Offloading Distributed Applications onto SmartNICs using iPipe

Ming Liu
University of Washington

Tianyi Cui
University of Washington

Henry Schuh
University of Washington

Arvind Krishnamurthy
University of Washington

Simon Peter
The University of Texas at Austin

Karan Gupta
Nutanix

Abstract

Emerging Multicore SoC SmartNICs, enclosing rich computing resources (e.g., a multicore processor, onboard DRAM, accelerators, programmable DMA engines), hold the potential to offload generic datacenter server tasks. However, it is unclear how to use a SmartNIC efficiently and maximize the offloading benefits, especially for distributed applications. Towards this end, we characterize four commodity SmartNICs and summarize the offloading performance implications from four perspectives: traffic control, computing capability, onboard memory, and host communication.

Based on our characterization, we build iPipe, an actor-based framework for offloading distributed applications onto SmartNICs. At the core of iPipe is a hybrid scheduler, combining FCFS and DRR-based processor sharing, which can tolerate tasks with variable execution costs and maximize NIC compute utilization. Using iPipe, we build a real-time data analytics engine, a distributed transaction system, and a replicated key-value store, and evaluate them on commodity SmartNICs. Our evaluations show that when processing 10/25Gbps of application bandwidth, NIC-side offloading can save up to 3.1/2.2 beefy Intel cores and lower application latencies by 23.0/28.0 μs.

CCS Concepts

• **Networks** → **Programmable networks**; **In-network processing**; • **Hardware** → **Networking hardware**;

Keywords

SmartNIC, Distributed applications

ACM Reference Format:
Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs using iPipe. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341302.3342079

1 Introduction

Multicore SoC (system-on-a-chip) SmartNICs have emerged in the datacenter, aiming to mitigate the gap between increasing network bandwidth and stagnating CPU computing power [13, 14, 19]. In the last two years, major network hardware vendors have released different SmartNIC products, such as Mellanox's BlueField [43], Broadcom's Stingray [7], Marvell (Cavium)'s LiquidIO [42], Huawei's IN5500 [24], and Netronome's Agilio [47]. They not only target acceleration of protocol processing (e.g., Open vSwitch [52], TCP offloading, traffic monitoring, and firewall), but also bring a new computing substrate into the data center to expand the server computing capacity at a low cost: SmartNICs usually enclose computing cores with simple microarchitectures that make them cost-effective.

Generally, these SmartNICs comprise a multicore, possibly wimpy, processor (i.e., MIPS/ARM), onboard SRAM/DRAM, packet processing and domain-specific accelerators, and programmable DMA engines. The different components are connected by high-bandwidth coherent memory buses or interconnects. Today, most of these SmartNICs are equipped with one or two 10/25GbE ports, and 100/200GbE products are imminent. These computing resources allow hosts to offload generic computations (including complex algorithms and data structures) without sacrificing performance and program generality. ***The question we ask in this paper is how to use SmartNICs efficiently to maximize offloading benefits for distributed applications?***.

There have been some recent research efforts that offload networking functions onto FPGA-based SmartNICs (e.g., ClickNP [38], AzureCloud [20]). They take a conventional domain-specific acceleration approach that consolidates most application logic onto FPGA programmable logic blocks. This approach is applicable to a specific class of applications that exhibit sufficient parallelism, deterministic program logic, and regular data structures that can be synthesized efficiently on FPGAs. Our focus, on the other hand, is to target distributed applications with complex data structures and algorithms that cannot be realized efficiently on FPGA-based SmartNICs.

Towards this end, we perform a detailed performance characterization of four commodity SmartNICs (i.e., LiquidIOII CN2350, LiquidIOII CN2360, BlueField 1M332A, and Stingray PS225). We break down a SmartNIC into four architectural components – traffic control, computing units, onboard memory, and host communication – and use microbenchmarks to characterize their performance. The experiments identify the resource constraints that we have to be cognizant of, illustrate the utility of hardware acceleration units, and provide guidance on how to efficiently utilize the resources.

We design and implement the iPipe framework based on our characterization study. iPipe introduces an actor programming model for distributed application development. Each actor has its own self-contained private state and communicates with other actors via messages. Our framework provides a distributed memory object abstraction and enables actor migration, responding to dynamic workload changes and ensuring the delivery of line-rate traffic. A central piece of iPipe is the actor scheduler that combines FCFS

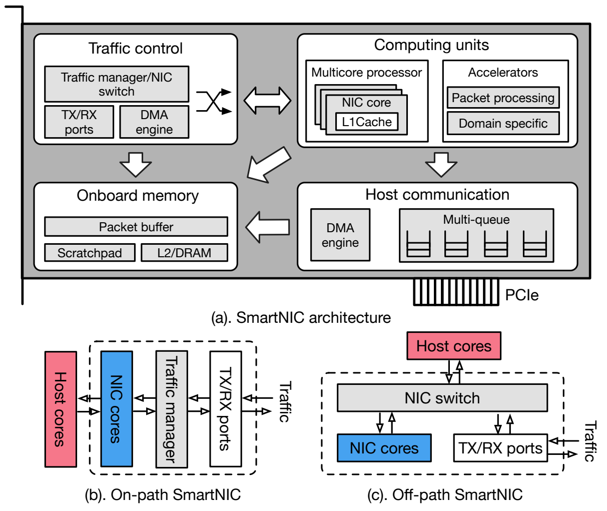


Figure 1: Architectural block diagram for a Multicore SoC SmartNIC and packet processing for the two types of SmartNICs.

(first come first serve) and DRR (deficit round robin) based processor sharing, which tolerates tasks with variable execution costs and maximizes a SmartNIC's resource utilization. iPipe allows multiple actors from different applications to coexist safely on the SmartNIC, protecting against actor state corruption and denial-of-service attacks. Taken together, iPipe's mechanisms enable dynamic and workload-aware offloading of arbitrary application logic, in contrast to prior work that focused on static offloading of specialized tasks (e.g., Floem [53] and ClickNP [38]).

We prototype iPipe and build three applications (i.e., a data analytics engine, a transaction processing system, and a replicated key-value store) using commodity 10GbE/25GbE SmartNICs. We evaluate the system using an 8-node testbed and compare the performance against DPDK-based implementations. Our experimental results show that we can significantly reduce the host load for real-world distributed applications; iPipe saves up to 3.1/2.2 beefy Intel cores used to process 25/10Gbps of application bandwidth, along with up to 23.0μs and 28.0μs savings in request processing latency.

2 Characterizing Multicore SoC SmartNICs

This section provides detailed performance characterizations of Multicore SoC SmartNICs. We explore their computational capabilities and summarize implications that guide the design of iPipe.

2.1 Multicore SoC SmartNICs

A Multicore SoC SmartNIC consists of four major parts (as shown in Figure 1-a): (1) computing units, including a general-purpose ARM/MIPS multicore processor, along with accelerators for packet processing (e.g., deep packet inspection, packet buffer management) and specialized functions (e.g., encryption/decryption, hashing, pattern matching, compression); (2) onboard memory, enclosing fast self-managed scratchpad and slower L2/DRAM; (3) traffic control module that transfers packets between TX/RX ports and the packet buffer, accompanied by an internal traffic manager or NIC switch that delivers packets to NIC cores; (4) DMA engines for communicating with the host.

