iPipe: A Framework for Building Datacenter Applications Using In-networking Processors

Ming Liu
University of Washington

Arvind Krishnamurthy
University of Washington

Simon Peter
UT Austin

Karan Gupta
Nutanix

Abstract

The increasing disparity of data center link bandwidth and CPU computing power motivates the use of in-networking processors to co-execute parts of datacenter applications. By offloading computations onto a NIC-side processor, we can not only save endhost server CPU cores, but also achieve lower request latency. However, building applications with an in-networking processor brings three challenges: programmability, offloading constraints, and multi-tenancy support.

This work proposes iPipe, a framework for developing datacenter services that can take advantage of an in-networking processor on a programmable NIC. iPipe provides an actor programming model and exposes various APIs through the iPipe runtime. It enables efficient NIC hardware utilization and fair computational resource sharing via a lightweight actor scheduler, distributed shared objects, a cross PCIe messaging tier, a shim networking stack, and a dynamic resource manager. We build three datacenter applications (i.e., a real-time data analytics engine, a distributed transaction system, and a replicated key-value store) based on iPipe and prototype them using commodity programmable NICs (i.e., Cavium LiquidIO). Real system based evaluations show that when processing 10Gbps of application bandwidth, NIC-side offloading reduces the average number of beefy Intel cores (of three applications) from 2.2 to 0.4, along with up to 15.8μs latency savings. We also demonstrate that iPipe is able to provide performance isolation with fair bandwidth allocation, and that it scales to multiple programmable NICs.

1 Introduction

Recent years have seen a rapid increase in network interface bandwidth [13, 14] for datacenter servers, outpacing the CPU computing power. For example, since 2009, Azure [22] has seen a 50x improvement in the network bandwidth, but not a 50x increase on CPU performance. Therefore, datacenter operators have to burn more CPU cores to fully utilize the network bandwidth, leaving fewer computing resources for tenant workload execution.

Emerging in-networking processor based programmable NICs (e.g., Cavium LiquidIO [10], Netronome Agilio [45], Mellanox BlueField [41]) offer a solution. An in-networking processor has an array of wimpy cores, with capabilities to (1) access limited on-board SRAM/DRAM via high-bandwidth coherent memory buses; (2) interact with packet processing/domain specific accelerators and programmable DMA engines, using high-performance interconnects. These in-network computing capabilities allow hosts to offload limited amounts of simple but general computations onto the programmable NIC, while continuing to support complex application logic on the hosts. By offloading lightweight but frequently invoked operations, we can accelerate datacenter applications while reducing the host CPU load without sacrificing program generality.

The key idea of this paper is using an in-networking processor on a programmable NIC to co-execute distributed datacenter workloads so that one can reduce both request execution latency on the fast path, as well as host CPU computation load. Specifically, we explore the feasibility of refactoring common datacenter applications, executing lightweight latency-sensitive application logic on a NIC-side in-networking processor and more general logic on the host CPU.

However, this idea is non-trivial to realize given the following challenges. First, there is no programming model support for such non-cache-coherent and heterogeneous programmable NIC accelerated servers, where an in-networking processor on the data path communicates with the host CPU using a high-bandwidth PCIe bus. Second, there are offloading constraints given the compute, memory and communication capabilities of the NIC. Specifically, an in-networking processor should not be oversubscribed for workload execution and must always be able to handle the full network bandwidth. When designing the data structures and laying out the application working set, one needs to consider the complex memory hierarchy along with unusual performance characteristics (e.g., remote host memory has asymmetric latency for reads/writes) so that the NIC processor is able to interact with the host processor effectively. Third, one should support multi-tenancy, where different applications are able to safely share the NIC's computational resource.

We design and implement the iPipe framework for programmable NICs, which addresses the above challenges and allows programmers to develop datacenter applications using an in-networking processor effectively. iPipe provides an actor programming model and a runtime system that includes the following components: (1) a lightweight work conserving scheduler that maximizes the NIC processor utilization without hurting the target link bandwidth; (2) distributed object abstractions that enable efficient use of remote host memory and flexible actor migration; (3) a cross PCIe messaging tier and a shim networking stack for communicat-

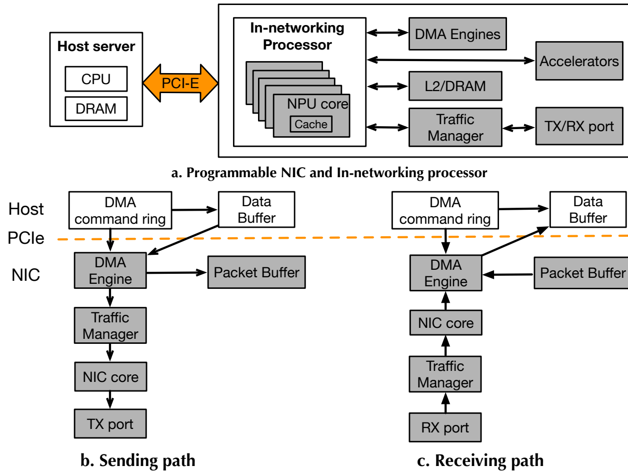


Figure 1: Architecture block diagram and packet communication path for a programmable NIC.

ing with host processors and RX/TX ports; (4) a dynamic resource manager that supports weighted max-min fairness on the link bandwidth and provides execution latency guarantees for multi-application consolidations. Crucially, programmers can express applications using the actor model and rely on the runtime to automatically schedule (and/or migrate) the actor execution on either the in-networking processor or the host CPU.

We prototype iPipe and build three applications (i.e., a real-time data analytics engine, a distributed transaction processing system, and a replicated key-value store) using commodity in-networking processor based programmable NICs (i.e., Cavium LiquidIO). We evaluate the system using an 8-node testbed and compare the performance against DPDK-based implementations. Our experimental results show we can significantly reduce the host load for three real-world datacenter applications; iPipe reduces the average number of beefy Intel cores used to process 10Gbps of application bandwidth from 2.2 to 0.4, along with up to 15.8μs savings in request processing latency. When consolidating three applications, iPipe is able to minimize tail latency by 11.5μs and provide fair bandwidth allocation. We characterized different iPipe primitives to understand where the latency savings come from and demonstrate the scalability of iPipe under multiple programmable NICs.

2 Background & Motivation

In-networking processors hold the potential to offload end-host computations. This section provides the necessary background and discusses the challenges of integrating an in-networking processor into the operation of a datacenter application.

2.1 Programmable NICs

Figure 1-a shows the architecture of a typical programmable NIC [10, 45, 41] with an in-networking processor. It comprises of a multi-core NPU (networking processor unit), device local memory (L2 cache/DRAM), a traffic manager, DMA engines, and TX/RX ports. These architectural components connect with each other using high-performance interconnects (e.g., coherent memory bus, I/O interconnect). The traffic manager has three functionalities: (1) fetch incoming/outgoing packets; (2) feed packets into the NIC core; and (3) track the packet order. An in-networking processor, packaged with some local L1 cache, can (1) perform general-purpose computations; (2) read/write the device local memory; (3) issue host DMA commands to the engine; and (4) send packets to the TX port. Programmable NICs also provide on-chip/off-chip accelerators to speed up certain operations (e.g., encryption, hash calculation, look up engines, packet buffer management, etc.).

This work uses a recent commercial programmable NIC (i.e., Cavium LiquidIO [10]) to build the iPipe framework. Its major computing unit is a Cavium OCTEON processor [8] with 12 cnMIPS wimpy cores, which run at 1.20GHz and enclose 32KB L1 cache. The NIC has 4MB shared L2 cache, 4GB on-board device memory, a number of acceleration engines, and two 10Gbps ports.

