DINT: Fast In-Kernel Distributed Transactions with eBPF

Yang Zhou* Xingyu Xiang†* Matthew Kiley Sowmya Dharanipragada‡ Minlan Yu
Harvard University †*Peking University* ‡*Cornell University*

Abstract

Serializable distributed in-memory transactions are important building blocks for data center applications. To achieve high throughput and low latency, existing distributed transaction systems eschew the kernel networking stack and rely heavily on kernel-bypass networking techniques such as RDMA and DPDK. However, kernel-bypass networking techniques generally suffer from security, isolation, protection, maintainability, and debuggability issues, while the kernel networking stack supports these properties well, but performs poorly.

We present DINT, a kernel networking stack-based distributed transaction system that achieves kernel-bypass-like throughput and latency. To gain the performance back under the kernel stack, DINT offloads frequent-path transaction operations directly into the kernel via eBPF techniques without kernel modifications or customized kernel modules, avoiding most of the kernel stack overheads. DINT does not lose the good properties of the kernel stack, as eBPF is a kernel-native technique on modern OSes. On typical transaction workloads, DINT even achieves up to 2.6× higher throughput than using a DPDK-based kernel-bypass stack, while only adding at most 10%/16% average/99th-tail unloaded latency.

1 Introduction

Serializable distributed transactions are important programming abstractions and building blocks for distributed data center applications, such as object store and online transaction processing (OLTP) systems. With the advance of battery-backed DRAM [14] and fast NVRAM [10], the bottleneck of distributed in-memory transactions shifts from the storage to the networking. This has spurred extensive research on how to implement fast distributed in-memory transactions using kernel-bypass networking techniques, such as RDMA [14, 29, 82] and DPDK [6, 28]. One of the key assumptions for these works is that kernel-bypass is the key to realizing fast distributed in-memory transactions that match the underlying hardware speed.

However, kernel-bypass is not a panacea—it essentially trades security [69], isolation [38,39], protection [3,63], maintainability [53,79], and debuggability [70,79] for performance. In addition to these issues, kernel-bypass techniques such as DPDK usually burn one or more CPU cores for busy-polling even at low loads; this is usually non-acceptable in public cloud deployments due to per-core pricing [80]. These issues collectively have led to the well-known Open vSwitch giving up DPDK-based dataplane designs recently [79].

Instead, we choose to embrace the kernel networking stack with interrupt-driven packet processing. The kernel networking stack provides nice properties of good security, isolation, protection, maintainability, debuggability, and load-aware CPU scaling—but not performance. Its poor performance mainly comes from three sources: heavy-weight networking stack traversing [19, 89], user-kernel context switching [89], and interrupt handling.

This paper therefore asks: *can we remove such kernel stack overheads while keeping all of its nice properties for distributed in-memory transactions?* To this end, we follow a decade-old methodology called *extensible kernels* [4], and realize it in modern OS kernels without any kernel code modifications or customized kernel modules. The key enabler is the eBPF technique that allows users to run customized functions easily, safely, and efficiently inside the kernel networking stack at run time. With eBPF, we can run transaction processing logic at the early stage of the kernel networking datapath without going to the user space, avoiding most of the kernel networking stack functions and user-kernel context switching. For the overhead of interrupt handling, it could be amortized by adaptive batching [3] that the kernel networking stack NAPI [34] already did. Besides the potential performance benefit, eBPF is a kernel-native technique shipped with and well-maintained by each release of modern Linux kernels. Due to its safety and kernel-native nature, it has been rapidly adopted by applications and cloud vendors [2, 16, 55]. For example, Meta runs over 40 eBPF programs on every server with ∼100 loaded on demand [75].

We introduce DINT[1], which accelerates distributed transaction systems using eBPF. DINT handles as many transactions as possible in the kernel to improve their critical path performance. In distributed transaction systems, a transaction usually involves three components in its critical path: it first acquires various locks from a lock manager, then reads relevant key-values from a key-value store and does local updates, next logs key-value updates to a log manager, and finally com-

*Equal contribution

[1] DINT as a noun is an archaic word, meaning force and power.

mits key-value updates to the key-value store. Offloading the three components to eBPF is challenging because *eBPF has a constrained programming model (for kernel safety)*.

To address this challenge, our key idea is to redesign transaction-related data structures following the principle of *kernel-offloading* for frequent critical paths to guarantee high performance, and use *user space programs as backups* for rare paths to support full functionalities.

First, the lock manager normally maintains many locks with efficient indexing and complex locking operations. However, it is hard for eBPF to handle hash collision during indexing, because eBPF only allows statically-bounded loops. Further, it is also hard to maintain shared lock states because eBPF does not support common synchronization primitives like Mutex. To address these issues, the DINT lock manager embraces lock sharing to avoid the slow and complex hash collision handling, and directly leverages low-level eBPF atomics to implement transaction locking.

Second, the key-value store normally stores a large number of key-values with different sizes, and requires frequent lookups and updates. However, eBPF does not support dynamic memory allocations, causing low memory efficiency for the key-values. To address these issues, the DINT key-value store directly stores small key-values, which dominate in transaction workloads [12, 47, 77], in kernel memory using a set-associative cache, while leaving large key-values to the user space, avoiding dynamic memory allocations in eBPF. DINT further designs a write-back mechanism with Bloom filters [5] to efficiently handle most key-value lookups and updates in the kernel, while guaranteeing the key-value consistency across the user and kernel.

Third, for the log manager, DINT designs efficient per-CPU log buffers to record logs directly in eBPF, while supporting log replaying from the user space during failure recovery.

We evaluate DINT on two OLTP workloads: a read-intensive TATP workload [47] and a write-intensive SmallBank workload [77]. DINT achieves up to 2.6× higher throughput than using a recent well-engineered kernel-bypass stack based on DPDK (i.e., Caladan [17]), while only adding at most 10% and 16% unloaded latency for the average and 99th-tail respectively. We achieve even higher throughput mainly because the kernel-bypass baseline builds a high-level abstraction for packets that incurs packet copy overhead between network buffers and application buffers, while DINT directly modifies incoming packets and forwards back. DINT's designs are also generic to transaction protocols to some extent—it easily supports an OCC (opportunistic concurrency control) protocol for the read-intensive workload and a 2PL (two-phase locking) protocol for the write-intensive workload.

In summary, this paper makes three contributions:

- We design and implement a high-performance distributed transaction system under the widely-deployed kernel networking stack and the widely-available common commodity NICs, with the key idea of kernel offloading via eBPF.

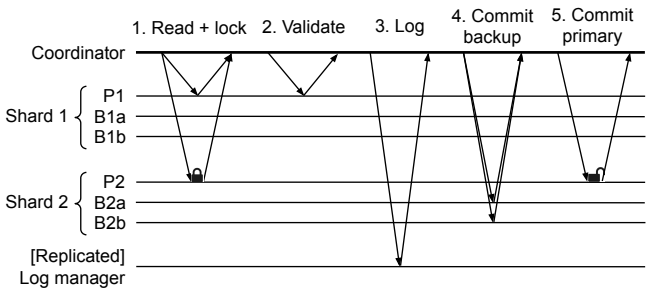


Figure 1: The FaSST [29] transaction protocol with two data shards and three-way replication. P = primary and B = backup. This example transaction reads from the shard 1 and writes to the shard 2.

- We design a state synchronization mechanism for the key-value cache across the kernel and user space that efficiently handles consistency and write-backs.
- We are the first to experimentally show that a distributed transaction system under the kernel networking stack can achieve kernel-bypass-like performance and latency.

DINT has some limitation: it currently targets UDP unreliable transport protocol to simplify packet processing in eBPF. Some research work [76] on designing offload-friendly reliable transports might help address our limitation.

2 Background

2.1 Distributed Transactions

We focus on serializable distributed transactions over a replicated sharded in-memory key-value store with replicated logging to handle failures. Along with recent works [14, 29, 72, 82] in this space, we assume logging into fast persistent storage like battery-backed DRAM or NVRAM (instead of disks) to match in-memory transaction speed, and having a separate fault-tolerance configuration manager to handle machine failures off the critical path of transaction processing. These works usually employ transaction protocols consisting of optimistic concurrency control (OCC) and two-phase commit for distributed atomic commit, and primary-backup replication to support high availability. Below, we briefly go through the critical path of one of such protocols from FaSST [29].

In the FaSST transaction protocol, each transaction has a set of keys to read (i.e., read-set) and a set of key-values to write (i.e., write-set), and a transaction coordinator issues transaction requests to finish each transaction. As shown in Figure 1, the primary in each shard runs a *lock manager*; both the primary and backups run a replicated *key-value store*; a set of servers run a replicated *log manager* (could just be on the primary and backups). To finish a transaction, the transaction coordinator executes the following phases:

1) **Read+lock:** the coordinator reads all values + locks + versions for the read-set and locks all key-values for the write-set. If any key-value in the two sets is already locked, the transaction aborts. The coordinator buffers key-value writes/updates locally.
2) **Validate:** the coordinator reads again all locks + versions

in the read-set, and checks if any read-set value has been changed or locked since the first phase. If so, the transaction aborts.