Table 1 lists the HW/SW specifications of four commercial Multicore SoC SmartNICs evaluated in this paper. They represent different design tradeoffs regarding performance, programmability, and flexibility. The first two LiquidIOII SmartNICs enclose an OCTEON [9] processor with a rich set of accelerators but run in the context of a light-weight firmware. Programmers have to use native hardware primitives to process raw packets, issue DMA commands, and trigger accelerator computations. BlueField and Stingray cards run a ARM Cortex-A72 [5] processor and host a full-fledged operating system. They offer a lower barrier for application development, and one can use traditional Linux/DPDK/RDMA stacks to communicate with local and external endpoints. The BlueField card even has NVDIMM support for fault-tolerant storage. Current SmartNICs typically have link speeds of 10/25 GbE, and 100/200 GbE units are starting to appear. Generally, a SmartNIC is a bit more expensive than a traditional dumb NIC. For example, a 10/25GbE SmartNIC typically costs 100~400$ more than a corresponding standard NIC [62].

Based on how SmartNIC cores interact with traffic, we further categorize SmartNICs into two types: ***on-path*** and ***off-path*** SmartNICs. The cores for on-path SmartNICs (Figure 1-b) are on the packet communication path and hold the capability to manipulate each incoming/outgoing packet. LiquidIOII CN2350/CN2360 are both on-path SmartNICs. Off-path SmartNICs (Figure 1-c), deliver traffic flows to host cores (bypassing NIC cores) based on forwarding rules installed on a *NIC switch*. Mellanox BlueField and Broadcom Stingray are examples of off-path SmartNICs. Both NIC vendors are further improving the programmability of the NIC switch (e.g., Broadcom TruFlow [8], Mellanox $ASAP^2$ [44]).

For both types of SmartNICs, host processing is the same as that with standard NICs. On the transmit path (where a host server sends out traffic), the host processor first creates a DMA control command (including the instruction header and packet buffer address) and then writes it into a command ring. The NIC DMA engine then fetches the command and data from host memory and writes into the packet buffer (located in the NIC memory). NIC cores on on-path SmartNICs pull in incoming packets (usually represented as work items), perform some processing, and then deliver them to TX/RX ports via the DMA engine. For off-path ones, packets are directly forwarded to either NIC cores or TX/RX ports based on switching rules. Receive processing (where a host server receives traffic from the SmartNIC) is similar but performed in the reverse order.

2.2 Performance Characterization

We characterize four Multicore SoC SmartNICs (listed in Table 1) from four perspectives: traffic control, computing units, onboard memory, host communication.

2.2.1 Experiment setup.

We use Supermicro 1U/2U boxes as host servers for both the client and server and an Arista DCS-7050S/Cavium XP70 ToR switch for 10/25GbE network. The client is equipped with a dumb NIC (i.e., Intel XL710 for 10GbE and Intel XXV710-DA2 for 25GbE). We insert the SmartNIC on one of the PCIe 3.0 ×8 slots in the server. The server box has a 12-core E5-2680 v3 Xeon CPU running at 2.5GHz with hyperthreading enabled, 64GB DDR3 DRAM, and 1TB Seagate HDD. When evaluating BlueField and Stingray cards, we use a 2U Supermicro server with two

SmartNIC model	Vendor	Processor	BW	L1	L2	DRAM	Deployed SW	Nstack	To/From host
LiquidIOII CN2350 [42]	Marvell	cnMIPS 12 core, 1.2GHz	2× 10GbE	32KB	4MB	4GB	Firmware	Raw packet	Native DMA
LiquidIOII CN2360 [42]	Marvell	cnMIPS 16 core, 1.5GHz	2× 25GbE	32KB	4MB	4GB	Firmware	Raw packet	Native DMA
BlueField 1M332A [43]	Mellanox	ARM A72 8 core, 0.8GHz	2× 25GbE	32KB	1MB	16GB	Full OS	Linux/DPDK/RDMA	RDMA
Stingray PS225 [7]	Broadcom	ARM A72 8 core, 3.0GHz	2× 25GbE	32KB	16MB	8GB	Full OS	Linux/DPDK/RDMA	RDMA

Table 1: Specifications of the four SmartNICs used in this study. BW = bandwidth. Nstack = networking stack.

8-core Intel E5-2620 v4 processors at 2.1GHz, 128GB memory, and 7 Gen3 PCIe slots.

We take the DPDK pkt-gen as the workload generator and augment it with the capability to generate different application layer packet formats at the desired packet interval. We use the Linux RDMA perftest utility [50] to measure the performance for various verbs. We report end-to-end performance metrics (e.g., latency/throughput), as well as readings from microarchitectural counters (e.g., IPC, L2 cache misses per kilo instruction or MPKI).

2.2.2 Traffic control.

As described above, traffic control is responsible for delivering packets to either NIC computing cores or the host. Here, we use an ECHO server that entirely runs on a SmartNIC to answer the following questions regarding computation offloading: (1) how many NIC cores are sufficient to saturate the link speed for different packet sizes, and how much computing capacity is available for "offloaded applications"? (2) what are the synchronization overheads in supplying packets to multiple NIC cores?

Figures 2 and 3 present experimental data for 10GbE LiquidIOII CN2350 and 25GbE Stingray PS225. When packet size is 64B/128B, neither NIC can achieve full link speed even if all NIC cores are used. However, when packet size is 256B/512B/1024B/1500B(MTU), the LiquidIOII requires 10/6/4/3 cores to achieve line rate, while Stingray needs 3/2/1/1 cores. Stingray uses fewer cores due to its higher core frequency (3.0GHz v.s. 1.20GHz). These measurements quantify the packet transmission costs, which is the default execution tax of a SmartNIC. Figure 4 further presents the achieved bandwidth as we increase the per-packet processing latency of 256B and 1024B size packets when we use all the NIC cores on the two SmartNICs. The maximum tolerated latency limit (or computing headroom) for these packet sizes is 2.5/9.8us and 0.7/2.6us for 10GbE LiquidIOII and 25GbE Stingray, respectively.

On-path SmartNICs often enclose a unique hardware traffic manager that can feed packets to NIC cores in an efficient way. Figure 5 reports the average and tail (p99) latency when achieving the maximum throughput for four different packet sizes using 6 and 12 cores. Interestingly, the latencies do not increase as we increase the core count; compared to the 6 core case, the 12 core experiments only add 4.1%/3.4% average/p99 latency on average across the four scenarios. These measurements indicate that the hardware traffic manager is effective at providing a shared queue abstraction with little synchronization overhead for packet buffer management.