The added programmability comes at a minimal cost. A two-port 10 Gbps LiquidIO [10] server adapter from Cavium costs about $350, comparable to an RDMA NIC (e.g., $438 for Mellanox ConnectX-4 EN), cheaper than an iWARP NIC (e.g., $733 for Chelsio T62100), and a bit more expensive than a traditional two-port 10Gbps NIC (e.g., Intel X710 costs around $290). Similar commercial programmable NICs include Netronome Agilio [45], and Mellanox BlueField [41].

Programmable NICs have packet communication paths similar to that of normal NICs, except that the computing unit can process the raw packet content. On the sending side (Figure 1-b), the host creates a DMA control command (including the instruction header and packet buffer address) and writes it into a command ring. The NIC DMA engine fetches the command and data from host memory and writes into the packet buffer (which is located in NIC memory). The traffic manager then generates a work item (including the address of the packet) and delivers it to the NIC core. After some processing, the NIC sends the packet out through the TX port. The receiving side (Figure 1-c) is similar but in the reverse order: Packets come from the RX port and are delivered to the NIC core via the traffic manager. The NIC core performs some computations and issues a DMA instruction word to the engine. The DMA engine then fetches descriptor ring contents (which includes a pointer to the host data buffer) and writes data from NIC to the host.

2.2 Key idea and challenges

The key thrust of this paper is offloading parts of application logic onto computing units (especially an in-networking processor) of a programmable NIC. Such offloaded computations should stay within the NIC-side computing and memory capability, without impacting other traffic. With such co-execution, one can achieve two benefits: (1) reducing end-host server CPU computing burden so that host CPU cores are able to consolidate more applications/VMs; (2) minimizing request execution latency by processing data in the packet communication path. Realizing this idea brings in three challenges:

Programmability. A programmable NIC accelerated server is a non-cache-coherent heterogeneous computing platform where (1) host/NIC processors communicate with each other via a reliable high-bandwidth PCIe bus; (2) both sides have certain hardware parallelism with performance asymmetric computing power; (3) NIC processors are in the data path between host and network. This is significantly different from existing heterogeneous systems, such as ARM's big.LITTLE, where the processors share the same cache coherence domain, and GPGPUs, where the host CPU is the computing coordinator and the GPU is off the communication path. Hence, we need a programming model that can achieve efficient concurrency while tolerating heterogeneity, partitioned memory, as well as chipset (i.e., PCIe) communication.

Offloading constraints. One should offload application computations within the programmable NIC hardware limits. First, an in-networking processor should not be oversubscribed. The NIC's main task is to transfer network packets between hosts and NIC TX/RX ports. When overloaded with other computations, the NIC will not be able to handle the link bandwidth. For example, on our platform, a NIC core takes around 0.37μs to read from RX and transmit to TX, in order to fully utilize the bandwidth with 12 cores, each core only has 0.85μs and 10.05μs left for additional processing of requests for 64B and 1KB packets, respectively. Second, from the in-networking processor's point of view, the system presents a deep non-uniform memory subsystem including four components: per-core L1 cache, global shared L2 cache, global device memory, and remote host memory. In addition, remote host memory has asymmetric latency for reads and writes. This makes it harder to design efficient data structures and to lay out the application working set. Third, PCIe performance is notoriously unstable and impacted by many factors [46]. A DMA engine usually provides blocked (which waits until the DMA completion word coming back)/non-blocking (which returns immediately after inserting the command to the DMA engine queue) primitives with different performance characteristics. One should choose the right one for communication.

Multi-tenancy support. Data center servers often execute workloads of multiple tenants simultaneously. When offloading application-level tasks to a programmable NIC, we have to ensure that multiple tenants can safely share the NIC. Hence, we need to (1) guarantee that different applications cannot touch each other's application working set; (2) minimize the application interference and ensure that tail latency increases, if any, are modest; (3) provide resource allocation across applications with flexible policies.

3 iPipe framework

iPipe is a framework for building applications using an in-networking processor efficiently while addressing the above-mentioned challenges. This section describes its programming model, APIs, and runtime system.

3.1 An application written using iPipe

Replicated key-value store (RKV) is a critical datacenter service [17, 15]. It mainly comprises of two key system components: a *consensus protocol*, and a *key-value data store*. Given its use of sophisticated data structures and algorithm, we use it as a running example to introduce various aspects of the iPipe framework.

Our application uses the traditional Multi-Paxos algorithm [34] to achieve consensus among multiple replicas. Each replica maintains an ordered log for every Paxos instance. There is a distinguished leader that receives client requests and performs consensus coordination using Paxos prepare/accept/learning messages. In the common case, consensus for a log instance can be achieved with a single round of accept messages, and the consensus value can be disseminated using an additional round (learning phase). Each node of a replicated state machine can then execute the sequence of commands in the ordered log to implement the desired replicated service. When the leader fails, replicas will run a two-phase Paxos leader election (which determines the next leader), choose the next available log instance, and learn accepted value from other replicas if its own log has gaps. Typically, the Multi-Paxos protocol can be expressed as a sequence of messages that are generated and processed based on the state of the RSM log. One can apply batch processing and piggy-backed commits to optimize system performance.

The application uses a log-structured merge tree (LSM) to implement the key-value store; LSM trees are widely used in many KV systems (such as Google's Bigtable [11], LevelDB [35], Cassandra [4], HBase [5], and MangoDB [39]). An LSM tree accumulates recent updates in memory and serves reads of recently updates values from in-memory data structures, flushes the updates to the disk sequentially in batches, and merges long-lived on-disk persistent data to reduce disk seek costs.

The LSM tree mainly comprises of two components: *memtable*, a sorted data structure (e.g., a SkipList or a B/B+ tree) and *SSTables*, collections of data items sorted by their

keys and organized into a series of levels. Each level has a size limit on its SSTables, and this limit grows at an exponential rate with the level number. A first level SSTable is obtained by dumping the Memtable to stable storage using a *minor compact* operation; this could result in the same key being stored in multiple SSTables in the first level. The SSTables at other levels are generated using *major compact* operations, which merges one SSTable from level L with all other overlapping ones from the next level L+1. This operation produces a series of new level L+1 files, updates the indexing information, and discards the input SSTable. Inserting a key-value pair requires updates to the Memtable. Deletions are a special case of insertions wherein a deletion marker is stored. Data retrieval might require multiple lookups on the Memtable and the SSTables (starting with level 0 and moving to higher levels) until a matching key is found. A bloom filter could be applied to accelerate this process. Minor/major compactions usually run in a background thread and are triggered when the Memtable is full or the number of SSTables at a given level reach a threshold.