3) **Log:** the coordinator writes a transaction record containing the write-set's key-values and their versions into the replicated log manager.
4) **Commit backup:** the coordinator updates the write-set values to corresponding backup replicas.
5) **Commit primary:** the coordinator updates the write-set values to corresponding primary replicas, increments key-value versions, and unlocks key-values.

Besides the OCC, there are many more concurrency control protocols for serializable distributed transactions. Another well-known one is two-phase locking (2PL) used in Spanner [9]; it locks before each read and write, and is suitable for write-intensive workloads. More advanced protocols include MDCC [40], Tapir [87], Janus [58], ROCOCO [57], which reduces the number of transaction phases by co-designing concurrency control and replication, and allows more concurrency by tracking fine-grained transaction dependencies.

Distributed transactions inside a data center typically have bottlenecks on the networking stack. For example, when we run the above transaction protocol using a typical OLTP workload under the kernel UDP stack (see §5.2 for a detailed setup), we observe 64% of CPU time is spent on traversing the kernel networking stack, 16% is on the user-kernel context switching, and 12% is on the interrupt handling. This further motivates the huge performance benefits of kernel offloading by avoiding kernel stack overheads.

2.2 eBPF in Kernel Networking Stack

eBPF basics: eBPF (extended Berkeley Packet Filter) is a kernel-native mechanism to let users write *safe, customized* programs that run inside the OS kernel without kernel code modifications or customized kernel modules. Users typically write a high-level C-like eBPF program that gets compiled into low-level eBPF bytecode by Clang/LLVM. Users can then load the eBPF bytecode to predefined attachment points or the so-called *eBPF hooks* in the kernel. Upon loading, the kernel will first verify if the eBPF bytecode meets the safety (e.g., no out-of-bounds memory accesses) and liveness (i.e., it will always terminate in finite steps) requirements. If so, the kernel will compile the eBPF bytecode to native machine code, and run it in a kernel-embedded virtual machine in an event-driven manner; otherwise, the kernel will reject it.

The Linux kernel networking stack has two main eBPF hooks: XDP (eXpress Data Path) [21, 67] and TC (Traffic Control) [50]. The XDP hook only works for ingress packets, and triggers the eBPF program immediately after the NIC driver receives the packet upon NIC interrupts, before `sk_buff` [35] creation. The TC hook works for both ingress and egress packets, and triggers the eBPF program between the NIC driver layer and UDP/TCP layers. For ethernet packet forwarding, TC has lower performance than XDP, as it has

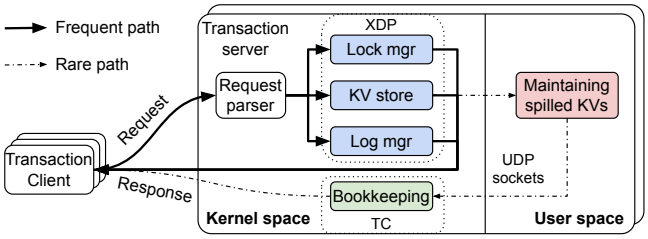


Figure 2: DINT's high-level architecture.

run more kernel networking stack functions.

eBPF maps: eBPF programs are event-driven, therefore program states that cross different invocations must be stored in a global heap-like memory region—eBPF maps are exactly for this purpose. eBPF maps are a variety of built-in data structures in the kernel to maintain eBPF program states with various eBPF helper functions. An eBPF map could contain up to $2^{32}-1$ elements each with maximum $2^{32}-1$ bytes, with total size bounded by the server memory; it must be declared and created statically with a fixed size. Typical eBPF maps include arrays, per-CPU arrays, stacks, and queues [49], with lookup and update functions [32]. The power of eBPF maps is that they can be shared among different eBPF programs and user-space processes. For example, the eBPF program attached to XDP can share an eBPF map with another program on TC and even with a user-space process.

eBPF programming constraints: Due to the safety and liveness verification by the kernel, eBPF programming has some constraints. Perhaps the most important one is not supporting dynamic memory allocations, as correctly handling memory allocation failure and verifying no memory leaks are challenging for eBPF. The second constraint is that eBPF only supports statically-determined bounded loops to ensure liveness. Finally, eBPF lacks high-level thread synchronization primitives such as Mutex. This is because eBPF code runs inside the kernel, and arbitrary/unexpected kernel sleeping by Mutex is dangerous. Instead, eBPF only supports spinlock (i.e., `bpf_spin_lock` [48]) with deep constraints that make it less useful: one cannot call any functions (including built-in eBPF helper functions) while holding the lock, and must release the lock before forwarding/dropping the packet.

3 DINT Design

Figure 2 shows the high-level architecture of DINT. DINT assumes an asymmetric transaction model or the so-called client-side transaction model, similar to [42, 56, 60, 87]. In this model, each transaction client, as the transaction coordinator, sends transaction requests to transaction servers to finish locking, key-value, and logging operations, and then receives responses. As described in Section 6, DINT could also support the symmetric model used in [14, 29, 82]. Like prior works, DINT shards transactions states (i.e., locks, key-values, and logs) among servers, and uses three-way replication and logging for high availability. DINT is generic to a variety

of transaction protocols, and currently supports two different ones: a 2PL-based protocol and an OCC-based protocol similar to FaSST [29].

Offloading request frequent path to kernel: To achieve high-performance transaction processing, DINT offloads frequent-path states and operations into the kernel, avoiding kernel stack overheads. Each DINT transaction server maintains *most of* its transaction states in the kernel memory via eBPF maps, and serves *most of* its transaction requests directly in the kernel via an eBPF program attached to the XDP hook. Since eBPF programs cannot generate new packets by themselves, DINT reuses the request packet by modifying its payload to carry the response message, and forwards it back to clients as the response.

Userspace as backups: To support the full functionalities of transaction processing, DINT handles rare-path states and operations in the user space. Each DINT transaction server runs a user-space process listening on UDP sockets to receive and handle *a small portion* of transaction requests that cannot be served directly in the kernel. Transaction responses returned from the user-space process will go through a bookkeeping eBPF program attached to the TC hook, which helps maintain transaction states in eBPF maps, e.g., releasing some internal locks (not transaction locks).

DINT uses UDP protocol between transaction clients and servers to allow easy parsing of transaction requests and responses in eBPF programs. While UDP protocol is lossy, packet losses happen rarely in modern data centers as shown by prior works [28, 29, 65]. When packet losses happen during severe network hardware failures, DINT would detect such losses using coarse-grained client-side timeouts and handle them by the transaction protocols, similar to FaSST [29]. DINT targets accelerating the handling of transaction requests/responses that can fit into one ethernet packet, i.e., up to 9KB for jumbo frames. This works well for transactions with mostly small key-values, which are quite common in many transaction processing workloads [12, 29, 47, 77, 82]. For large key-values, DINT could just pass them to the user-space process to handle, at the cost of lower throughput.

3.1 DINT Lock Manager

The DINT lock manager is responsible for the transaction concurrency control, i.e., controlling how multiple transaction clients concurrently access individual key-values. Such concurrency control mainly involves quickly indexing lock states by lock IDs and maintaining the shared lock states. These two operations are challenging for the constrained programming model in eBPF that lacks dynamic memory allocations, only supports bounded loops, and has nearly no high-level thread synchronization primitives like Mutex (§2.2). For example, lock state indexing usually requires implementing a hash table in eBPF; however, handling hash collisions is nearly impossible or very inefficient in eBPF for either open hashing that requires dynamically allocating a new hash table entry or closed hashing that may require unbounded loops.

To support efficient lock state indexing and shared lock state maintenance in eBPF, DINT leverages two techniques:

- lock sharing to avoid handling hash collisions. Lock sharing means two lock IDs may get mapped to and use the same lock state. DINT further designs a mechanism to avoid possible deadlocks during lock sharing.
- leveraging low-level eBPF atomics [23] to carefully synchronize shared states operations.

Lock sharing: DINT leverages eBPF array maps (i.e., `BPF_MAP_TYPE_ARRAY` [32]) to implement static tables of lock states in the kernel space. Typical lock states include lock status bits, sharer counters (for read-write locks), etc. Each lock ID gets mapped to one shared lock state in the table via a hash function, and later lock acquiring/releasing operations just work on this lock state. Lock sharing avoids handling tricky hash collisions, at the cost of slightly increasing the failure probability when acquiring locks.

However, deadlocks could happen if a transaction client tries to acquire two locks that get mapped to the same lock state (assuming exclusive locking). This is because: the first acquiring operation succeeds, while the second acquiring fails/blocks; however, the first acquiring will not release the lock until the transaction finishes, while the second acquiring always blocks the transaction progress. To resolve such possible deadlocks, DINT lets the lock manager check if any two exclusive lock acquiring operations on the same lock state come from the same transaction client, by maintaining a holder client ID (e.g., IP and port pair) for each exclusive lock; if so, the lock manager directly returns a locking success message.