Design implications: ***I1:*** Packet transmission through a SmartNIC core incurs a nontrivial cost. Further, the packet size distribution of incoming traffic significantly impacts the availability of computing cycles on a Multicore SmartNIC. One should monitor the packet (request) sizes to adaptively offload application logic. ***I2:*** For on-path SmartNICs, hardware support reduces synchronization overheads and enables scheduling paradigms where multiple workers can efficiently pull incoming traffic from a shared queue.

2.2.3 Computing units.

To explore the execution behavior of the computing units, we use the following: (1) a microbenchmark suite comprising of representative in-network offloaded workloads from recent literature; (2) low-level primitives to trigger the domain-specific accelerators. We conduct experiments on the 10GbE LiquidIOII CN2350 and report both system and microarchitecture results in Table 3 (in Appendix A).

We observe the following results. First, the execution times of the offloaded tasks vary significantly from 1.9/2.0us (replication and load balancer) to 34.0/71.0us (ranker/classifier). Second, low IPC (instruction per cycle)[1] or high MPKI (misses per kilo-instructions) are indicators of high computing cost, as in the case of the rate limiter, packet scheduler, and classifier. Tasks with high MPKI are memory-bound tasks. As they are less likely to benefit from the complex microarchitecture on the host, they might be ideal candidates for offloading. Third, SmartNIC accelerators provide fast domain-specific processing appropriate for networking/storage tasks. For example, the MD5/AES engine is 7.0X/2.5X faster than the one on the host server (even using the Intel AES-NI instructions). However, invoking an accelerator is not free since the NIC core has to wait for execution completion and also incurs cache misses (i.e., higher MPKI) in feeding data to the accelerator. Batching can amortize invocation costs but could result in tying up NIC cores for extended periods. Other SmartNICs (e.g., BlueField and Stingray) display similar characteristics.

SmartNICs also enclose specialized accelerators for packet processing. Consider the LiquidIOII ones (CN2350/CN2360), for example. It has packet input (PKI) and packet output units (PKO) for moving data between MAC and the packet buffer and a hardware-managed packet buffer equipped with fast packet indexing. When compared with the SEND operations for two host-side kernel-bypass networking stacks, DPDK and RDMA, the hardware assisted messaging on LiquidIOII shows 4.6X and 4.2X speedups, respectively, when averaged across the different packet sizes (as shown in Figure 6).

Design implications: ***I3:*** The wimpy processor on a SmartNIC presents cheap parallelism (in terms of cost), and one should take advantage of such computing power by running applications with low IPC or high MPKI. Further, the offloading framework should be able to handle tasks with a wide range of execution latencies without compromising the packet forwarding latency for on-path SmartNICs and the execution latency of lightweight operations for both types of SmartNICs. ***I4:*** Accelerators are critical resources on a SmartNIC. For example, one can offload networking protocol processing, such as checksum calculation, tunneling encapsulation/decapsulation, using the packet processing units. The crypto engines (e.g., AES, MD5, SHA-1, KASUMI) enable various encryption, decryption, and authentication tasks. The compression unit and pattern matching block will benefit inline data reduction and

[1]Note that the cnMIPS OCTEON [9] is a 2-way processor and the ideal IPC is 2.

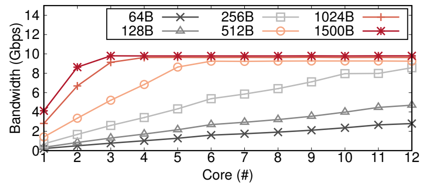


Figure 2: SmartNIC's bandwidth as we vary the number of NIC cores on the 10GbE LiquidIOII CN2350.

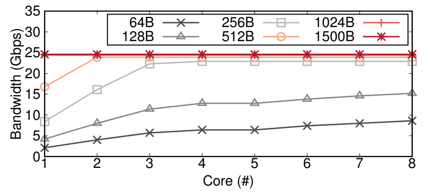


Figure 3: SmartNIC's bandwidth as we vary the number of NIC cores on the 25GbE Stingray PS225.

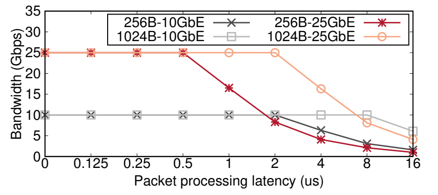


Figure 4: SmartNIC's bandwidth given a per-packet processing cost on the 10GbE LiquidioII CN2350 and the 25GbE Stingray PS225.

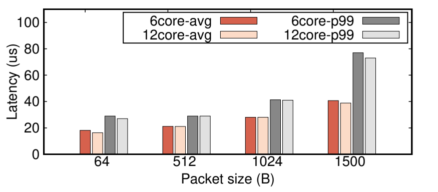


Figure 5: Average/p99 latency when operating at the maximum throughput on the 10GbE LiquidIOII CN2350.

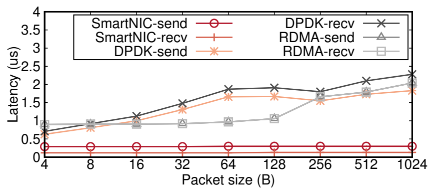


Figure 6: Comparison of send/recv latency on the 10GbE LiquidIOII CN2350 with RDMA/DPDK operations on the host.

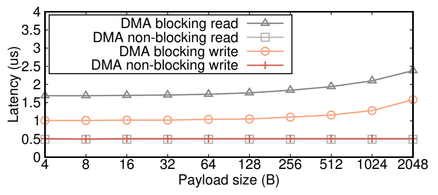


Figure 7: Per-core blocking/non-blocking DMA read/write latency as we increase the payload size.

	L1 (ns)	L2 (ns)	L3 (ns)	DRAM (ns)
LiquidIOII CNXX	8.3	55.8	N/A	115.0
BlueField 1M332A	5.0	25.6	N/A	132.0
Stingray PS225	1.3	25.1	N/A	85.3
Host Intel server	1.2	6.0	22.4	62.2

Table 2: Access latency of different levels of the memory hierarchies on the SmartNICs and the Intel server. The cache line sizes for LiquidIOII NICs are 128B while the rest are 64B. The performance of LiquidIOII CN2350 and CN2360 is similar.

flow/string classification, respectively. When using these accelerators, one should consider performing batched execution if necessary (at the risk of increasing queueing for incoming traffic).

2.2.4 Onboard memory.

Generally, a SmartNIC has five onboard memory resources in its hierarchy: (1) Scratchpad/L1 cache is per-core local memory. It has limited size (e.g., LiquidIO has 54 cache lines of scratchpad) with fast access speed. (2) Packet buffer. This is onboard SRAM along with fast indexing. On-path SmartNICs (like LiquidIOII) usually have hardware-based packet buffer management, while off-path ones (such as BlueField and Stingray) do not have a dedicated packet buffer region. (3) L2 cache, which is typically shared across all NIC cores. (4) NIC local DRAM, which is accessed via the onboard high-bandwidth coherent memory bus. Note that a SmartNIC can also read/write the host memory using its DMA engine, and this capability is evaluated in the next section.