3.2 Actor programming model

iPipe provides an actor programming model [26, 2, 49] for application development. iPipe uses the actor-based model, instead of say dataflow or thread-based models, because (1) the actor model is able to support computing heterogeneity and hardware parallelism automatically; (2) it doesn't rely on shared memory abstractions; (3) today's datacenter workloads use RPCs (remote procedure calls) as the communication paradigm and some of them are control-heavy; (4) our objective is to offload computations associated with latency-sensitive, general-purpose message handlers.

```
typedef struct _actor{
    int app_id;                 // App unique ID;
    int actor_id;               // Actor unique ID;
    int weight;                 // Allocation weight;
    uint16_t port;              // Communication port
    void *private_state;        // Internal states;
    Init *init_handler;         // State init func;
    Exec *exec_handler;         // Message exec func;
    spinlock_t *exec_lock;      // Concurrency lock;
    concur_queue *mailbox;      // Message queue;
    actor_entry *actor_tbl;     // Actor table;
    rate_limiter *rt;           // Rate limiter;
} __attribute__((packed)) actor;
```

An actor is a computation agent that performs three kinds of operations based on incoming type of messages: (1) trigger its execution handlers and manipulate its private state; (2) create actors and register them into the runtime; (3) interact with other actors by sending messages asynchronously. Actors don't share memory. The code snippet above shows the major fields of an actor structure in iPipe. When creating an actor, one needs to provide *init_handler* and *exec_handler* for state initialization and message execution. *private_state* can use different data types (as described in Sections 3.4 and 3.5) and *mailbox* is a multi-producer multi-consumer concurrent

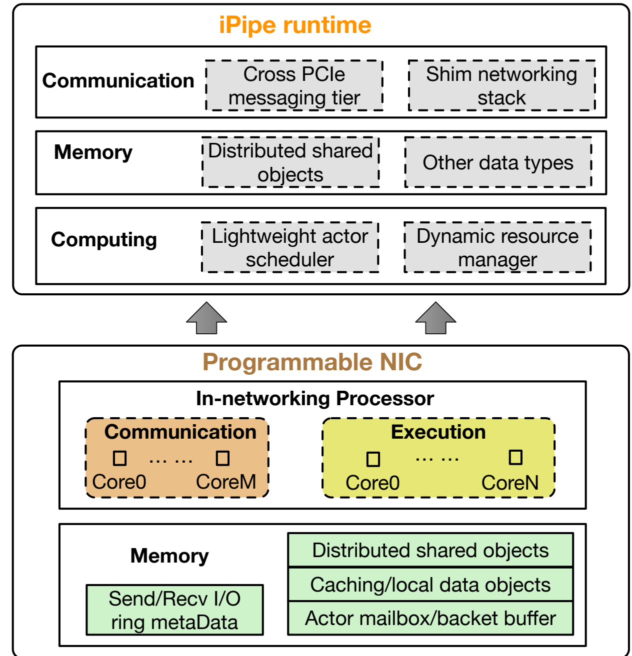


Figure 2: An overview of the iPipe runtime on the programmable NIC. The host side runtime has similar components.

FIFO queue, which is used to save incoming asynchronous messages. *exec_lock* is used to decide whether an actor can be executed on multiple cores (discussed in Section 3.6). *rt* is the rate limiter to control the incoming message rate. Our runtime will generate a unique *port* number, set the *app_id*, and assign *actor_tbl* that contains the communication address for all actors. An actor can also provide a weight that will be used for bandwidth allocation.

RKV use case: Each data shard is handled by three major actors: (1) The first one is NIC-resident and handles the Multi-Paxos logic; (2) The second one handles the LSM logic for implementing the memtable. It is instantiated on the NIC, but it is migrated to the host when the memtable exceeds the limit and its state is serialized into an SSTable before it is removed/deleted from the runtime; (3) The third one is host-resident and is used to process SSTable reads/writes.

3.3 iPipe runtime and APIs

The iPipe runtime (see Figure 2) manages and schedules the computational resources of the programmable NIC and exposes APIs for managing the computing, memory, and communication resources as shown in (Table 1). Specifically, iPipe provides a distributed shared object abstraction that enables flexible actor migration, as well as support for a software managed cache and NIC-local objects (see Sections 3.4 and 3.5). In the communication layer, iPipe provides a cross PCIe message passing tier (Section 3.7) for commu-

	API	Explanation
Actor	actor_create	create an actor
	actor_register	register an actor into the runtime
	actor_init	initialize the actor private state
	actor_delete	delete the actor from the runtime
	actor_migrate	migrate an actor to host
DSO	dso_malloc	allocate a dso obj.
	dso_free	free a dso obj.
	dso_mmset	set space in a dso with value
	dso_mmcpy	copy data from a dso to a dso
	dso_mmmove	move data from a dso to a dso
CACHE	cache_get	read a <key, value> based on key
	cache_put	write/update the <key, value>
	cache_del	delete a <key, value> pair
MSG	msg_init	init. a remoge message I/O ring
	msg_read (*)	read new messages form the ring
	msg_write	write messages into the ring
Nstack	nstack_new_wqe	create a new WQE
	nstack_hdr_cap	build the packet header
	nstack_send	send a packet to the TX
	nstack_get_wqe	get the WQE based on the packet
	nstack_recv(*)	receive a packet from the RX

Table 1: iPipe major APIs. There are five categories: actor management (Actor), distributed shared object (DSO), local memory (LMEM), software managed cache (CACHE), message passing (MSG), and networking stack (Nstack). We don't list the LMEM ones as they are similar to the DSO operations. We list only the most important Nstack operations; the Nstack API has additional methods for packet manipulation. APIs with * are mainly used by the runtime as opposed to actor code.

nications between the host and the NIC, as well as a simplified networking stack that delivers packets from/to TX/RX ports (see Section 3.8). iPipe has an actor scheduler (see Section 3.6), which schedules (and/or migrates) actor execution instances on the host/NIC, and uses a resource manager that supports multi-tenancy based on user policies (see Section 3.9).

The iPipe runtime provides actors with interfaces to interact with other actors, manage memory, and initiate or receive network messages. It also provides automatic mechanisms for efficient resource utilization in the form of transparent scheduling and migration of actors and distributed shared objects, so that the NIC's hardware constraints are not violated. Generally, an actor *exec_handler* is implemented as a couple of message handlers. For example, the Paxos actor on the leader has four handlers on client request, leader election promise, leader election accept acknowledge, and command accept acknowledge messages. Within the handler procedure, one can use various iPipe primitives.

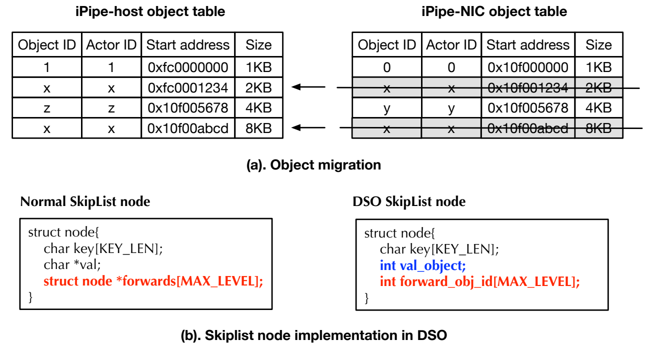


Figure 3: iPipe distributed share objects.

3.4 Distributed shared objects

iPipe provides the distributed shared object abstraction (DSO) to enable flexible actor migration. iPipe maintains an object table (Figure 3-a) on both sides and utilizes the local memory manager to allocate/deallocate copies. At any given time, a DSO has only one copy, either on the host or on the NIC. We also do not allow an actor to perform reads/writes on objects across the PCIe because remote memory accesses are 10x slower than local ones (shown in Section 5.2). Instead, iPipe would automatically move DSOs along with the actor and all DSO read/write/copy/move operations are performed locally. If an actor on the programmable NIC cannot hold a large working set due to memory limits, it will migrate itself from the NIC to the host using the *actor_migrate* primitive.

Our DSO provides five APIs as shown in Table 1. When creating an object on the NIC, iPipe first allocates a local memory region using the *dlmalloc2* allocator and then inserts an entry (i.e., object ID, actor ID, start address, size) into the NIC object table. Upon *dso_free*, iPipe frees the space allocated for the object and deletes the entry from the object table. *dso_memset, dso_memcpy, dso_memmove* resemble memset/memcpy/memmove APIs in glibc, except that it uses the object ID instead of a pointer. When migrating an actor to the host, as shown in Figure 3, our runtime (1) collects all objects that belong to the actor; (2) sends the object data to the host side using messages and DMA primitives; (3) creates new objects on the host side and then inserts entries into the host-side object table; (4) deletes related entries from the NIC-side object table upon deleting the actor. The host-side DSO works similarly, except that it uses the glibc memory allocator. When using DSOs to design a data structure, one has to use the object ID for indexing instead of pointers.