By leveraging low-level eBPF atomics, DINT supports a variety of locking mechanisms for concurrency control protocols, including the basic read-write locking for 2PL and version-based locking for OCC, in a fail-and-retry manner [8, 18, 83]. Supporting more advanced concurrency control protocols [40, 57, 58, 87] is also possible in DINT, as they are essentially underpinned by the two basic locking mechanisms; we discuss further in Section 6.

Read-write locking: This locking mechanism includes two types of locks: exclusive locks and shared locks. Transaction clients send lock acquiring/releasing requests with lock IDs to the lock manager, and the manager responds with either success or failure. Lock acquiring requests may receive failure responses, while lock releasing requests always receive success responses. If a client receives a failure response, it will re-send the lock acquiring request again after an optional period of time, until receiving the success response (i.e., fail-and-retry).

To implement the read-write locking, the DINT log manager maintains a per-lock spinlock bit, a per-lock counter that counts how many sharers hold the lock, and a per-lock status bit indicating if this lock is held exclusively. Upon receiving an exclusive lock acquiring request, the lock manager looks up the corresponding spinlock bit and executes eBPF

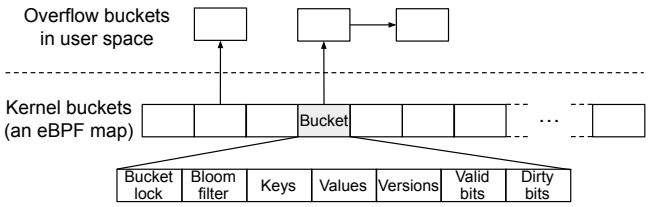


Figure 3: The layout of the key-value store in DINT (assume using the version-based locking).

atomics to check if it can acquire the spinlock. It runs the `__sync_val_compare_and_swap()` function inside eBPF to atomically test-and-set the spinlock bit. This function gets compiled into corresponding ISA-specific operations and is equally efficient as in the user space. If the lock test-and-set succeeds, which means the lock state is not being modified by other transaction clients, the load manager will check the exclusiveness status bit—if the bit is clear, it will set the bit and return a success response; in any other cases, a failure response is returned to let the client retry. Handling the exclusive lock releasing and shared lock operations involves similar atomic operations.

Version-based locking: Version-based locking is widely used in recent high-performance distributed transactions systems [14, 29, 82], together with the OCC protocol to avoid locking operations for key-value reads. It involves version checking to make sure the read key-values used in a transaction are not stale (see §2.1).

To implement version-based locking, DINT maintains a table for the lock status bit indexed by the lock ID, and maintains a per-key-value version counter in a key-value store that we discuss in the next Section. Every read operation directly reads the key-value and corresponding version from the key-value store. Every write operation tries to test-and-set the lock status bit (i.e., exclusive lock); if test-and-set fails, the transaction aborts. After acquiring all write locks and then finishing all transaction writes locally, the transaction coordinator reads the key-value versions again and compares them with the old versions. If the two version vectors do not change, the coordinator can successfully log and commit the transaction, and increment the versions; otherwise, the transaction aborts.

3.2 DINT Key-Value Store

The DINT key-value store maintains the mapping between keys and values, and supports various operations like GET, INSERT, UPDATE, and DELETE. Conventional user-space key-value store [54, 71] would normally maintain a hash index that maps keys to dynamically-allocated values. Unfortunately, this design does not work for eBPF that lacks dynamic memory allocations.

Figure 3 illustrates how DINT addresses this challenge by storing key-values into an in-kernel set-associative cache backed by a fixed-size eBPF map, while spilling overflowed key-values (includes corresponding versions) into the user space. The eBPF map contains many kernel buckets indexed by the key via a hash function, and each bucket stores multiple key-values and valid bits (denoting whether a key-value field stores object data)[2]. By default, DINT stores 4 key-values per kernel bucket. Inside each kernel bucket, DINT stores keys contiguously for better cache locality during lookups; DINT provisions each value field with a fixed size that can cover most of the transaction objects (e.g., dozens of bytes in TPC-C and SmallBank workloads [82, Table 3]). Any kernel bucket that gets too many key-values will spill some key-values into the user space (putting into the overflow buckets); any key-value that cannot fit into the fixed value field in the kernel bucket will also spill into the user space.

A kernel bucket contains a bucket-level lock implemented using eBPF atomics to synchronize concurrent key-value operations on the same bucket. We note that this lock is different from the transaction locks in Section 3.1. Each key-value operation will first try acquiring the bucket lock before touching the bucket data, in a fail-and-retry manner. Most of the time, the key-value operation finishes directly in eBPF and returns the response to clients, before which it releases the bucket lock. In rare cases where its interested key-value is in the user space, the operation needs to pass the operation request/packet to the user-space process via the UDP sockets. Under such cases, the operation still holds the bucket lock when going to the user space, and only releases the lock when it returns to eBPF. By "returns to eBPF", we mean that the response packet sent back by the user-space process will trigger an eBPF program attached to the TC egress hook, which releases the bucket lock.

However, to support high-performance key-value operations in this kernel-user-hybrid key-value store, we must address two additional challenges:

- How to efficiently perform INSERT and UPDATE operations while maintaining *read-all-write* consistency? Prior eBPF-offloaded key-value store BMC [19] adopts a simple write-through cache design and performs well when all operations are GETs. However, in workloads like TATP [47] where 20% of transactions involve INSERTs/UPDATEs, BMC would perform poorly because every such operation will go to the user space.
- How to minimize the chance of going to the user space, especially when clients issue many GET requests for non-existing keys? Non-existing key lookups would require enumerating all keys mapped to the kernel bucket including those spilled into the user space, incurring high kernel stack overheads. Such lookups are common in transaction workloads; e.g., 68.75% of GETs for TATP's largest table target non-existing keys.

To this end, DINT designs a write-back mechanism that lazily and efficiently maintains the read-after-write consistency, and leverages a per-kernel-bucket Bloom filter to avoid frequently going to the user space for non-existing key lookups.

[2]Maintaining the valid bit for each key-value should be straightforward; for conciseness, we do not explicitly describe it unless necessary.

3.2.1 Write-Back Key-Value Store in eBPF

As shown in Figure 3, a kernel bucket contains a dirty bit for each stored key-value, indicating whether the value is different from the user space; a key-value that only exists in eBPF will always have the dirty bit set. Below, we go through how DINT efficiently realizes each of the key-value operations across eBPF and the user space. A recurring theme in the design of each operation is that: DINT tries to support the majority of key-value operations directly in eBPF by leveraging the dirty bit, while maintaining consistency.

GET (Figure 4a): For simplicity, we assume the looked-up key exists in the key-value store; we describe the non-existing case in the next Section. In the frequent path (a) where the GET operation finds the key in the kernel bucket, DINT directly returns the requested value to the client. In the rare path (b) where the GET operation does not find the key: if the kernel bucket is full, DINT chooses one existing key-value to evict following a certain policy (described later) to make a space for the looked-up key-value; otherwise, DINT chooses one dirty key-value (if any) for lazily writing back to the user space. DINT then optionally piggybacks the chosen key-value on the packet and forwards it to the user-space process; DINT uses the `bpf_xdp_adjust_tail()` function [48] to increase the packet size for piggybacking. Once the process receives the request packet, it will look up the key-value in the overflow buckets, and send back the response packet to the client via the UDP sockets. For the piggybacked key-value, the user-space process will update it into the overflow buckets. Finally, the response packet goes through the TC egress eBPF program, which clears the dirty bit of the piggybacked key-value (if any), and fills the requested key-value into an empty or non-dirty key-value field in the kernel bucket.

INSERT (Figure 4b): For simplicity, we assume the to-be-inserted or the incoming key-value does not exist in the key-value store; if it already exists, we could just return a insertion failure message to the client. In the frequent path (a) where the INSERT operation finds an empty slot in the kernel bucket, DINT directly writes the incoming key-value there, sets an initial version and the dirty bit, and returns to the client. In the rare path (b) where there is no empty slot in the kernel bucket, DINT chooses a key-value to evict. Then there will be two cases:

- If the to-be-evicted key-value is not dirty, DINT will directly replace it by the incoming key-value with a *set dirty bit* and an initial version, and return to the client. DINT can directly return as the to-be-evicted key-value has the same copy in the user space, so there is no need to write it back. Since the incoming key-value is marked dirty, a later eviction will lazily write it back to the user space.
- If the to-be-evicted key-value is dirty, DINT will piggyback it on the request packet, replace the bucket's to-be-evicted key-value by the incoming key-value with a *clear dirty bit* and an initial version, then pass the request packet to the user space. The user-space process will then update both the evicted key-value and the incoming key-value into the overflow buckets, and send back a response packet to the client via the UDP sockets.

