for fpga-based data acquisition systems. In *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017*, volume 10445, pages 1269–1276. SPIE, 2017.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[46] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[47] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multicore in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

[48] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM sigplan notices*, 44(6):155–165, 2009.

[49] Dixin Tang, Hao Jiang, and Aaron J Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR*, volume 2, page 1. Citeseer, 2017.

[50] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[51] Masoud Hemmatpour, Bartolomeo Montrucchio, Maurizio Rebaudengo, and Mohammad Sadoghi. Analyzing in-memory nosql landscape. *IEEE Transactions on Knowledge and Data Engineering*, 34(4):1628–1643, 2022.

[52] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 773–785, USA, 2017. USENIX Association.

[53] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

[54] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 437–450, USA, 2016. USENIX Association.

[55] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage*, 16(4), oct 2020.

[57] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292. USENIX Association, February 2021.

[58] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD/PODS '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[60] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. *Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook*, page 209–224. USENIX Association, USA, 2020.

[61] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. RSS++ load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 318–333, 2019.

[62] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[63] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[64] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[65] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 863–880, 2019.

[66] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240, 2020.

[67] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 425–438, USA, 2016. USENIX Association.

[68] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, page 26–32, New York, NY, USA, 2018. Association for Computing Machinery.

[69] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 104–120, New York, NY, USA, 2017. Association for Computing Machinery.

[70] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[71] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[72] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.

[73] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, 2014.

[74] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking (extended version). *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021.

[75] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. *arXiv preprint arXiv:2105.10329*, 2021.

[76] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proceedings of the VLDB Endowment*, 11(5):648–662, 2018.

[77] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 511–526, 2020.