RKV use case: The skiplist based memtable is implemented using DSO. As shown in Figure 3-b, a traditional skiplist node includes a key string, a value string, and a set of forwarding pointers. With DSO, the key field is the same. Value and fowarding pointers are replaced by object IDs.

When traversing, one will use the object ID to get the start address of the object, cast the type, and then read/write its contents.

3.5 Local memory objects and software managed cache

iPipe also provides two other kinds of data types to optimize object access performance. One is local memory object that is allocated on either the NIC or the host but cannot be migrated. We make extensive use of local memory objects for fast local data access and for indexing distributed shared objects.

The other one is a software managed cache to store frequently accessed data. This software cache borrows techniques from previous work [21]. It applies a 4-way concurrent cuckoo hash function and uses an optimistic locking design [33, 21]. Our cache operates in a write-through mode. Programmers are responsible for (1) setting up indexing keys (arbitrary strings up to 64B in the current prototype) and (2) cache management: upon write/update operations, one should delete the cached content. When there is a read miss coming back from the host, one could insert the data into the cache. If there is a read hit, an in-networking processor is able to independently perform some computations.

3.6 Actor scheduler

The goal of the actor scheduler is to consolidate on the in-networking process as much computation as possible without triggering over-subscription and a loss in the communication bandwidth. Hence, an actor scheduler should: (1) fully utilize available cores for actor execution; (2) apply a lightweight bookkeeping mechanism to monitor actor computing costs; (3) migrate heavyweight actors to the host side.

As shown in Figure 2, we dedicates certain NIC cores for communication and use the rest for application workload execution. (In the case of LiquidIO, M=2 and N=10.) Note that all communication cores run a request dispatcher that pushes a fetched work queue item into an actor's mailbox. Instead of using a raw Ethernet packet, we use the work queue item as the request abstraction, and this includes packet metadata, packet buffer, and programmable NIC environment metadata (e.g., physical NIC port number).

We develop a light-weight work-conserving scheduler that runs inside hardware threads on all the NIC cores reserved for computing. These NIC computing cores share one global runnable actor queue. Each core walks through the queue in a round-robin fashion. When running an actor instance, it performs batched computations that processes N requests (if any) at once, which improves the locality. When the *exec_lock* is not set, we allow multiple cores to run an actor simultaneously. It is the programmer's responsibility to guarantee that an actor's private datastructures support concurrency. We note that: (1) our scheduler doesn't reply on hardware timers since it impacts the firmware performance of our programmable NIC; (2) we don't support scheduling priority and treat all offloaded actors equally.

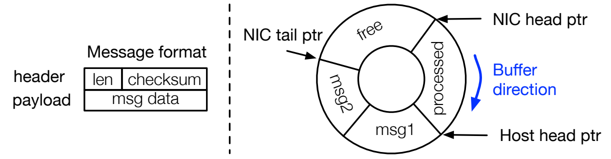


Figure 4: Receive circular buffer for the message passing.

The dispatcher keeps monitoring two metrics: (1) incoming bandwidth (BW_{in}) from the RX port; (2) expected average packet execution latency ($latency_{expected}$). We use a simple bitmap approach to calculate them. Specifically, we categorize packets into several buckets based on Ethernet frame size (e.g., 65-127B, 128-191B, 192-255B, etc.) and represent them as a counter bitmap. For each time epoch (T), we clear the bitmap at the beginning and collect data at the end. Based on this information, we know the total size of incoming packets ($Size_{total}$) and the total number of packets (N). Hence, $BW_{in} = \frac{Size_{total}}{T}$, $packet_{avg} = \frac{Size_{total}}{N}$, $latency_{expected} = \frac{BW_{in}}{N}$. The scheduler uses a similar approach to monitor: (1) consumed bandwidth ($BW_{consumed}$) of all located actors; (2) per-actor execution latency. Even though our profiling doesn't report accurate statistics, this approximation is effective enough and introduces little computing overheads.

When $BW_{in} > BW_{consumed}$, actors are not able to process incoming bandwidth. This results in two things: (1) actor mailbox size will build up, which increases the average and tail latencies for requests; (2) the size available for the packet buffer on the programmable NIC will be squeezed, affecting the performance of the iPipe networking stack. Therefore, when BM_{in} exceeds $BM_{consumed}$ by a threshold (we use a value of 10%), we will migrate an actor to the host side. A candidate for migration is an actor that has high execution latency (larger than $latency_{expected}$). Our scheduler takes a greedy approach: instead of walking through the entire actor queue and picking the heaviest one, the scheduler migrates any candidate actor whose execution latency is higher than the threshold. We apply a cold migration approach: the scheduler will first moves the mailbox, and push the global shared objects to the other side (Figure 3). Then, it will delete the actor from the queue and release the actor's resources. Note that local memory and software managed cache objects are discarded during the migration. This is consistent with the programming guideline that one should always using DSOs to store persistent state and take advantage of the other two object types to improve the performance of state access. Also, iPipe assumes that there is an actor implementation on the host side with the same functionality.

3.7 Cross PCIe message passing tier

We use a message passing mechanism to communicate between the host and in-networking processors. Applications create a set of I/O channels (depending on the communication frequency), and each channel includes two circular buffers for sending and receiving. A buffer is unidirectional and stored in the host memory. The in-networking processor writes into the receive buffer, and the host core polls it to detect new messages. The send buffer works in reverse.

Figure 4 shows the receive buffer. The unused portions of the buffer (marked as "free" and "processed") are zeroed out to detect new messages. We fix each message entry at 2KB and split messages when they are larger. Since the DMA engine may not write the message contents in a monotonic sequence (unlike RDMA NICs), we add a 4B checksum to verify the integrity of the whole message. The host core uses the header to figure out the length of the message and delivers the content to the application layer. We perform at most three retries if the checksum doesn't match and then discard the message.

We considered two ways to synchronize the header pointer between the host and the NIC: one is using lazy-update and the other is to piggyback the updates along with transmitted messages. As host send behavior is application dependent, we take the first approach, wherein the host notifies the in-networking processor when it has processed half of the buffer via a dedicated message. To fully use the PCIe bandwidth, we perform DMA reads/writes in a batched way. The message buffer is NUMA aware (on a multi-socket server) in order to reduce cross QPI remote memory traffic. Note that the polling overhead will increase when we add more I/O channels to handle frequent host invocations and high bandwidth. Table 1 shows the messaging API list.

RKV use case: When there is a key-value read that cannot be processed by the memtable actor resident on the NIC, the actor will forward this request to the host SSTable actor via the write primitive. After processing the request, the SSTable actor will deliver the results via the host-side *msg_write* primitive, and our NIC-side runtime will pull it using the read primitive.

3.8 Shim networking stack

iPipe takes advantage of packet processing accelerators (e.g., hardware managed resource pools) to build its networking stack. Specifically, it polls incoming packets from the traffic manager and transmits outgoing packets via the packet output unit (i.e., PKO). The traffic manager fetches working items (including packet data and metadata) from a packet input module (i.e., PKI). PKI and PKO use a hardware managed pool to efficiently allocate and free packet buffers.

iPipe also performs Layer2/Layer3 protocol processing, such as packet encapsulation/decapsulation, fragmentation, checksum verification, etc. While building a packet, iPipe uses the DMA scatter-gather technique to combine the header and payload if they are not colocated. Note that, because of the bounded computing limitation, we perform datagram processing on the NIC and push reliability mechanisms (e.g., flow/congestion control, loss recovery) to the host application layer. When passing the request to the host, the stack is able to direct flows to different I/O channels based on the host core locality.