UPDATE (Figure 4c): For simplicity, we assume the to-be-updated or the incoming key-value exists in the key-value store; if it does not exist, we could just return an update failure message to the client. In the frequent path (a) where the key-value is found in the kernel bucket, DINT directly updates the key-value there with a set dirty bit, increments the version counter, and returns to the client[3]. In the rare path (b) where the key-value is not found, DINT chooses a key-value to evict. No matter whether the to-be-evicted key-value is dirty or not, DINT always needs to go to the user space, in order to fetch (and increment) the version counter corresponding to the incoming key-value. Therefore, DINT will piggyback the to-be-evicted key-value on the request packet, replace the bucket's to-be-evicted key-value by the incoming key-value with a clear dirty bit and an *undefined version*, then pass the request packet to the user space. The user-space process will then update both the evicted key-value and the incoming key-value into the overflow buckets, increment the version counter of the incoming key-value, and send back a response packet to the client via the UDP sockets. More importantly, this response packet will piggyback the updated version of the incoming key-value, so that the TC egress eBPF program can update the bucket's undefined version to the updated one.

Eviction policy: DINT currently uses a simple eviction policy: it tries to evict the first non-dirty key-value when enumerating the bucket; if all the key-values are dirty, it then evicts a random key-value. Prioritizing evicting non-dirty key-value avoids going to the user space as much as possible (especially under INSERT operations). Randomly choosing a key-value if all is dirty minimizes the compute for selecting a victim key-value. Implementing a more complex eviction policy, e.g., based on key-value accessing frequency, might help further reduce the chance of going to the space. But such policy should be compute-light; otherwise, it may incur performance drops [19]. Recent fast cache eviction algorithms such as QD-LP-FIFO [84] may shed light on this space, and we leave such exploration as future work.

Remark: So far, DINT carefully leverages the dirty bits to run the majority of key-value operations directly in eBPF, while maintaining read-all-write consistency and correct key-value versions. For the DELETE operation, we deliberately leave it for the next Section to describe, as it is highly related to how we efficiently handle non-existing key lookups.

3.2.2 Handling GETs for Non-Existing Keys

So far, in the DINT key-value store, GET requests for non-existing keys would enumerate all keys mapped to the indexed kernel bucket including those spilled into the user space, incurring high kernel stack overheads. Conventional key-value

[3] If the new value is too large for the kernel bucket, we will evict the key-value to the user space.

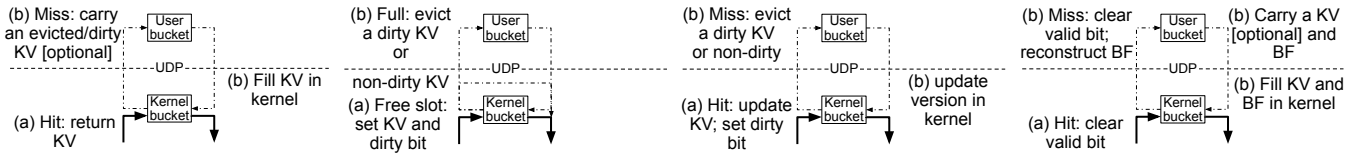


Figure 4: DINT key-value store operations. Solid thick lines indicate frequent paths, while dotted thin lines mean rare paths. BF = Bloom Filter.

stores implemented in the user space do not have such a problem because the enumeration overhead for them is only a few more memory accesses; however, for the key-value store in eBPF, the overhead escalates into expensive kernel networking stack traversing and user-kernel context switching [19,89].

To handle non-existing key GETs efficiently, DINT maintains a small Bloom filter [5] in each kernel bucket, representing the membership of key-values spilled into the user space (Figure 3). The Bloom filter is updated whenever a key-value gets spilled into the user space. When a GET operation does not find the looked-up key in its kernel bucket, it looks up the Bloom filter to check if the key possibly exists in the user space. If the Bloom filter answers no, then the GET operation can guarantee that the key does not exist in the key-value store, and directly return none to the client; otherwise, the operation must go to the user space to check the overflow buckets (see §3.2.1). Since the Bloom filter never reports an existing key as non-existing (i.e., no false negative errors), the above "early returning" in the GET operation is always correct. DINT currently provisions 64 bits for the Bloom filter in each kernel bucket, sufficient to handle dozens of spilled key-values. To reduce the hash calculation overhead for the Bloom filter, DINT reuses the highest six bits of the raw hash value from the key-value store. We choose to implement our custom Bloom filter instead of using the built-in Bloom filter eBPF map [33], in order to avoid extra eBPF map lookup overhead.

However, the Bloom filter design creates a challenge for the key-value DELETE operation. This is because: when the to-be-deleted key-value is in the user space, the DELETE operation will need to remove the key-value from the Bloom filter; however, the Bloom filter does not support membership removal in order to guarantee no false negative errors. To address this challenge, DINT lets the user-space process reconstruct a new Bloom filter for the remaining key-values whenever it deletes one, and then updates the new Bloom filter to the kernel. Reconstructing the Bloom filter is doable, as the user space records all the spilled key-values in its overflow buckets. Formally, the DELETE operation works as follows.

DELETE (Figure 4d): For simplicity, we assume the to-be-deleted key-value exists in the key-value store. In the frequent path (a) where the INSERT operation finds the key-value in the kernel bucket, it clears the valid bit and directly returns to the client. In the rare path (b) where the key-value is not found in the kernel bucket and the Bloom filter reports its existence in the user space, the DELETE operation must forward the request packet to the user space. The user-space process will look up the key-value in the overflow buckets, clear its valid bit, reconstruct a new Bloom filter based on the remaining spilled key-values, and send back a response packet to the client. The response packet will piggyback the new Bloom filter and an optional spilled key-value (if existing, and this key-value should not be covered in the new Bloom filter), and trigger the TC egress eBPF program, which fills the Bloom filter and key-value into the kernel bucket.

3.3 DINT Log Manager

High-performance distributed transaction systems store transaction logs in memory for failure recovery (assuming battery-backed DRAM or fast NVRAM [14,29]). The transactions logs grow up as the transaction systems run: if they exceed the log space (e.g., memory capacity of the machine), the transaction systems usually truncate the oldest logs [14] or dump them into disks [78]; DINT follows the truncating manner. Since the logging operation is on the transaction critical path, DINT aims to provide a fast logging mechanism entirely inside the eBPF in failure-free cases, while supporting complex offline recovery in failure cases.

To this end, DINT leverages the eBPF maps to implement a circular log buffer abstraction entirely in the kernel. A circular log buffer allows pushing log entries to the tail to support logging operations in transaction systems; it also allows popping log entries from the head (from the user space) to support log replaying during failure recovery. DINT implements such a circular log buffer using a large-size eBPF array map to store log entries, and another eBPF array map to maintain the head and tail, both inside the kernel. These two eBPF maps are also accessible to the user space for log replaying. To avoid thread contentions during logging operations, DINT provisions a circular log buffer on each CPU core. This is achieved by using the eBPF per-CPU array map (`BPF_MAP_TYPE_PERCPU_ARRAY` [32]). When the log manager looks up a per-CPU array map, it will automatically get the map entry corresponding to its local CPU core.

4 DINT Implementation

Our DINT prototype consists of 2.1K lines of eBPF (for kernel code) and 4.3K lines of C++ (for user-space code). DINT uses Clang/LLVM-16 to compile the eBPF program into eBPF bytecode. The eBPF bytecode gets attached to and runs inside the XDP and TC hooks of the standard kernel networking stack, atop unmodified Linux OSes. The user-space process

uses the standard POSIX kernel-visible threads (i.e., pthreads) and the Linux UDP socket to receive rare-path request packets and send response packets. Our prototype currently supports two different transaction protocols, i.e., a 2PL-based protocol and an OCC-based protocol, demonstrating the genericity of DINT's designs to some extent. Our DINT prototype currently does not implement failure recovery to handle machine failures; as described in Section 2.1, we assume a separate configuration manager would handle them off the critical path, thus not impacting the critical-path performance we focus on.

To reduce the performance impact of user-kernel context switching when passing request packets to the user space, DINT runs the user-space process (that handles rare-path requests) on CPU cores that do not receive NIC interrupts or run eBPF programs, similar to prior work [89]. This is achieved by configuring the IRQ affinity of the NIC device to exclude the rare-path handling core. Note that the rare-path handling core does not do any busy polling and can be shared with other applications.

To better reason about the performance of DINT, we build two baseline transaction processing systems that run in the user space. One baseline uses the standard kernel UDP socket with `SO_REUSEPORT` enabled to reduce thread contentions [19], and pthreads. Another baseline uses the UDP stack from the kernel-bypass runtime Caladan [17] that supports DPDK-style packet busy-polling and user-space threading for fast context switching. Both baselines leverage DINT's performance optimizations (e.g., lock sharing) if helpful, but without eBPF programming constraints—so that they can handle hash collisions efficiently using state-of-the-art solutions [44]. The two baselines consist of 6.1K lines of C++.