We use a pointer chasing microbenchmark (with random stride distance) to characterize the access latency for different memory hierarchies for four SmartNICs and compare it with the host server. Table 2 illustrates that there is significant diversity in memory subsystem performance across SmartNICs. Also, the memory performance of many of the SmartNICs is worse than the host server (e.g., the access latency of SmartNIC L2 cache is comparable to the L3 cache on the host server), but the well-provisioned Stingray has performance comparable to the host.

Design implications: ***I5:*** On-path SmartNICs favor inline packet manipulation with the help of a hardware-based packet buffer manager. For stateful computation offloading, when the application working set exceeds the L2 cache size of a SmartNIC, executing memory-intensive workloads on the SmartNIC might result in a performance loss than running on the host.

2.2.5 Host communication.

A SmartNIC 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 x8 endpoint (which is the version used by all of our SmartNICs). Many runtime factors usually impact communication performance. 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 [21, 28, 48]. With respect to throughput, PCIe is limited by the transaction layer packet 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 7 and 8 show our performance measurements of the 10GbE LiquidIO CN2350. Non-blocking operations insert a DMA instruction word into the queue and do not wait for completion. Hence, the read/write latency of non-blocking operations are independent of the packet size. For blocking DMA reads/writes, a large message can fully utilize the PCIe bandwidth. For example, with 2KB payloads, one can achieve 2.1/1.4 GB/s per-core PCIe write/read bandwidth, outperforming the 64B case by 8.7X/6.0X. These measurements indicate that one should take advantage of the DMA scatter and gather technique to aggregate PCIe transfers.

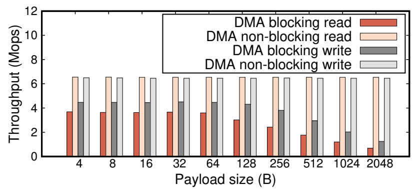


Figure 8: Per-core blocking/non-blocking DMA read and write throughput for different payload sizes.

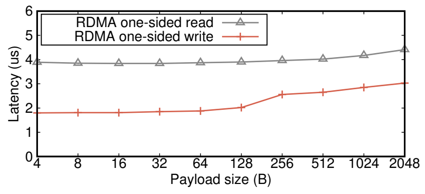


Figure 9: Per-core RDMA one-sided read and write latency for different payload sizes on the 25GbE BlueField 1M332A.

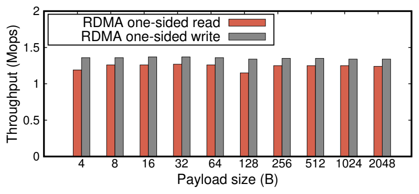


Figure 10: Per-core RDMA one-sided read and write throughput for different payload sizes on the 25GbE BlueField 1M332A.

Some SmartNICs (like BlueField and Stingray) expose RDMA verbs instead of native DMA primitives. We characterize the one-sided RDMA read/write latency from a SmartNIC to its host using the Mellanox BlueField card, as these operations resemble the DMA blocking operations. We find that RDMA primitives nearly double the read/write latency of blocking DMA ones (Figure 9). In terms of throughput, as shown in Figure 10, when message size is less than 256B, RDMA read/write only achieves a third of the per-core throughput of blocking DMA read/write. When the message is larger than 512B, RDMA and native DMA achieve similar performance.

Design implications: ***I6:*** There are significant performance benefits to using non-blocking DMA and aggregating transfers into large PCIe messages (via DMA scatter and gather). RDMA read/write verbs perform worse than native DMA primitives for small messages, likely due to software overheads.

3 iPipe framework

This section describes the design and implementation of our iPipe framework. We use the insights gathered from our characterization experiments to address the following challenges.

- **Programmability:** A commodity server equipped with a SmartNIC is a non-cache-coherent heterogeneous computing platform with asymmetric computing power. We desire simple programming abstractions that can be used for developing general distributed applications.
- **Computation efficiency:** There are substantial computing resources on a SmartNIC (e.g., a multicore processor, modest L2/DRAM, and plenty of accelerators), but one should use them efficiently. Inappropriate offloading could cause NIC core overloading, bandwidth loss, and wasteful execution stalls.
- **Isolation:** A SmartNIC can hold multiple applications simultaneously. One should guarantee that different applications cannot touch each others' state, there is no performance interference between applications, and tail latency increases, if any, are modest.

3.1 Actor programming model and APIs

iPipe applies an actor programming model [1, 23, 59] for application development. iPipe uses the actor-based model for two reasons. First, the actor model can support computing heterogeneity and hardware parallelism automatically. One can easily map an actor execution instance onto a physical computing unit, such as a NIC or host core. Second, actors have well-defined associated states and can be migrated between the NIC and the host dynamically. This attribute enables dynamic control over a SmartNIC's computing capabilities and allows the system to adapt to traffic workload characteristics. More importantly, unlike dataflow or pipeline models (as in Floem [53]), which are designed to support data-intensive control-light workloads, the actor-based model can support control-intensive, non-deterministic, and irregular communication patterns that arise in complex distributed applications.

An actor is a computation agent that performs two kinds of operations based on the type of an incoming message: (1) trigger an execution handler and manipulate its private state; (2) interact with other actors by sending messages asynchronously. Actors do not share memory. We choose message passing as the communication paradigm given the following considerations: (1) the communication latencies that we observe between the NIC and host are in the order of microseconds; (2) actors are independent entities without shared state. In our system, every actor has an associated structure with the following fields: (1) *init_handler* and *exec_handler* for state initialization and message execution; (2) *private_state*, which can use different data types (as described in Section 3.3); (3) *mailbox* is a multi-producer multi-consumer concurrent FIFO queue, which is used to store incoming asynchronous messages; (4) *exec_lock*, used to decide whether an actor can be executed on multiple cores; (5) some runtime information, such as *port*, *actor_id*, and a reference to the *actor_tbl*, which contains the communication addresses of other actors. An actor runs on one or more cores, and it is the programmers' responsibility to provide concurrency control for operating on an actor's private state.

The iPipe runtime enables the actor-based model by providing support for actor allocation/destruction, runtime scheduling of actor handlers, and transparent migration of actors and its associated state. (See Table 4 in the Appendix B.1 for the runtime API.) Specifically, iPipe has three key system components: (1) an actor scheduler that works across both SmartNIC and host cores and uses a hybrid FCFS/DRR scheduling discipline to enable execution of actor handlers with diverse execution costs; (2) a distributed object abstraction that enables flexible actor migration and supports a software managed cache to mitigate the cost of SmartNIC to host communications; (3) a security isolation mechanism that protects actor state from corruption and denial-of-service attacks. We describe these components below.

3.2 iPipe Actor Scheduler

iPipe schedules actor executions on both the SmartNIC and the host cores. The scheduler assigns actor execution tasks to computing cores and specifies a custom scheduling discipline for each actor task. In designing the scheduler, we not only want to maximize the computing resource utilization on the SmartNIC but also ensure that the computing efficiency does not come at the cost of increased tail latencies or compromising the NIC's ability to convey traffic. Recall that, in the case of on-path SmartNICs, all traffic is conveyed through SmartNIC cores, so executing actor handlers could adversely impact the latency and throughput of other traffic.