RKV use case: The Paxos actor makes extensive use of the networking API to communicate between the leader and followers.

3.9 Dynamic resource management

We allow multiple applications to simultaneously use the programmable NIC and share its computational resources. iPipe therefore needs to provide resource allocation mechanisms that support the execution of multiple concurrent applications and also provide performance and security isolation.

iPipe provides fair bandwidth allocation and tries to minimize spikes in execution latency. It applies the weighted max-min [53] fairness method and sets up rate limiters based on the assigned bandwidths. Weights are initialized by the programmers and adjusted when there is a request latency spike (which is detected using the exponentially weighted moving average [52] method). Each weight adjustment operation will trigger a bandwidth and rate limiter recalculation for all actors. For instance, suppose three applications have weights 2, 2, and 4, along with bandwidth demands 4Gbps, 6Gbps, and 2Gbps. At first, the algorithm allocates bandwidth as 2.5Gbps, 2.5Gbps, and 5Gbps based on the weights. Since app3's demand is satisfied while those of app1 and app2 are not, in the second round, the algorithm will re-assign the additional 3Gbps from app3 to app1 and app2 using their weights. Finally, app1 and app2 obtain $2.5Gbps + 3Gbps \times \frac{2}{2+2} = 4Gbps$, respectively.

We rely on the hardware paging mechanism of the Cavium cnMIPS processor [8] to isolate execution environments of the NIC firmware, iPipe runtime, and actors. The host side works similarly, except that it considers the actors that pulls requests from the message ring. iPipe only considers one resource type (i.e, link bandwidth) and our future work will take multiple resources (e.g., NIC cores, local and global memory, as well as bandwidth) into considering using the DRF [23] algorithm.

3.10 Host-side runtime

Host-side runtime behaves similarly as the NIC-side one, except that: (1) The host actor scheduler runs as a pthread; (2) The host processor never migrates actors to the NIC side proactively because it has no information about the load on an in-networking processor, which is sensitive to the computation over-subscription. Instead, we use a pull-based mechanism initiated by the NIC; (3) It holds the full I/O ring content; and (4) The host side networking stack provides packet

manipulation APIs with no packet buffer management as it primarily uses the messaging tier to forward packets.

4 Implementation

This section describes the use of DMA primitives and provides some guidelines on designing the data structures on the NIC side. We then describe two other applications built with iPipe.

4.1 Choice of DMA primitives

A NIC-side in-networking processor communicates with host processors using DMA engines through the PCIe bus. PCIe is a packet switched network with 500ns-2us latency and 7.87 GB/s theoretical bandwidth per Gen3 x4 endpoint (which is the one used in our server). Its performance is usually impacted by many runtime factors. With respect to latency, DMA engine queueing delay, PCIe request size and its response ordering, PCIe completion word delivery, and host DRAM access costs will all slow down PCIe packet delivery [24, 30, 46]. With respect to throughput, PCIe is limited by transaction layer packet (TLP) overheads (i.e., 20-28 bytes for header and addressing), the maximum number of credits used for flow control, the queue size in DMA engines, and PCIe tags used for identifying unique DMA reads.

Generally, a DMA engine provides two kinds of primitives: blocking accesses, which wait for the DMA completion word from the engine, and non-blocking ones, which allow the processing core to continue executing after sending the DMA commands into the command queue. Figures 5 and 6 show our performance characterizations of the LiquidIO NIC. Non-blocking operations insert a DMA instruction word into the queue and don't wait for completion. Hence, its read/write latency and throughput are independent of packet size. We take advantage of the low overhead non-blocking primitives while implementing the message passing tier (Section 3.7).

For blocking operations, reads perform worse than writes (i.e., a 1KB host memory read takes 2.10us while the write is 1.28us). This is because read is a non-posted PCIe transaction, which requires the host to return a completion TLP to indicate that the transaction was successfully processed. On the other hand, write is a posted PCIe transaction and has no such requirements. Moreover, both latency and throughput become worse when we increase the payload size beyond 128B since larger packets are split into several small PCIe transactions. When multiple cores simultaneously issue DMA requests, contention will happen at either the DMA command queue or the PCIe root complex. We carefully batch multiple requests to fully utilize the PCIe bandwidth. In our case, we use blocking operations when passing shared objects between the two sides (Section 3.4). Note that, since blocking primitives can be optimized by overlapping the communication with some local computation, we provide a special primitive ***blocked_dma_read/write_overlap(func, msg)***, which executes the function *func* while waiting for the completion word for *msg*.

4.2 Index caching

An in-networking processor is a wimpy one since (1) it doesn't utilize a sophisticated microarchitecture for memory level parallelism (i.e., few missing status holding registers (MSHRs)) and (2) it has limited per-core cache. Therefore, its local memory access is slower than that of host memory. One will see diminishing returns in terms of latency if there is significant NIC-local memory dereferencing while traversing a data structure. To demonstrate this point, We use the SkipList as an example and measure the operation latency difference between host-local and NIC-local accesses as we increase the number of list elements. For 1024K elements, host processors outperform the NIC by 0.84us and 1.17us for INSERT/FIND, approaching one PCIe transaction latency. This indicates that when designing a data structure for an in-networking processor, one should use the index caching technique to accelerate the data access. iPipe's local memory objects and software cached objects can be used for this purpose. This is a widely used method for building systems using one-sided RDMA [50, 18].

4.3 More applications on iPipe

We also use iPipe to build two other datacenter applications: a distributed transactions system and a real-time analytics engine. For brevity, we provide just an overview of the underlying algorithms and how we develop them on iPipe.

Distributed Transactions. We build a distributed transactions system that uses optimistic concurrency control and two-phase commit for distributed atomic commit, following the design used by other systems [54, 29]. Note that we choose to not add a replication layer as we try to eliminate the application function overlap with our replicated key-value store. The application includes a coordinator and participants that run a transaction protocol. Assume the set of keys read and written by the transaction are R (read set) and W (write set). The protocol works as follows: Phase1(read and lock): the coordinator reads values for the keys in the R set and locks the keys in the W set. If any key in R or W is already locked, the coordinator aborts the transaction and replies with the failure status; Phase2 (validation): after locking the write set, the coordinator checks the version of its read set by issuing a second read. If any key is locked or its version has changed after the first phase, the coordinator aborts the transaction; Phase3 (log): the coordinator logs the key/value/version information into its coordinator log and then sends a reply to the client with the result; Phase4 (commit): the coordinator sends commit messages to nodes that holds the W set. After receiving this message, the participant will update the key/value/version, as well as unlock the key.

In iPipe, we implement the coordinator and participant as

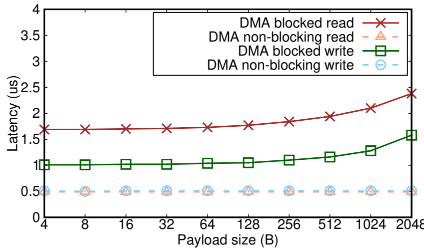


Figure 5: Per-core blocked/non-blocking DMA read/write latency when increasing payload size.

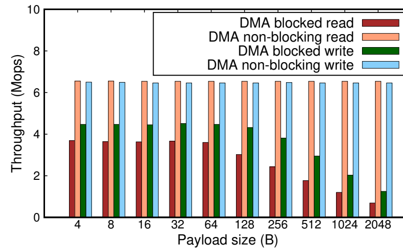


Figure 6: Per-core blocked/non-blocking DMA read/write throughput when increasing payload size.