5 Evaluation

This section aims to answer the following questions:

1. What is the throughput and latency of DINT compared to kernel-bypass approaches (§5.1 and §5.2)?
2. Can DINT support different transaction protocols on transaction workloads efficiently (§5.1 and §5.2)?
3. Can DINT provide load-aware CPU scaling (§5.3)?
4. What are the effects of the write-back mechanism, Bloom filter, and rare paths on DINT's performance (§5.6)?

Testbed: We use 13 `r650` physical machines from CloudLab [15]. Each machine has two 36-core (72 logic-core) Intel Xeon Platinum 8360Y CPUs at 2.4GHz, 256GB memory, and a dual-port Mellanox ConnectX-6 100Gb NIC via PCIe 4.0×16. All machines are connected via a Dell Z9432F switch under the same rack. For all experiments, we use a single CPU in the same NUMA domain as the NIC to enforce NUMA locality; we also use a single 100Gb NIC port, as CloudLab currently only wires one such port of `r650` to the switch.

For all experiments, each machine runs an unmodified Ubuntu 20.04 OS. For eBPF and UDP-related experiments, we use kernel v6.1.0 which has full support for eBPF atomics. We use the built-in Mellanox NIC driver on Linux kernel v6.1.0 that has a default NAPI poll budget/batch size of 64 upon each interrupt. We disable Mellanox NIC's interrupt coalescing feature [11], as we find it hurts latency while not increasing throughput, similar to prior work [89]. For Caladan-related experiments, we are not able to run the Caladan runtime on kernel v6.1.0, as it requires a customized kernel module that relies on specific kernels; instead, we manage to run it on kernel v5.8.0. Since Caladan uses the kernel-bypass networking stack and threading, different OS kernels should not have a significant impact on its performance.

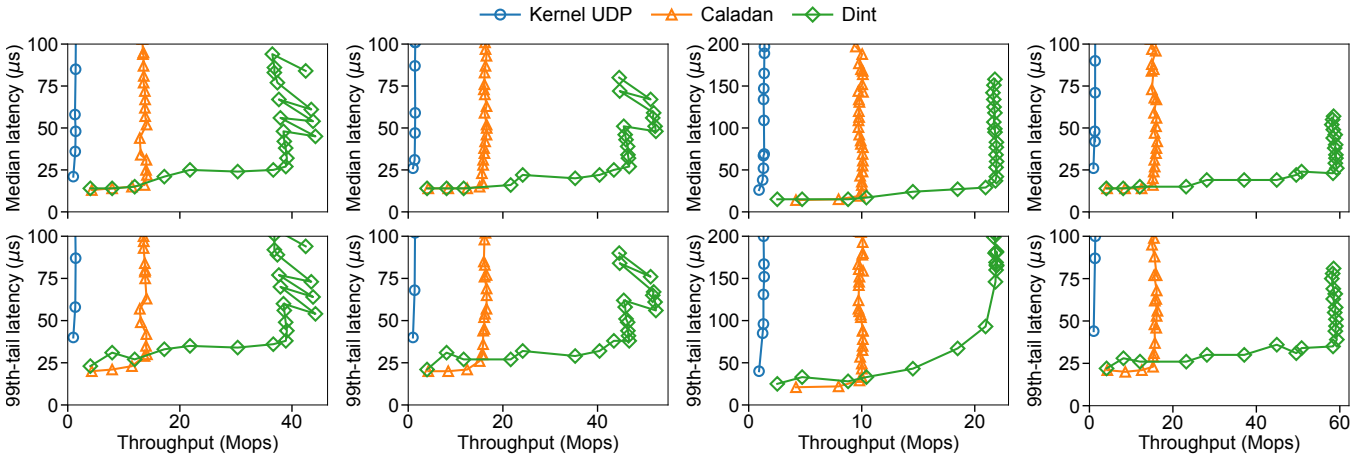
Measurement methodology: For transaction benchmarking, we use 3 machines to run transaction servers with three-way replication and sharding; that is, each machine is the primary for one shard and a replica for the other two. For microbenchmarks that benchmark individual lock manager, key-value store, and log manager, we use 1 machine to run the microbenchmark server without replication or sharding to understand their standalone performance. We use the rest machines to run multiple transaction/microbenchmark clients that issue requests in a closed-loop manner. To avoid the client machines becoming the bottleneck, we provision 8 cores on each transaction/microbenchmark server; the client machines further use Caladan's kernel-bypass UDP stack and user-space threading to generate requests. We then vary the number of clients, and measure the achieved throughput and client-perceived median/average and 99th-tail latency, similar to prior transaction works [6, 42, 56, 87].

Comparison baselines: As mentioned in Section 4, we compare DINT to two baseline transaction processing systems: one is based on the Linux kernel UDP socket, another is based on the UDP stack from the kernel-bypass runtime Caladan [17]. For simplicity, we just use kernel UDP and Caladan to refer to these two baselines respectively. The Caladan baseline is a challenging baseline that features DPDK-style packet busy-polling, NIC RSS to evenly spread packets among available cores, and well-implemented and efficient user-space UDP stack and threading.

We provision the memory sizes of the user-space key-value store (for the two baselines), eBPF key-value store (for DINT), and lock table (for all three) to be 1.5× of the key-values/locks in corresponding workloads, similar to FaSST [29]. By default, kernel UDP and Caladan use all provisioned cores to handle requests, while Caladan uses one extra core to run its scheduler. DINT devotes one core out of the provisioned cores to handle rare-path requests (§4), while the rest cores handle frequent-path requests.

5.1 Microbenchmarks

To understand how each DINT component compares to baselines, we implement a series of microbenchmarks, including a 2PL-based and an OCC-based lock manager with skewed locking requests (80% shared locking requests or version reads), a key-value store with 40B skewed reads, and a log manager with 56B writes. These microbenchmark parameters



(a) 2PL lock manager. **(b)** OCC lock manager. **(c)** Key-value store. **(d)** Log manager.

Figure 5: Microbenchmark load-latency curves (both median and 99th-tail).

(e.g., skewness, value size) are derived from the TATP workload [47]. The two lock managers and the key-value store are provisioned with 36 million lock/key slots, while their requests target 24 million locks/keys.

Lock manager: Figure 5a and 5b show how the latencies (both median and 99th-tail) of the 2PL and OCC lock manager vary with different achieved throughput for different systems, respectively. Each system performs similarly across the two lock managers with the OCC lock manager being slightly faster, as version reads in OCC do not run atomic operations. Overall, DINT achieves 3.1×-3.2× higher throughput than Caladan, with 0%-8%/5%-55% higher unloaded median/99th-tail latency, while kernel UDP performs much worse than others. We notice that DINT has throughput fluctuations at high loads; we think this is because the achieved batch size during interrupt handling gets changed unstably.

It might be supersizing that DINT achieves even higher throughput than the kernel-bypass Caladan system. However, this is achievable, as Caladan wraps raw UDP packets into a high-level connection-oriented abstraction (i.e., `rt::UdpConn`) for applications, which incurs packet copy overhead between network buffers and application buffers, thus losing some performance, while DINT directly works on low-level UDP/ethernet packets. Additionally, each Caladan transaction server creates a `rt::UdpConn` for each transaction client and spawns a user-space thread to handle corresponding transaction requests. Although `rt::UdpConn` only maintains simple connection states with small packet copy and user-space threading is efficient (e.g., 50ns per context switch [61]), they still consume extra CPU time, compared to DINT that directly modifies incoming ethernet packets and forwards back.

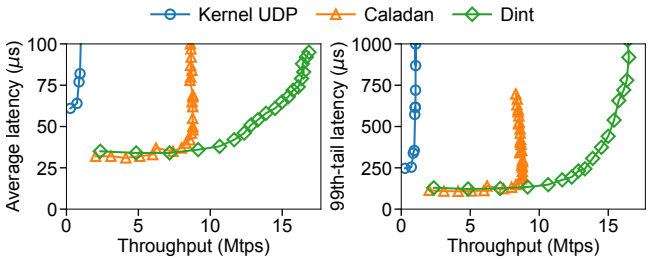
In terms of latency, kernel-bypass Caladan achieves lower minimum latency than kernel-stack DINT, e.g., 13μs vs. 14μs of the median and 20μs vs. 23μs of the 99th-tail for the 2PL lock server. The latency gap, especially for the 99th-tail, is mainly caused by the interrupt-driven nature of DINT, which includes the overheads of NIC interrupt delivery and running interrupt handler. We note that such overheads can be effectively amortized under high loads, thus not impacting throughput. We think the small increased latency is acceptable, as the current data center network usually has one or a few tens of microseconds RTT [20, 51].