3.2.1 Problem formulation and background.

The runtime system executes on both the host and the SmartNIC, determines on which side an actor executes, and schedules the invocation of actor handlers. There are two critical decisions in the design of the scheduler: (a) whether the scheduling system is modeled as a centralized, single queue model or as a decentralized, multi-queue model, and (b) the scheduling discipline used for determining the order of task execution. We consider each of these issues below.

It is well-understood that the decentralized multi-queue model can be implemented without synchronization but suffers from temporary load imbalances leading to increased tail latencies. Fortunately, on-path SmartNICs enclose hardware traffic managers that provide support for a shared queue abstraction with low synchronization overheads (see Section 2.2.2). We, therefore, resort to using a centralized queue model on the SmartNIC and a decentralized multi-queue model on the host side, along with NIC-side support for flow steering. We discuss later how to build such a scheduler for off-path SmartNICs.

We next consider the question of what scheduling discipline to use and how that impacts the average and tail response times for scheduled operations (i.e., both actor handlers and message forwarding operations). Note that the response time or sojourn time is the total time spent, including queueing delay and request processing time. If our goal is to optimize mean response time, then Shortest Remaining Processing Time (SRPT) and its non-preemptive counterpart, Shortest Job First (SJF), are considered optimal regardless of the task size and interarrival time distributions [56]. However, in our setting, we also care about the tail response time; even if a given application can tolerate it, other applications sharing the resources might be impacted. Further, a high response latency also means that an on-path SmartNIC would not be able to perform its basic duty of forwarding traffic in a timely manner. If we were to consider minimizing the tail response time, then First Come First Served (FCFS) is considered optimal when task size distribution has low variance [61]. However, FCFS has been shown to perform poorly when the task size distribution has high dispersion or is heavy-tailed [3]. In contrast, Processor Sharing is considered optimal for high variance distributions [64].

In addition to the issues described above, the overall setting of our problem is unique. Our runtime manages the scheduling on both the SmartNIC and the host with the flexibility to move actors between the two computing zones. Crucially, we want to increase the occupancy on the SmartNIC, without overloading it or causing tail latency spikes, and the runtime can shed load to the host if necessary. Furthermore, given that the offloaded tasks will likely have different cost distributions (as we saw in our characterization experiments), we desire a solution that is suitable for a broad class of tasks.

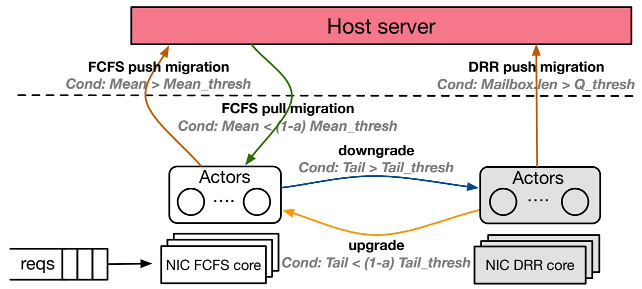


Figure 11: An overview of iPipe scheduler on the SmartNIC. *Cond* is the condition that triggers an operation.

3.2.2 Scheduling algorithm.

We propose a hybrid scheduler that: (1) combines FCFS and DRR (deficit round robin) service disciplines; (2) migrates actors between SmartNIC and host processors when necessary. Essentially, the scheduler takes advantage of FCFS for tasks that have low dispersion in their service times and delegates tasks with a higher variance in service times to a DRR service discipline. The scheduler uses DRR for high variance tasks as DRR is an efficient approximation of Processor Sharing in a non-preemptible setting [58]. Further, the scheduler places as much computation as possible on the SmartNIC and migrates actors when the NIC cannot promptly handle incoming packets. For performing these transitions, the scheduler collects statistics regarding the average and the tail execution latencies, actor-specific execution latencies, and queueing delays. We mainly describe the NIC-side scheduler below and then briefly describe how the host-side scheduler differs from it.

The scheduler works as follows. Initially, all scheduling cores start in FCFS mode, where they fetch packet requests from the shared incoming queue, dispatch requests to the target actor based on their flow information, and perform run-to-completion execution (see lines 5-6, 11-12 of ALG 1 in Appendix). When the measured tail latency of operations in the FCFS core is higher than *tail_thresh*, the scheduler downgrades the actor with the highest dispersion (a measure that we describe later) by pushing the actor into a DRR runnable queue and spawns a DRR scheduling core if necessary (lines 13-16 ALG 1). All DRR cores share one runnable queue to take advantage of the execution parallelism.

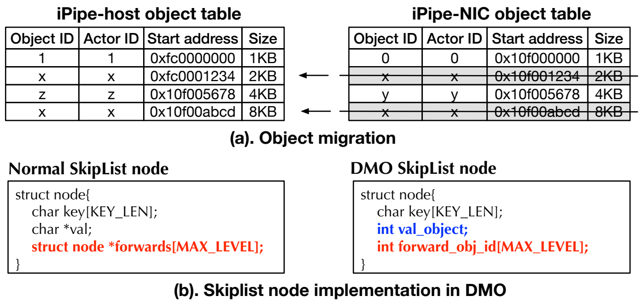
We next consider the DRR cores (see ALG 2 in Appendix). These cores scan all actors in the DRR runnable queue in a round-robin way. When the deficit counter of an actor is larger than its estimated latency, the core pops a request from the actor's mailbox and conducts its execution. The DRR quantum value for an actor, which is added to the counter in each round, is the maximum tolerated forwarding latency for the actor's average request size (obtained from the measurements in Section 2.2.2). When the measured tail latency of operations performed by FCFS is less than $(1-\alpha)tail\_thresh$ (where α is a hysteresis factor), the actor with the lowest dispersion in the DRR runnable queue is pushed back to the FCFS group (lines 10-12 of ALG 2).

Finally, when the scheduler detects that the mean request latency for FCFS jobs is larger than *mean_thresh*, it recognizes that there is a queue build-up at the SmartNIC. It then migrates to the host processor the actor that contributes the most to the NIC's processing load (lines 17-23 ALG 1). Similarly, when the mean request latency of the FCFS core group is lower than $(1-\alpha)mean\_thresh$ and if there is sufficient CPU headroom in the FCFS cores, the scheduler pulls from the host server the actor that will incur the least load back to the SmartNIC. To minimize synchronization costs, we use a dedicated core of the FCFS group for the migration tasks.