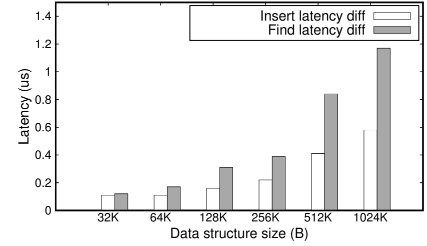


Figure 7: INSERT and FIND latency difference between a host processor and an in-networking processor when increasing the number of node of a SkipList.

actors running on the NIC. The key storage abstractions required to implement the protocol are the coordinator log [16] and the data store, which we realize using a traditional extensible hashtable [25]. Both of these are realized using distributed shared objects. We also cache responses from outstanding transactions. When the log reaches a storage limit, a new actor will be created on the NIC, and the old log will be migrated to the host side for checkpointing.

Real-time Analytics. Data processing pipelines use a real-time analytics engine to gain instantaneous insights into vast and frequently changing datasets. We acquired the implementation of FlexStorm [31] and extended its functionality. All data tuples are passed through three workers: *filter*, *counter*, and *ranker*. The filter applies a pattern matching module to discard uninteresting data tuples. The counter uses a sliding window and periodically emits a tuple to the ranker. Ranking workers sort incoming tuples based on count and then emit the *top-n* data to an aggregated ranker. Each worker uses a topology mapping table to determine the next worker to which the result should be forwarded. There are also threads for de-multiplexing and multiplexing packets as tuples are communicated from one worker to another.

In iPipe, we implement the three workers as actors. Filter actor is a stateless one. Counter uses the software managed cache for statistics. Ranker is implemented using a distributed shared object, and we consolidate all top-n data tuples into one object. During the execution, we find that when the ranker actor performs quicksort to order tuples, the computation significantly impacts the NIC's ability to receive new data tuples. Our runtime will then migrate this actor to the host side.

5 Evaluation

Our evaluations aim to answer the following questions:

1. Compared with host-side execution, what are the latency savings of running parts of an applications on an in-networking processor (§ 5.2, $ 5.4)?
2. With the iPipe framework, how much throughput can an application achieve for a given number of host CPU cores? Alternately, how much fewer host CPU cores are required to achieve a certain throughput (§ 5.3)?
3. How effective is iPipe at managing concurrent use of the programmable NIC by different applications (§ 5.5)?
4. Can iPipe scale to multiple programmable NICs ($ 5.6)?

5.1 Prototyped system

We build iPipe on the Cavium LiquidIOII [10] (described before) with commercial off-the-shelf (COTS) Supermicro 1U/2U servers. Our iPipe runtime spreads across the NIC firmware and host system with 4061 LOCs and 3183 LOCs, respectively. On the NIC side, we take advantage of low-level primitives provided by the Cavium Development Kit [9]. The firmware applies the run-to-completion model across all 12 cores. Our runtime is embedded into the firmware, which starts after the firmware environment initialization. As described before, iPipe dedicates 1 core for the message tier, 1 core for the TX/RX, and the rest for application execution. The firmware, runtime, and applications reside in different *bootmem* (which are large chunks of system memory abstracted by the firmware). On the host side, we use pthreads for runtime execution and allocate 1GB pinned hugepages for the message ring. Each runtime thread periodically polls requests from the channel and performs actor executions. It transition to idling C-states when there is no more work. Programmers use the C langugage to build applications. Our three workloads, real-time analytics (RTA), distributed transactions (DT), replicated key-value store (RKV), built with iPipe have 1583 LOCs, 2225 LOCs, and 2133 LOCs, respectively, and we compare with similar implementations that use DPDK.

Our workload generator is also implemented using DPDK and invokes operations in a closed-loop manner. For RTA, each packet contains one fixed size (128B) data tuple. For the distributed transaction, each request is a multi-key read-write transaction, and each has 2 reads and 1 write. For RKV, we generate the <key,value> pair in each packet, with the following characteristics: 95% read, zipf distribu-

tion (skewness=0.99), and 1 million keys (used in previous work[47, 37]).

Our testbed runs on an 8-node cluster, attached to an Arista DCS-7050S ToR switch. Seven of them are 1U boxes and the last one is a 2U server (all from Supermicro). Each of the 1U machines has one 12-core E5-2680 v3 processor at 2.5GHz and 64GB DDR4 memory. Four of them are equipped with a dual-port 10Gbps Intel XL710 NIC and the remaining three have Cavium LiquidIOII installed. We use the 2U server for the multi-NIC evaluations, where it has two 8-core Intel E5-2620 v4 processors at 2.1GHz, 128GB memory, and 7 Gen3 PCIe slots. All servers are equipped with a Seagate HDD, and we apply the Intel_pstate governor for power management.

5.2 Latency of iPipe primitives

We characterize the latency of various iPipe primitives using a set of microbenchmarks. Specifically, we use a pointer workload to evaluate the local/remote memory reads/writes. The networking SEND/RECV are characterized using a simple ECHO workload. We measure the messaging layer primitive performance by repeatedly reading from/writing to one I/O channel. Our microbenchmarks yield the following observations.

First, the in-networking processor has a deep non-uniform memory subsystem with the following attributes: (1) local DRAM access is more than 10X faster than remote host access; (2) remote host writes outperform remote host reads. This motivates why we introduce three kinds of data objects (i.e, distributed shared object, local memory objects, and software managed cache). When building applications, one should lay out the application working set carefully using these three types of objects. Second, the iPipe stack takes at most 0.13μs and 0.22 μs for packet receiving and sending, irrespective of the packet size (since the hardware managed packet buffer for each packet is always fixed at 2KB). Third, the messaging primitives consume 0.10μs and 0.22μs for sending/receiving packets with 1KB payloads. The host memory synchronization overheads are amortized. We also measured the throughput and found that it could saturate the 10Gbps bandwidth for all message sizes.

5.3 Throughput and core usage

Figure 8 and Table 3 reports the host CPU cores used when achieving the maximum throughput. We also presents the average CPU usage for our iPipe host core. Since the core goes to idle periodically when there are no incoming requests, it doesn't fully use the host core. Compared with the DPDK case, iPipe saves 1 core for the RTA worker, 2.66 cores for the DT coordinator, 2 cores for the DT participant, 1.43 cores for the RKV leader, and 1.77 core for the RKV follower.

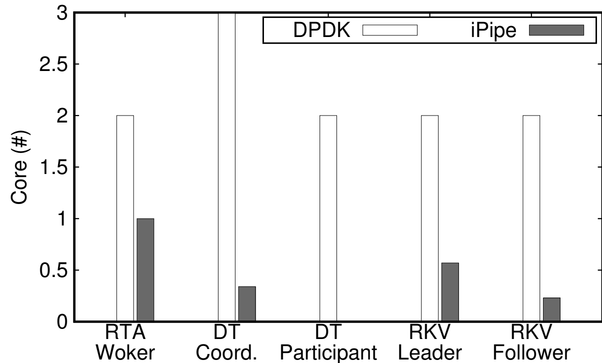


Figure 8: Host used CPU cores comparison between DPDK and iPipe on three different applications. For iPipe case, we report the average CPU usage.

5.4 Application latency

Latency v.s. throughput. Figure 9 presents the latency (observed by clients) and throughput as we increase the client request rate, comparing among three cases (i.e., DPDK with 1 core, DPDK that can achieve the highest throughput, and iPipe).