Key-value store: Figure 5c shows the load-latency curves for the key-value store. Both Caladan and DINT's performance gets dropped compared to the lock managers, due to more compute in key-value operations. DINT achieves 2.17× higher throughput than Caladan, while having 0%-7%/27%-57% higher unloaded median/99th-tail latency. The minimum latency for Caladan and DINT is 14μs vs. 15μs for the median, and 21μs vs. 25μs for the 99th-tail, demonstrating DINT only incurs small interrupt handling overheads.

Log manager: Figure 5d shows the load-latency curves for the log manager. Similarly, DINT outperforms Caladan on throughput (by 3.6×) but sacrifices latency (by 0%-7% for unloaded median and 5%-40% for unloaded 99th-tail). Regarding the absolute performance number, DINT achieves up to 7.4 Mops per core. This translates into as low as 0.14μs per operation/packet, demonstrating the efficiency of offloading frequent-path operations into the kernel. Both DINT and Caladan achieve higher throughput on the log manager than the lock managers, as the serial log appending operations have better cache locality.

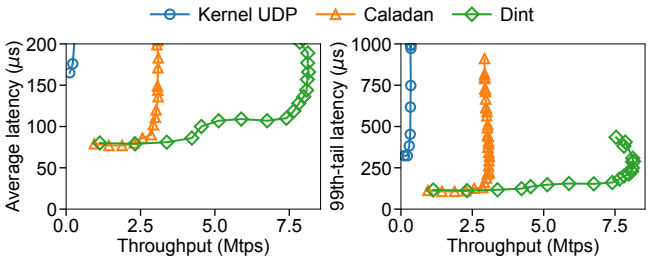
5.2 Transaction Benchmarks

We now evaluate DINT and other baselines on typical OLTP workloads, including TATP [47] and SmallBank [77]. TATP is a read-intensive OLTP benchmark modeling database behaviors of telecommunication providers. It features small key-values (8B keys and 40B values), 80% read-only transactions that read one or more keys, and 20% transactions that modify key-values. We provision 7 million TATP subscribers sharded across the three transaction servers. Similar to prior works [14, 29], we use the OCC-based transaction protocol



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 6: OCC on TATP workload. Mtps = Million transactions per second.



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 7: 2PL on SmallBank workload.

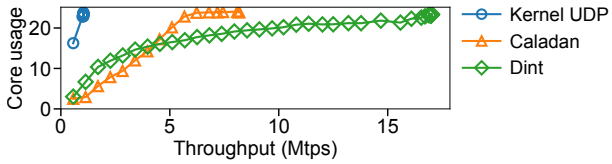


Figure 8: Core usage vs. throughput (on TATP).

(see §2.1) for the read-intensive TATP workload.

SmallBank is a write-intensive OLTP benchmark modeling bank account transactions, with 8B keys and values, and 85% write transactions. We provision 24 million bank accounts sharded across the three transaction servers. We use a 2PL-based transaction protocol suitable for write-intensive workloads. Compared to OCC, the 2PL-based protocol uses read-write locks in the read+lock phase without the validate phase; it has similar log and commit phases (§2.1).

DINT can easily support both transaction protocols by leveraging different lock managers and slightly changing client behaviors, demonstrating the genericity of its designs.

TATP: Figure 6a and 6b show how the average[4] and 99th-tail transaction latencies of different systems change when varying the throughput, respectively. DINT achieves 1.9× higher transaction throughput than Caladan with 6%-10%/12%-16% higher unloaded average/99th-tail latency. As described in Section 5.1, the higher throughput of DINT benefits from directly manipulating and forwarding raw ethernet/UDP packets immediately after the NIC driver receives the packets, in contrast to Caladan that works on a high-level connection-oriented abstraction. Meanwhile, batching effectively amortizes interrupt handling overheads in DINT, leading to a high sustained load on transaction servers. On the other hand, such batching inevitably causes higher latency for DINT when compared to the kernel-bypass polling-based Caladan, i.e., 3μs/14μs higher minimum average/99th-tail latency.

Although not an apple-to-apple comparison, we cite published performance numbers of RDMA-based transaction systems to demonstrate the throughput achieved by DINT is within the same order of magnitude as RDMA-based ones. For example, FaSST reports 8.7 Mtps/machine with 14 cores and 1 million TATP subscribers per machine [29, §6.2], while DINT achieves 5.62 Mtps/machine with 8 cores and 2.3 million subscribers per machine.

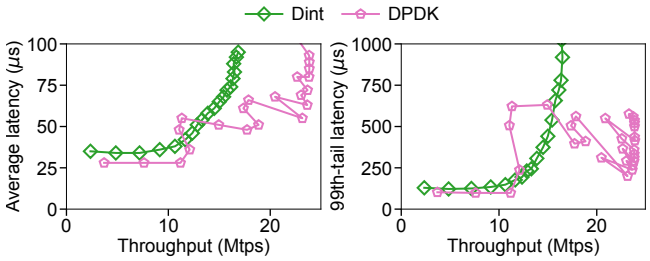
SmallBank: Figure 7a and 7b show the average and 99th-tail transaction latencies of different systems when varying transaction throughput under the SmallBank workload. DINT achieves 2.6× higher throughput than Caladan, while only adding 1%-5%/3%-9% unloaded average/99th-tail latency; the added minimum average/99th-tail latency is 2μs/5μs. Each SmallBank transaction consists of ∼10 transaction requests on average, including locking and key-value operations; therefore, DINT could sustain ∼82 million/sec request rate on 24 cores across three machines. Therefore, DINT's per-core request rate, i.e., ∼3.4 mops, is also within the same order of magnitude as RDMA two-sided operations, i.e., 3.6 mops reported by [81, Figure 3] on a ConnectX-6 NIC.

[4]We show the average rather than the median, as transaction workloads contain many small transactions that dominate the median latency.

5.3 CPU Utilization

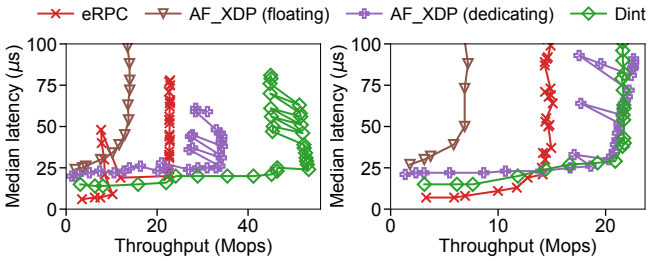
We now examine whether DINT can scale CPU usage as load changes, avoiding burning CPU cores. We use the same TATP workload as in Section 5.2, but provision enough number of clients, specify different transaction rates (by adjusting the sleeping time interval between two consecutive transaction requests in each client), and measure the CPU core usage of transaction servers. For kernel UDP and DINT, they rely on NIC interrupt to wake up any sleeping kernel-visible thread (i.e., pthread) when packets arrive. For Caladan, it supports a CPU-efficient non-spinning mode where the dedicated scheduler busy polls the NIC, and wakes up sleeping user-space threads when needed via IPIs (Inter-Process Interrupt); the Caladan scheduler also reallocates CPU cores every 5μs for the application process based on various load signals (e.g., packet and thread queueing delay [17]), to provision just-enough CPU cores for the current load.

Figure 8 shows how the CPU core usage varies with different throughput for different systems. Until 5 Mtps load, Caladan achieves the lowest core usage and can additively allocate more cores as the throughput increases, due to its fast core reallocation. After 5 Mtps load, DINT achieves lower CPU usage than Caladan and can additively scale its CPU usage to 17 Mtps, because of packet batching during interrupt handling. Kernel UDP has the worst CPU scaling curve, caused by the high overheads of frequent kernel networking stack traversing and user-kernel context switching. Neverthe-



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 9: Comparing raw DPDK with DINT (on TATP).



(a) OCC lock manager. **(b)** Key-value store.

Figure 10: Comparing eRPC and AF_XDP with DINT.

less, to enable more efficient CPU scaling for DINT under low loads, one way could be consolidating multiple NIC interrupts onto fewer cores to leverage batching to reduce per-packet processing overheads. We discuss more in Section 6.

5.4 Comparison to Raw DPDK

Figure 9 compares raw DPDK performance with DINT on the TATP workload. Here, "raw DPDK" means busy polling a batch of transaction packets (up to 64, similar to kernel NAPI) from the NIC, processing transaction requests, and then directly modifying and forwarding packets back in a batch as responses. Therefore, it is more efficient than the Caladan baseline, but requires busy polling all cores. Overall, DINT achieves 71% of the raw DPDK performance with 21%-25%/24%-28% higher unloaded average/99th-tail latency. The lower performance of DINT is mainly caused by two factors: 1) DINT devotes one core (out of eight) to the user-space process, and 2) DINT is interrupt-driven, trading some performance for better CPU efficiency by not busy polling any core (see §5.3). We note that the curves of the raw DPDK experience latency spikes in the middle due to insufficient packet batching.