3.2.3 Bookeeping execution statistics. Our runtime monitors the following statistics to assist the scheduler: (1) Request execution latency distribution of all actors: We measure μ, the execution latency of each request (including its queueing delay) using microarchitectural time stamp counters. To efficiently approximate the tail of the distribution, we also track the standard deviation of the request latency σ and use $\mu+3\sigma$ as a tail latency measure. Note that this is close to the P99 measure for normal distributions. All of these estimates are updated using exponentially weighted moving averages (EWMA). (2) Per-actor execution cost and dispersion statistics. For each actor i, we track its request latency μ_i, the standard deviation of the latency σ_i, request sizes, and the request frequency. We use $\mu_i + 3\sigma_i$ as a measure of the dispersion of the actor's request latency. Again, we use EWMA to update these measures. (3) Per-core/per-group CPU utilization. We monitor the per-core CPU usage for the recent past and also use its EWMA to estimate its current utilization. The CPU group utilization (for FCFS or DRR) is the average CPU usage of the corresponding cores. Finally, we use measurements from our characterization study to set the thresholds *mean_thresh* and *tail_thresh*. We consider the MTU packet size at which the SmartNIC can sustain line rate and use the average and P99 tail latencies experienced by traffic forwarded through the SmartNIC as the corresponding thresholds (Section 2.2.2). These thresholds mean that we provide the same level of service with offloaded computations as when we have full line-rate processing of moderately sized packets.

3.2.4 FCFS and DRR core auto-scaling. All cores start in FCFS mode. When an actor is pushed into the DRR runnable queue, the scheduler spawns a core for DRR execution. When all cores in the DRR group is nearly fully used ($CPU_{DRR} \geq 95\%$ and the CPU usage of the FCFS group is less than $\frac{100\times(FCFS_{Core\#}-1)}{FCFS_{Core\#}}\%$, FCFS is able to spare a core for DRR service, and the scheduler will migrate a core to the DRR group. We use a similar condition for moving a core back to the FCFS group.

3.2.5 SmartNIC push/pull migration. We only allow the SmartNIC to initiate the migration operation since it is much more sensitive than the host processor in case of overloading. As described above, when there is persistent queueing (i.e., the mean response time is above a threshold), the scheduler will move an actor to the host side. In particular, actor migration is triggered based on the SmartNIC's processing load and the incoming request traffic. We pick the actor with the highest load (i.e., average execution latency scaled by frequency of invocation) for migration and ensure that the actor does not serve requests when it is being migrated. We perform migration in four steps. First, the actor transitions into the *Prepare* state and removes itself from the runtime dispatcher. An actor in the DRR group is also removed from the DRR runnable queue. The actor stops receiving incoming requests and buffers them in the iPipe runtime. Second, the actor finishes the execution



Object ID	Actor ID	Start address	Size
1	1	0xfc0000000	1KB
x	x	0xfc0001234	2KB
z	z	0x10f005678	4KB
x	x	0x10f00abcd	8KB

Object ID	Actor ID	Start address	Size
0	0	0x10f000000	1KB
~~x~~	~~x~~	~~0x10f001234~~	~~2KB~~
y	y	0x10f005678	4KB
~~x~~	~~x~~	~~0x10f00abcd~~	~~8KB~~

```
struct node{
    char key[KEY_LEN];
    char *val;
    struct node *forwards[MAX_LEVEL];
}
```

```
struct node{
    char key[KEY_LEN];
    int val_object;
    int forward_obj_id[MAX_LEVEL];
}
```

Figure 12: iPipe's distributed memory objects.

of its current tasks and transitions to the *Ready* state. Note that, for an actor in the DRR group, it finishes executing all the requests in its mailbox. Third, the scheduler moves the distributed objects of an actor to the host runtime, starts the host actor, and marks the NIC actor state as *Gone*. Finally, the scheduler forwards the buffered requests from the NIC to the host and rewrites their destination addresses. We will then label the NIC actor as *Clean*. Appendix B.3 provides more details on the migration process and its performance.

3.2.6 SmartNICs with no hardware traffic managers. We now consider SmartNICs that do not provide a shared queue abstraction to the NIC processor (especially off-path ones like BlueField and Stingray). There are two possible ways to overcome this limitation. One is to apply a dedicated kernel-bypass component (such as the IOKernel module in Shenango [51]) that processes all incoming traffic and exposes FCFS cores a single queue abstraction. This module will run on one or several NIC cores exclusively, depending on the traffic load. Another way is to add an intermediate shuffle layer across FCFS cores. Essentially, this shuffle queue is a single-producer, multiple-consumer one. This approach could cause load imbalances due to flow steering. Similar to ZygOS [54], one should also allow a FCFS core to steal other cores' requests when it becomes idle with no pending requests in its local queue. Note that both approaches bring in performance overheads, so future on-path/off-path SmartNICs could benefit from adding a hardware traffic manager to simplify NIC-side computation scheduling.

Summary: The scheduler manages the execution of actor requests on both the SmartNIC and the host. We use a hybrid scheme that combines FCFS and DRR on both sides. With the scheme outlined above, lightweight tasks with low dispersion are executed on the SmartNIC's FCFS cores, lightweight tasks with high dispersion are executed on the SmartNIC's DRR cores, and heavyweight tasks are migrated to the host. These decisions are performed dynamically to meet the desired average and tail response times.

3.3 Distributed memory objects

iPipe provides a distributed memory object (DMO) abstraction to enable flexible actor migration. Actors allocate and de-allocate DMOs as needed, and a DMO is associated with the actor that allocated it; there is no sharing of DMOs across actors. iPipe maintains an object table (Figure 12-a) on both sides and utilizes the local memory manager to allocate/de-allocate copies. At any given time, a DMO 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 (as shown in Section 2.2). Instead, iPipe would automatically move DMOs along with the actor, and all DMO read/write/copy-/move operations are performed locally. During the initialization phase, iPipe creates large equal-sized chunks of memory regions for each registered actor. On LiquidIOII SmartNICs, we realize this via the "global bootmem region" of the firmware. The iPipe runtime maintains the mapping between actor ID, its base address, and size. During execution, an actor can only allocate/reclaim/access objects within its region. When an actor consumes more memory than the framework provides, the DMO allocation will fail.

When using DMOs to design a data structure, one has to use object ID for indexing instead of pointers. This approach provides a level of indirection so that we can change the actual location of the object (say during migration to/from the host) without impacting an actor's local state regarding DMOs. As an example, in our replicated key-value store application (discussed later), we built a Skip List based Memtable via a DMO. As shown in Figure 12-b, a traditional Skip List node includes a key string, a value string, and a set of forwarding pointers. With a DMO, the key field is the same, but value and forwarding pointers are replaced by object IDs. When traversing the list, one will use the object ID to get the start address of the object, cast the type, and then read/write its contents.

Our characterization experiments (Section 2.2.4) have shown that the scratchpad memory on the LiquidIOII NIC provides the fastest performance but has limited resources. Instead of exposing this to applications, we decide to keep this memory resource internally and use it for storing the iPipe bookkeeping information (such as the framework and actor execution statistics).

3.4 Security Isolation

iPipe allows multiple actors to execute concurrently on a SmartNIC. iPipe handles the following two attacks: (1) actor state corruption, where a malicious actor manipulates other actors' states; (2) denial-of-service, where an actor occupies a SmartNIC core and violates the service availability of other actors. We first describe how to protect against these two attacks on the LiquidIOII SmartNICs, using a lightweight firmware, and then discuss how to apply similar techniques to other SmartNICs that have a full-fledged OS.