First, at low to medium request rate, NIC-side offloading reduces request execution latency by 3.0μs, 15.8μs, 9.3μs of three applications. These benefits come from three facts: (1) PCIe transaction savings (since parts of application logic directly run on the NIC); (2) fast networking primitives along with hardware accelerated buffer managements; (3) wimpy computations that have similar execution performance when running on the host processor. DT benefits the most as both the coordinator and participants run on the in-networking process and host CPU doesn't involve in the performance critical path.

Second, when only using one CPU core with DPDK, iPipe improves 27.8%, 75.0%, and 26.2% throughput of three applications, respectively. This is because the NIC cores brings in more computing power. When using more host cores, both cases achieve the same throughput.

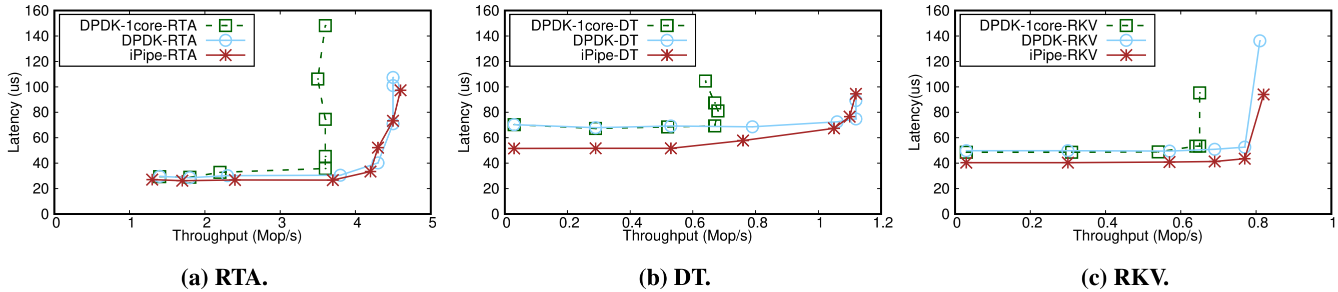
99th tail latency. Figure 10 reports the average and tail (99th) latency of three applications when achieving 80% of the maximum throughput. iPipe reduces 12.1μs, 6.0μs, and 9.7μs for three applications, respectively. This is not only due to the fast request processing (discussed above), but also because that (1) the actor scheduler running on the networking processor will migrate heavy actors when it hurts latency; (2) the traffic manager starts to throttle the traffic when the NIC core cannot fetch packets fast enough.

5.5 Multi-tenancy and iPipe

We evaluate the iPipe dynamic resource management (DRM) under various application consolidation scenarios shown in Figures 11 and 12. Also, in this experiment, we

Primitives	4B	8B	16B	32B	64B	128B	256B	512B	1024B
iPipe lmem read (ns)	15.8	20.3	31.7	35.0	36.7	70.9	100.8	101.7	110.0
iPipe lmem write (ns)	16.7	23.3	31.7	35.0	38.3	90.8	105.8	106.7	111.7
iPipe nstack-RX (us)	0.12	0.12	0.12	0.12	0.12	0.13	0.13	0.13	0.13
iPipe nstack-TX (us)	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.22
iPipe rmem read (us)	1.46	1.46	1.47	1.47	1.51	1.54	1.60	1.71	1.86
iPipe rmem write (us)	0.86	0.85	0.86	0.86	0.88	0.89	0.94	0.99	1.13
iPipe send message (us)	0.01	0.02	0.04	0.05	0.06	0.09	0.09	0.10	0.10
iPipe recv message (us)	0.13	0.13	0.13	0.13	0.13	0.13	0.14	0.14	0.16

Table 2: Latency of iPipe NIC-side primitives under different data size. The size of nstack and messaging are payload. The cache line size of the networking processor is 128B.



(a) RTA. **(b) DT.** **(c) RKV.**

Figure 9: Latency versus throughput for three applications, compared among DPDK-1core, DPDK, and iPipe cases.

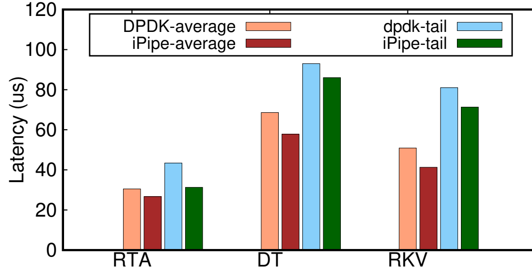


Figure 10: Average and tail latency when achieving 80% maximum throughput, compared between DPDK and iPipe cases.

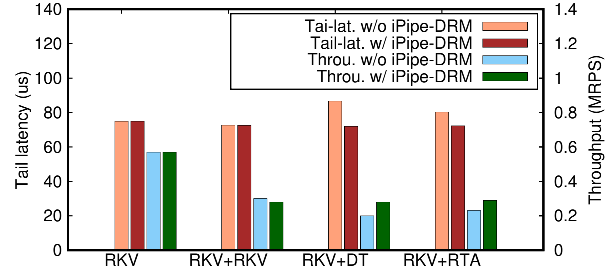


Figure 11: Tail latency and throughput of the RKV application, when consolidating with RKV, DT, and RTA, compared between w/ and w/o iPipe DRM.

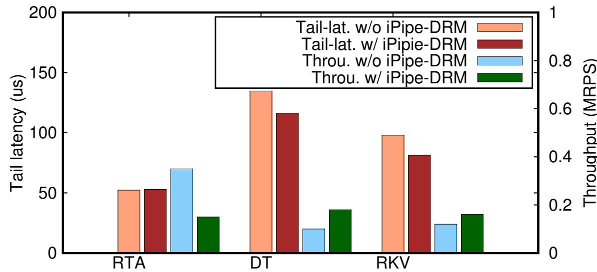


Figure 12: Tail latency and throughput comparison between w/ and w/o iPipe dynamic resource management when consolidating three applications.

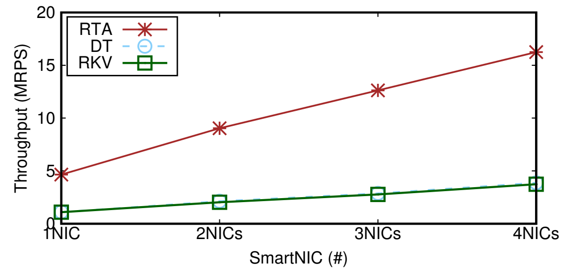


Figure 13: Maximum throughput (MRPS) when adding more SmartNICs for three applications. This experiment uses a 2U Supermicro beefy server.

	RTA	DT	RKV
DPDK	4.65	1.12	1.15
iPipe	4.65	1.17	1.09

Table 3: Maximum throughput (in million requests/second) comparison between DPDK and iPipe cases.

configure all three workloads with the same request size. There are three observations. First, iPipe DRM can mitigate execution interference in terms of tail latency. When consolidating the RKV with DT and RTA, DRM helps reduces 14.7μs and 8.0μs tail latency, respectively (We don't see a tail latency increase when two RKVs run together). When co-locating three of them, on average, DRM saves 11.5μs tail latency of three applications. Second, iPipe DRM is able to guarantee fair bandwidth allocation. Considering the application scenario (Figure 12), without DRM, RTA achieves nearly twice (0.35MRPS) of throughput as the other two (0.10/0.12MRPS). When applying DRM, three applications achieve similar share (0.15/0.18/0.16MRPS). Third, DRM incurs little execution overheads. When running RKV alone w/ and w/o DRM (Figrue 11), its latency and throughput varies little.