5.5 Comparison to More Baselines

We now compare the performance of DINT with more baselines that leverage other networking stacks. In particular, we compare to eRPC [28] and AF_XDP socket [31]. eRPC is a kernel-bypass event-driven RPC library that builds on top of raw ethernet packets with its own efficient reliable transport protocol. It supports both DPDK and RDMA in busy-polling manners; our testing uses DPDK. AF_XDP is a new kernel socket family that leverages eBPF/XDP to directly DMA packet payload to a pre-registered user-space memory region, so that user-space applications can efficiently receive and send packets in a zero-copy manner. AF_XDP appears to applications as a set of socket APIs, so the application's packet processing logic can be written in a normal programming language (e.g., C/C++, Go) without the strict kernel verification as in eBPF. We run AF_XDP with two modes: floating where all provisioned cores handle NIC interrupts and run transaction servers, and dedicating where half of the cores handle NIC interrupts and another half run transaction servers.

Figure 10a and 10b shows the load-latency curves of eRPC, AF_XDP, and DINT for the OCC lock manager and key-value store respectively. For both applications, eRPC achieves the lowest minimum latency—8μs lower than DINT on both applications. For the lock manager, DINT achieves the highest throughput, and outperforms AF_XDP by 1.6× and eRPC by 2.3×. eRPC suffers from latency spikes at low loads because of insufficient RPC batching. For the key-value store that has more compute per operation, DINT has similar throughput as AF_XDP while achieving 29% lower minimum latency, because of directly handling requests in the kernel without going into the user space; DINT achieves 1.4× higher throughput than eRPC. The throughput results for eRPC must be taken with a grain of salt: eRPC builds a generic loss-tolerant RPC abstraction with session management, while DINT relies on transaction semantics to handle packet losses and works on raw ethernet/UDP packets.

KV workload [Throughput (Mops)]	Write-through (BMC [19])	Write-back	Write-back+BF (DINT)
All GETs, all exists	21.6	21.7	21.7
80% GETs, all exists	1.0	21.1	20.9
80% GETs, 31% exists	0.4	0.5	25.0

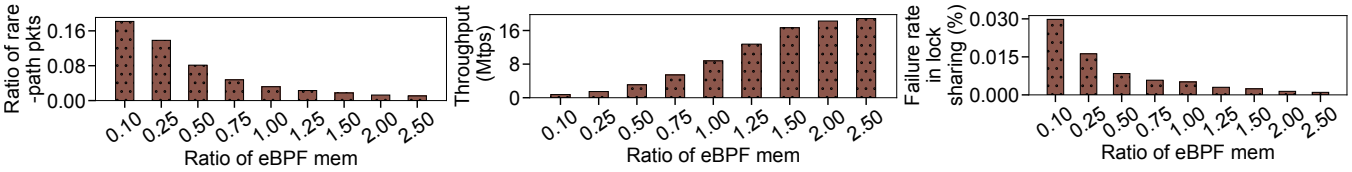
Table 1: Impact of write-back and Bloom filter. "80% GETs" and "31% exists" are based on the TATP workload and its largest table.

One interesting observation is that AF_XDP in the floating model performs much worse than the dedicating mode; similar results occur for DINT on the CPU placement of rare-path request handling process as described in Section 4. This is caused by the high user-kernel context switching overheads when co-locating interrupt handling and the application process on the same cores. We discuss further in Section 6.

5.6 Design Drill-Down

5.6.1 Impact of Write-Back and Bloom Filter

Table 1 shows how the write-back and Bloom filter designs impact DINT performance on different key-value store workloads. With all GETs and all keys existing, the write-through, write-back, and write-back + Bloom filter achieve similar throughput. Once with 20% PUTs, the write-through throughput drops to 1.0 Mops because of handling PUTs in the user space, while the other two keep similarly high throughput. Furthermore, adding 68.75% key-value operations for non-



(a) Rare-path ratio. **(b)** Throughput. **(c)** Failure rate in lock sharing.

Figure 11: Impact of varying the eBPF memory size under the TATP workload. "Ratio of eBPF memory" is against the workload dataset size including both locks and key-values.

Interrupt collocation	Lowest unloaded average/99p latency	Maximum throughput
Collocating with app	35/139 μs	7.0 Mtps
Not collocating (DINT)	34/122 μs	16.9 Mtps

Table 2: Impact of collocating interrupt processing and the application on the same cores (on TATP).

existing keys, only the write-back + Bloom filter can achieve high throughput, as it handles most key-value operations in the kernel, for both existing and non-existing keys.

5.6.2 Impact of Rare-Path Ratio

Figure 11a and 11b shows how different rare-path ratios (by changing the eBPF memory size) impact DINT's transaction throughput. The rare-path ratio significantly impacts DINT performance. For example, with 10% of the workload dataset size in the eBPF memory, which gives 18% of rare-path packet ratio, DINT only achieves 740 Kops. Once we provision the eBPF memory to be 1.5× of the workload dataset size, similar to how FaSST [29] provisions its hash table, there will be only 1.7% of rare-path packet ratio, and DINT reaches 16.7 Mops. This supports the DINT's design principle of offloading frequent-path operations as much as possible into the kernel.

5.6.3 Impact of Lock Sharing

Figure 11c shows how the failure rate caused by lock sharing varies with different sizes of the eBPF memory (i.e., different sizes of the lock table). Overall, the failure rate is under 0.03%; when we provision the eBPF memory to be 1.5× of the workload dataset size, the failure rate is only around 0.002%. This confirms that lock sharing works well on typical OLTP workloads.

5.6.4 Impact of Interrupt Collocation with Applications

Table 2 shows how collocating interrupt processing with the application impacts transaction latency and throughput. Interrupt collocation slightly increases the lowest unloaded average/99th-tail latency by 3%/14% compared to no collocation, because interrupt processing contends CPU cores with application threads; it significantly reduces the maximum throughput by 59%, as high interrupting rate easily starves the application threads, bottlenecking the system performance.

6 Discussion and Future Work

Symmetric vs. asymmetric models: DINT adopts an asymmetric client-side transaction model [42, 56, 60, 87], where each transaction server "passively" handles incoming transaction requests. DINT then leverages eBPF/XDP to offload transaction server operations into the kernel. In contrast, a symmetric model [14, 29, 82] requires the transaction server to also act as a client to issue transaction requests. This creates challenges to DINT, as eBPF/XDP itself cannot generate new packets. Fortunately, by leveraging the AF_XDP technique (see §5.5) that provides fast packet sending functionality, DINT could support symmetric models efficiently. We leave the integration of DINT with AF_XDP as future work.

Implications to networking stack research: DINT shows that the kernel networking stack can achieve kernel-bypass-like throughput and latency, but has worse CPU efficiency under low loads than well-engineered kernel-bypass stacks (§5.3). Therefore, we call for more research on optimizing the CPU efficiency of the kernel networking stack that offloads application operations. One idea may be smartly consolidating NIC interrupts to just-enough CPU cores by manipulating the NIC IRQ affinity, which leverages batching during interrupt handling to reduce per-packet processing overheads. This shares the same goal as Shenango [61] and Caladan [17], but targets the interrupt-driven kernel networking stack.

Another research problem is how to isolate the kernel stack-offloaded operations and user-space operations, as naively co-locating both on the same cores would cause severe performance drop due to frequent user-kernel context switching (see §4 and §5.5). DINT currently uses a simple static partitioning policy, but a more advanced dynamic partitioning policy could possibly provide better performance.

Implications to transaction protocol research: Co-designing transaction protocols with eBPF allows for both high performance and good CPU efficiency. In this work, we co-design an OCC/2PL-based transaction protocol in DINT. DINT should also be able to support more advanced transaction protocols like MDCC [40] and Tapir [87] that essentially rely on read-write and version-based locking. To support advanced protocols like ROCOCO [57] and Janus [58] that maintain transaction dependency DAGs in the lock manager, DINT would need to maintain complex graph data structures in eBPF, which calls for more co-designs to address the challenge of eBPF programming constraints. In an attempt

to reduce CPU utilization, many transaction systems have pushed to incorporate network offload devices like RDMA and smartNICs [14, 73]. However, these devices are much more expensive than commodity NICs, and come with customized network stacks that have high maintenance overheads in terms of engineering. DINT provides the opportunity for accomplishing similar goals without the need for expensive customized hardware, and provides a new point in the design space for transaction protocol developers to explore.

Wish list for eBPF: DINT suffers from the fixed-size eBPF maps, and no dynamic memory allocations for handling large key-values. Therefore, the most helpful eBPF feature would be supporting dynamic memory allocations so that offloaded states could be more memory-efficient. Another helpful feature would be the egress XDP hook. When developing DINT, we were thinking of using the AF_XDP socket to process rare-path request packets (instead of the slower UDP socket); however, AF_XDP faces troubles with the egress bookkeeping of in-kernel states (§3.2), as it relies on the ingress-only XDP while bypassing the egress TC hook. Currently, the only way for AF_XDP to work is by calling eBPF functions in the user space, but this suffers from high syscall overheads. If the kernel supports the egress XDP hook, DINT could instead leverage the faster AF_XDP socket to handle rare paths.