Actor state corruption. Since iPipe provides the distributed memory object abstraction to use the onboard memory DRAM, we rely on the processor paging mechanism to secure the object accesses. LiquidIOII CN2350/CN2360 SmartNICs employ a MIPS processor (which has a software-managed TLB) and a lightweight firmware for memory management. As discussed above in Section 3.3, we partition memory into regions and allocate each region to an actor. Invalid reads/writes from an actor causes a TLB miss and will trap into the iPipe runtime. If the address is not in the region, access is not granted.

Denial-of-service. A malicious actor might occupy a NIC core forever (e.g., execute an infinite loop), violating actor availability. We address this issue using a timeout mechanism. LiquidIOII CN2350/CN2360 SmartNICs include a hardware timer with 16 timer rings. We give each core a dedicated timer. When an actor executes, it clears out the timer and initializes the time interval. The timeout unit will traverse all timer rings and notify the NIC core when there is a timeout event. If a NIC receives the timeout notification, iPipe deregisters the actor, removes it from the dispatch table/runnable queue (if it is in the DRR group), and frees the actor resource.

SmartNICs with full OS. When there is an OS deployed on the SmartNIC (such as with BlueField/Stingray), iPipe will run each actor as individual threads in different address spaces. Thus, the hardware paging mechanism prevents actors from accessing the private state of other actors. Further, given the availability of a full OS, iPipe can employ a software timeout mechanism based on POSIX signals.

3.5 Host/NIC communication

We use a message-passing mechanism to communicate between the host and the SmartNIC. iPipe creates a set of I/O channels, and each one includes two circular buffers for sending and receiving. A buffer is unidirectional and stored in the host memory. One can also reuse the NIC driver buffer for such communication. NIC cores write into the receive buffer, and a host core polls it to detect new messages. The send buffer works in reverse. We use a lazy-update mechanism to synchronize the header pointer between the host and the NIC, wherein the host notifies the SmartNIC when it has processed half of the buffer via a dedicated message. We use batched non-blocking DMA reads/writes for the implementation. In order to avoid the case of a DMA engine not writing the message contents in a monotonic sequence (unlike RDMA NICs), we add a 4B checksum into the message header to verify the integrity of the whole message. Table 4 (in the Appendix B.1) shows the messaging API.

4 Applications built with iPipe

When using iPipe to develop distributed applications, there are four basic steps: (1) refactor the application logic into functionally independent components and represent them as actors; (2) define the actors' request formats and register them into the iPipe runtime; (3) allocate and initialize the actor private state (with the DMO APIs); (4) realize the actor *exec_handler* based on its application logic using iPipe provided utilities. We implement three distributed applications with iPipe: a replicated key-value store, a distributed transaction system, and a real-time analytics engine.

Replicated key-value store. Replicated key-value store (RKV) is a critical datacenter service, comprising of two system components: a *consensus protocol*, and a *key-value data store*. We use 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 values from other replicas if its 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.

For the key-value store, we implement the log-structured merge tree (LSM) that is widely used for many KV systems (such as Google's Bigtable [11], LevelDB [36], Cassandra [4]). An LSM tree accumulates recent updates in memory and serves reads of recently updated values from an in-memory data structure, flushes the updates to the disk sequentially in batches, and merges long-lived on-disk persistent data to reduce disk seek costs. There are two key system components: *Memtable*, a sorted data structure (i.e., Skip List) 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 exponentially with the level number. Low-level SSTables are merged into high-level ones via minor/major compact operations. Deletions are a special case of insertions wherein a deletion marker is added. Data retrieval might require multiple lookups on the Memtable and the SSTables (starting with level 0 and moving to high levels) until a matching key is found.

In iPipe, we implement RKV with four kinds of actors: (1) consensus actor, receives application requests and triggers the Multi-Paxos logic; (2) LSM Memtable actor, accumulates incoming writes/deletes and serves fast reads; (3) LSM SSTable read actor, serves SSTable read requests when requests are missing in the Memtable; (4) LSM compaction actor, performs minor/major compactions. The consensus actor sends a message to the LSM Memtable once during the commit phase. When requests miss in the Memtable actor, they are forwarded to the SSTable read actor. Upon a minor compaction, the Memtable actor migrates its Memtable object to the host and issues a message to the compaction actor. Our system has multiple shards, based on the NIC DRAM capacity. The two SSTable related actors are on the host because they have to interact with persistent storage.

Distributed Transactions. We build a distributed transaction processing system that uses optimistic concurrency control and two-phase commit for distributed atomic commit, following the design used by other systems [29, 65]. Note that we choose not to include a replication layer as we want to eliminate the application function overlap with our replicated key-value store. The application includes a coordinator and participants that run a transaction protocol. Given a read set (R) and a write set (W), the protocol works as follows: Phase 1 (read and lock): the coordinator reads values for the keys in R and locks the keys in W. If any key in R or W is already locked, the coordinator aborts the transaction and replies with the failure status; Phase 2 (validation): after locking the write set, the coordinator checks the version of keys in 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; Phase 3 (log): the coordinator logs the key/value/version information into its coordinator log; Phase 4 (commit): the coordinator sends commit messages to nodes that store the W set. After receiving this message, the participant will update the key/value/version, as well as unlock the key. When the coordinator receives acknowledgments, it sends a reply to the client with the result. The commit point of the transaction protocol is when the coordinator successfully records the transaction information in its log.

In iPipe, we implement the coordinator and participant as actors running on the NIC. The storage abstractions required to implement the protocol are the coordinator log [60] and the data store, which we realize using a traditional extensible hashtable [22]. Both of these are realized using distributed shared objects. We also cache responses from outstanding transactions. There is also a logging actor pinned to the host since it requires persistent storage access. When the coordinator log reaches a storage limit, the coordinator migrates its log object to the host side and sends a checkpointing message to the logging actor.

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 [30] and extended its functionality. All data tuples are passed through three workers: *filter*, *counter*, and *ranker*. The filter applies a pattern matching module [15] 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.

In iPipe, we implement the three workers as actors. Filter actor is a stateless one. Counter uses a software-managed cache for statistics. Ranker is implemented using a distributed shared object, and we consolidate all top-n data tuples into one object. Among them, ranker performs quicksort to order tuples, which could impact the NIC's ability to receive new data tuples when the network load is high. In such cases, iPipe will migrate the actor to the host side.