5.6 Multiple NICs and iPipe

This section evaluates the scalability of iPipe in terms of supporting multiple programmable NICs. We take the 2U Supermicro box as the testbed. As we have only one big server, when deploying three applications, we run all actors locally (without external communication). As a result, we reduce the number of data shards for both DT and RKV (to avoid the computation overloading). Each programmable NIC has its own I/O channel. To eliminate the case that computations are bounded by the host CPU cores, we attach two programmable NICs to each 8-core processor. As shown in Figure 13, the throughput of each of the three applications scales well with more NICs. For example, RKV increases from 1.09MRPS, to 2.03MRPS, 2.77MRPS, 3.74 MRPS with 2/3/4 NICS.

6 Related work

Programmable NIC acceleration. There are some recent studies that use emerging programmable NICs for application acceleration. For example, ClickNP [36] provides a flexible and high performance networking function framework. It applies the Click [32] dataflow programming model and allows joint CPU/NIC processing on network functions (through PCIe I/O channels). Researchers have also explored the possibility of dynamically offloading stateful middlebox functions onto a programmable NIC [43]. HotCocoa [6] aims to offload the entire congestion control algorithm onto the programmable NIC using a set of hardware abstractions. Floem [1] provides a set of programming abstractions (e.g., logic queue, per-packet state, and caching construct) that helps programmers to easily split applications across the CPU and the NIC. iPipe differs from the above systems in the following ways: (1) it targets RPC-based datacenter workloads and provides an actor-based programming model; (2) it examines the offloading of general-purpose application-specific computations onto a programmable NIC, with the runtime system ensuring that NIC's hardware constraints are not violated; (3) it supports multi-tenancy and concurrent use of the programmable NIC by multiple applications.

RDMA-based datacenter applications. Recent years have seen a growing use of RDMA in datacenter environments due to its low-latency, high-bandwidth, and low CPU utilization benefits. These applications include key-value store system [42, 19, 28], DSM system [44, 19], database and transactional system [12, 20, 51, 29]. Generally, RDMA provides fast data access capabilities but provides limited opportunities to reduce the host CPU computing load. While one-sided RDMA operations allow applications to bypass remote server CPUs, they are hardly used in general distributed systems given the narrow set of remote memory access primitives associated with them. In contrast, *iPipe* provides a framework to offload simple but general computations onto programmable NICs. It does however borrow some techniques approaches from related RDMA projects (e.g., lazy updates for the send/receive rings in FaRM [19]).

In-network computations. The recently RMT switches [7] and smart NICs [41, 10, 45, 40, 48] enable programmability along the packet data plane. Researchers have proposed the use of in-network computation, where one can offload compute operations from endhosts into these network devices in order to reduce datacenter traffic and improve applications performance. For example, IncBricks [38] is an in-network caching fabric with some basic computing primitives. NetCache [27] is another in-network caching design, which uses a packet-processing pipeline on a Barefoot Tofino switch to detect, index, store, invalidate, and serve key-value items. DAIET [3] conducts data aggregation (for MapReduce and TensorFlow) along the network path using programmable switches. In contrast to these projects that use programmable switches and middleboxes, *iPipe* targets programmable NICs.

7 Conclusion

This paper proposes iPipe, a framework that allows programmers to develop datacenter applications using a NIC-side in-networking processor. iPipe addresses the programmability, offloading constraints, and multi-tenancy challenges by supporting an actor programming model and providing a set of utility APIs via the iPipe runtime system. We build three datacenter workloads using iPipe and prototype them on a commodity programmable NICs. Our evaluations show that by offloading lightweight computations to the NIC, one can achieve latency benefits as well as host CPU savings.

References

[1] Floem: Programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, 2018), USENIX Association.

[2] AGHA, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] AMEDEO, S., IBRAHIM, A., ABDULLA, A., MARCO, C., AND PANOS, K. In-network computation is a dumb idea whose time has come.

[4] APACHE. The Apache Cassandra Database. http://cassandra.apache.org, 2017.

[5] APACHE. The Apache HBase. https://hbase.apache.org, 2017.

[6] ARASHLOO, M. T., GHOBADI, M., REXFORD, J., AND WALKER, D. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets-XVI, ACM, pp. 108–114.

[7] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 99–110.

[8] CAVIUM. Cavium OCTEON Multi-core Processor. http://www.cavium.com/octeon-mips64.html, 2017.

[9] CAVIUM. Octeon Development Kits. http://www.cavium.com/octeon_software_develop_kit.html, 2017.

[10] CAVIUM. Cavium LiquidIO SmartNICs. https://cavium.com/pdfFiles/LiquidIO_II_CN78XX_Product_Brief-Rev1.pdf, 2018.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)* (Seattle, WA, 2006), USENIX Association.

[12] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[13] CISCO. The New Need for Speed in the Datacenter Network. http://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-734328.pdfdf, 2015.

[14] CISCO. Cisco Global Cloud Index: Forecast and Methodology, 2015-2020. http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf, 2016.

[15] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX Association, pp. 261–264.

[16] CRISTIAN, F., ET AL. Coordinator log transaction execution protocol, 1990.

[17] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 205–220.

[18] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 401–414.

[19] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), pp. 401–414.

[20] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles* (2015), ACM, pp. 54–70.

[21] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 371–384.

[22] FIRESTONE, D. Hardware-Accelerated Networks at Scale in the Cloud. https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf, 2017.

[23] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 323–336.

[24] GOLDHAMMER, A., AND AYER JR, J. Understanding performance of pci express systems. *Xilinx WP350, Sept 4* (2008).

[25] HANSON, T. D. Uthash Hashtable. https://troydhanson.github.io/uthash/, 2017.

[26] HEWITT, C., BISHOP, P., AND STEIGER, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[27] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 121–136.

[28] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 295–306.

[29] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 185–201.

[30] KAMINSKY, A. K. M., AND ANDERSEN, D. G. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference* (2016), p. 437.

[31] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81.

[32] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000), 263–297.

[33] KUNG, H.-T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS) 6*, 2 (1981), 213–226.

[34] LAMPORT, L., ET AL. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[35] LEVELDB. LevelDB Key-Value Store. http://leveldb.org, 2017.

[36] LI, B., TAN, K., LUO, L. L., PENG, Y., LUO, R., XU, N., XIONG, Y., CHENG, P., AND CHEN, E. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 1–14.

[37] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.

[38] LIU, M., LUO, L., NELSON, J., CEZE, L., KRISHNAMURTHY, A., AND ATREYA, K. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ACM, pp. 795–809.

[39] MANGODB. MangoDB Document-oriented database. https://www.mongodb.com, 2017.

[40] MELLANOX. Mellanox Innova Ethernet Adapter. http://www.mellanox.com/page/products_dyn?product_family=228&mtag=programmable_adapter_cards_l, 2017.

[41] MELLANOX. Mellanox BuleField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic, 2018.

[42] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference* (2013), pp. 103–114.

[43] MOON, Y., PARK, I., LEE, S., AND PARK, K. Accelerating flow processing middleboxes with programmable nics.

[44] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference* (2015), pp. 291–305.

[45] NETRONOME. Netronome Agilio SmartNIC. https://www.netronome.com/products/agilio-cx/, 2018.

[46] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 327–341.

[47] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[48] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., ET AL. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (2014), IEEE, pp. 13–24.

[49] SRINIVASAN, S., AND MYCROFT, A. Kilim: Isolation-typed actors for java. In *European Conference on Object-Oriented Programming* (2008), Springer, pp. 104–128.

[50] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[51] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 87–104.

[52] WIKIPEDIA. Expotential Moving Average. https://en.wikipedia.org/wiki/Moving_average, 2018.

[53] WIKIPEDIA. Max-min fairness. https://en.wikipedia.org/wiki/Max-min_fairness, 2018.

[54] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 263–278.