7 Related Work

Distributed in-memory transactions: By leveraging battery-backed DRAM or NVRAM, distributed transactions are no longer bottlenecked by disk IOs, but the networking IOs. This has spurred a series of research that leverages RDMA to implement distributed in-memory transactions, e.g., FaRM [14], FaSST [29], `DrTM` [83], `DrTM+R` [8], `DrTM+H` [82], and Prism [6]. Rather than using RDMA that bypasses kernels, DINT sticks to the most common commodity NICs with the kernel networking stack for better security, isolation, protection, maintainability, and debuggability, without losing performance.

High-performance networking stacks: The inefficiency of traditional kernel networking stack has motivated the designs of many kernel-bypass networking stacks, e.g., mTCP [24], eRPC [28], Snap [51], Demikernel [86] and more [17, 30, 37, 61, 63, 74]. These stacks generally require DPDK-style busy polling, and trades security, protection, maintainability, and more for high performance. Instead, DINT provides comparable high performance without busy polling for distributed transaction applications, while guaranteeing kernel-based security, protection, maintainability, etc.

Perhaps the most relevant work to DINT in this space is IX [3] which implements a protected kernel networking stack and achieves kernel-bypass performance. To achieve high networking performance, IX leverages adaptive batching to amortize user-kernel transition overheads, while DINT relies on the built-in batching of the existing kernel networking stack to amortize interrupt handling overheads. One advantage of DINT over IX is that DINT directly works for existing widely-deployed Linux kernels without any kernel modifications or customized kernel modules.

Hardware offloading for applications: Offloading network-intensive operations to specialized hardware such as FPGA [1, 22, 41, 45], SmartNICs [36, 43, 46, 64, 72, 81], and programmable switches [13, 25, 26, 85] significantly improves application performance. However, they are generally hard to deploy in today's cloud environments [28, 89], as these advanced hardware are not widely available in the public cloud. In contrast, DINT aims to be generic and readily-deployable without relying on any specialized hardware by leveraging the kernel-native eBPF techniques on widely-deployed modern Linux kernels and CPU platforms.

eBPF applications: eBPF is mostly used for packet filtering [52], infrastructure monitoring [2, 66], and L4 load balancing [16] in industry. Recent research has proposed more applications including: accelerating key-value stores [19], sidecar proxies [68], Paxos [89], DBMS proxies [7], gathering congestion control signals [59], guiding request scheduling [27], offloading storage functions [88], and optimizing locks [62]. DINT is a new eBPF application targeting distributed transactions.

8 Conclusion

DINT is a distributed in-memory transaction system under the kernel networking stack, yet achieving kernel-bypass-like throughput and latency. DINT achieves this by offloading transaction data structures and operations into the kernel via eBPF techniques, significantly reducing kernel stack overheads. Compared to a transaction system implemented using Caladan, a well-engineered kernel-bypass networking stack, DINT even achieves 2.6× higher throughput and only adds 10%/16% unloaded average/99th-tail latency.

More importantly, DINT challenges the conventional belief that the kernel networking stack is not suitable for distributed in-memory transactions, or generally, μs-scale networked applications; DINT shows that, with proper application-kernel co-design enabled by eBPF, one important class of such applications under the kernel networking stack can achieve kernel-bypass-like performance. DINT code is available at `https://github.com/DINT-NSDI24/DINT`.

Acknowledgments

We thank our shepherd Tom Barbette and the anonymous reviewers for their insightful comments. We thank Cloudlab [15] for providing us with the development and evaluation infrastructure. We also thank Zhiying Xu and Junzhi Gong for their helpful feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by the Google PhD Fellowship.

References

[1] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication. In *Proceedings of USENIX NSDI*, pages 357–374, 2023.

[2] The Cilium Authors. Cilium: eBPF-Based Networking, Observability, Security. https://cilium.io/.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[4] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of ACM SOSP*, pages 267–283, 1995.

[5] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of ACM SOSP*, pages 228–242, 2021.

[7] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces with User-Bypass. *Proceedings of the VLDB Endowment*, 16(11):3335–3348, 2023.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of ACM EuroSys*, pages 1–17, 2016.

[9] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[10] Intel Corporation. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[11] NVIDIA Corporation. Understanding interrupt moderation. https://enterprise-support.nvidia.com/s/article/understanding-interrupt-moderation.

[12] The Transaction Processing Council. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[13] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at Network Speed. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–7, 2015.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[16] Facebook. Katran: A High-Performance Layer 4 Load Balancer. https://github.com/facebookincubator/katran.

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[18] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. Citron: Distributed Range Lock Management with One-sided RDMA. In *Proceedings of USENIX FAST*, pages 297–314, 2023.

[19] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[20] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of ACM CoNEXT*, pages 54–66, 2018.

[22] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[23] Brendan Jackman. Atomics for eBPF. https://lwn.net/Articles/840224/.

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of USENIX NSDI*, pages 35–49, 2018.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of ACM SOSP*, pages 121–136, 2017.

[27] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[28] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[30] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[31] The Linux kernel development community. AF_XDP. https://docs.kernel.org/networking/af_xdp.html.

[32] The Linux kernel development community. BPF_MAP_TYPE_ARRAY and BPF_MAP_TYPE_PERCPU_ARRAY. https://docs.kernel.org/bpf/map_array.html.

[33] The Linux kernel development community. BPF_MAP_TYPE_BLOOM_FILTER . https://docs.kernel.org/bpf/map_bloom_filter.html.

[34] The Linux kernel development community. NAPI. https://docs.kernel.org/networking/napi.html.

[35] The Linux kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html.

[36] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of ACM SOSP*, pages 756–771, 2021.

[37] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of USENIX ATC*, pages 863–880, 2019.

[38] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proceedings of USENIX NSDI*, pages 31–48, 2023.

[39] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *Proceedings of USENIX NSDI*, pages 287–305, 2022.

[40] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of EuroSys*, pages 113–126, 2013.

[41] 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 ACM SOSP*, pages 137–152, 2017.

[42] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[43] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing. In *Proceedings of USENIX ATC*, pages 951–966, 2022.

[44] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of USENIX NSDI*, pages 429–444, 2014.

[45] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella.

RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *Proceedings of USENIX NSDI*, pages 1293–1308, 2023.

[46] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proceedings of ACM SIGCOMM*, pages 318–333. 2019.

[47] IBM Software Group Information Management. Telecom Application Transaction Processing Benchmark. https://tatpbenchmark.sourceforge.net/.

[48] Linux Programmer's Manual. bpf-helpers(7). https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[49] Linux Programmer's Manual. bpf(2). https://man7.org/linux/man-pages/man2/bpf.2.html.

[50] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[51] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[52] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[53] John McNamara. API/ABI Stability and LTS: Current state and Future. https://www.dpdk.org/wp-content/uploads/sites/35/2017/09/DPDK-Userspace2017-Day2-2-ABI-Stability-and-LTS-Current-state-and-Future.pdf.

[54] The memcached contributors. Memcached - a Distributed Memory Object Caching System. https://memcached.org/.

[55] Microsoft. eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows.

[56] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling RDMA RPCs with Flock. In *Proceedings of ACM SOSP*, pages 212–227, 2021.

[57] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of USENIX OSDI*, pages 479–494, 2014.

[58] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of USENIX OSDI*, pages 517–532, 2016.

[59] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[60] The University of Texas at Austin. Natacha Crooks. A client-centric approach to transactional datastores. https://repositories.lib.utexas.edu/handle/2152/81352.

[61] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[62] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

[63] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of USENIX OSDI*, pages 663–679, 2018.

[65] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI*, pages 43–57, 2015.

[66] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[67] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[68] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[69] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *Proceedings of USENIX Security*, pages 4277–4292, 2021.

[70] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of ACM HotOS*, pages 152–158, 2021.

[71] Salvatore Sanfilippo. Redis: An In-Memory Database That Persists on Disk. https://github.com/redis/redis.

[72] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[73] 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.

[74] ScyllaDB. SeaStar High Performance Server-Side Application Framework. https://github.com/scylladb/seastar.

[75] Alexei Starovoitov. BPF at Facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[76] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of ACM HotNets*, pages 61–68, 2021.

[77] The H-Store Team. SmallBank Benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[78] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of ACM SOSP*, pages 18–32, 2013.

[79] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, pages 245–257, 2021.

[80] VMware. Update to VMware's per-CPU Pricing Model | VMware. http://web.archive.org/web/20211023072913/https://news.vmware.com/company/cpu-pricing-model-update-feb-2020.

[81] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing Off-Path SmartNIC for Accelerating Distributed Systems. In *Proceedings of USENIX OSDI*, pages 987–1004, 2023.

[82] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[83] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of ACM SOSP*, pages 87–104, 2015.

[84] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of ACM HotOS*, pages 70–79, 2023.

[85] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020.

[86] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[87] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[88] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

[89] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, pages 1391–1407, 2023.