5 Evaluation

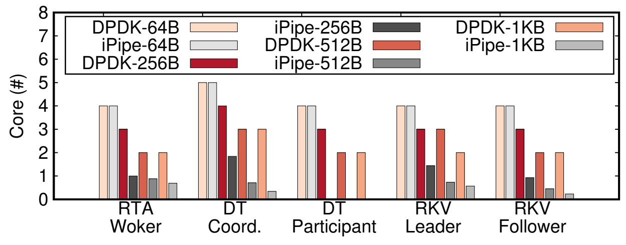
Our evaluations aim to answer the following questions:

- What are host CPU core savings when offloading computations using iPipe? (§5.2)
- What are the latency savings with iPipe? (§5.3)
- How effective is the iPipe actor scheduler? (§5.4)
- What is the overhead of the iPipe framework? ($5.5)
- When compared with the SmartNIC programming system Floem [53], what are the design trade-offs in terms of performance and programmability? (§5.6)
- Can we use iPipe to build other applications such as network functions? How does it perform? (§5.7)

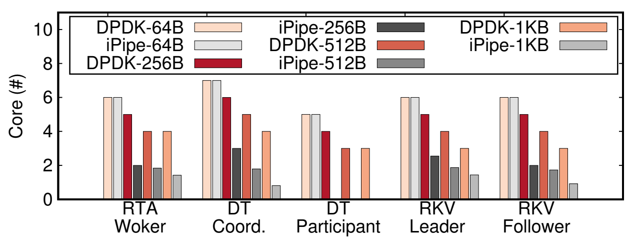
5.1 Experimental methodology

We use the same testbed as our characterization experiments in Section 2.2.1. For evaluating our application case studies, we mainly use the LiquidIOII CN2350/CN2360 (10/25 GbE) as we had a sufficient number of cards to build a small distributed testbed. We built iPipe into the LiquidIOII firmware using the Cavium Development Kit [10]. On the host side, we use pthreads for iPipe execution and allocate 1GB pinned huge pages for the message ring. Each runtime thread periodically polls requests from the channel and performs actor execution. The iPipe runtime spans across the NIC firmware andthe host system with 10683 LOC and 4497 LOC, respectively. To show the effectiveness of the actor scheduler, we also present results for the Stingray card.

Programmers use the C language to build applications (which are compiled with GNU toolchains for the SmartNIC and the host). Our three applications, real-time analytics (RTA), distributed transactions (DT), and replicated key-value store (RKV), built with iPipe have 1583 LOC, 2225 LOC, and 2133 LOC, respectively. We compare them with similar implementations that use DPDK. Our workload generator is implemented using DPDK and invokes operations in a closed-loop manner. For RTA, we generate the requests based on a Twitter dataset [35]. The number of data tuples in each request vary based on the packet size. For DT, each request is a multi-key

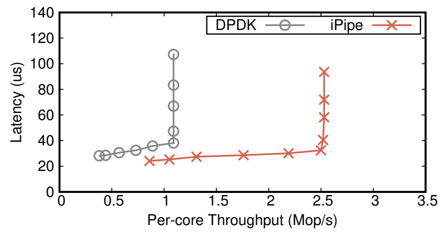


(a) 10GbE w/ LiquidIOII CN2350.

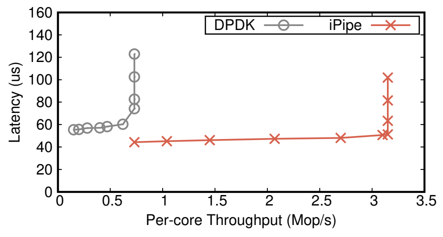


(b) 25GbE w/ LiquidIOII CN2360.

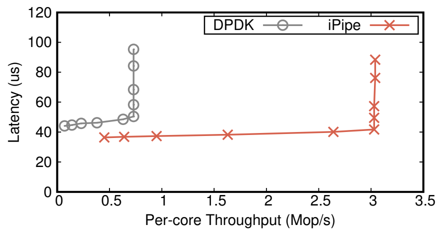
Figure 13: Number of CPU cores used by DPDK and iPipe as we vary the packet size on 10GbE and 25GbE networks.



(a) RTA.

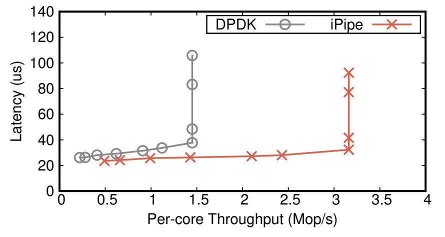


(b) DT.

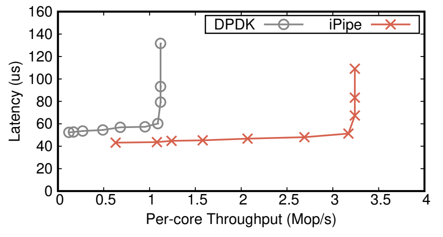


(c) RKV.

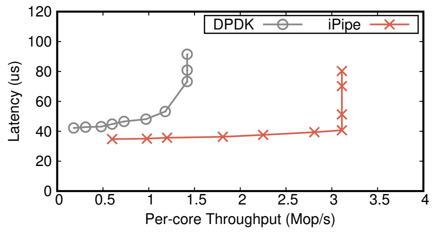
Figure 14: Latency versus per-core throughput for three applications on 10GbE network. Packet size is 512B.



(a) RTA.



(b) DT.



(c) RKV.

Figure 15: Latency versus per-core throughput for three applications on 25GbE network. Packet size is 512B.

read-write transaction including two reads and one write (as used in prior work [29]). For RKV, we generate the <key,value> pair in each packet, with the following characteristics: 16B key, 95% read and 5%write, zipf distribution with skew of 0.99, and 1 million keys (following the settings in prior work[39, 49]). For both DT and RKV, the value size increases with the packet size.

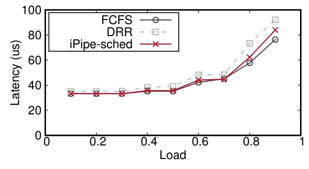
We deploy each of the applications on three servers, equipped with SmartNICs in the case of iPipe and standard Intel NICs in the case of DPDK. The RTA application runs an RTA worker on each server, the DT application runs coordinator logic on one server and participant logic on two servers, and the RKV application involves a leader node and two follower nodes.

5.2 Host core savings

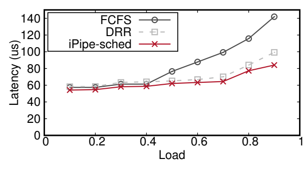
We find that we can achieve significant host core savings by offloading computations to the SmartNIC. Figure 13 reports the average host server CPU usage of three applications when achieving the maximum throughput for different packet sizes under 10/25GbE networks. First, when packet size is small (i.e., 64B), iPipe will use all NIC cores for packet forwarding, leaving no room for actor execution. In this case, one will not save host CPU cores. Second, host CPU usage reduction is related to both packet size and bandwidth. Higher link bandwidth and smaller packet size bring in more packet level parallelism. When the SmartNIC is able to absorb enough requests for execution, one can reduce host CPU loads significantly. For example, applications built on iPipe save 3.1, 2.6, and 2.5 host cores for 256/512/1KB cases, on average across three applications using the 25GbE CN2360 cards. Such savings are marginally reduced with the 10GbE CN2350 ones (i.e., 2.2, 1.8, 1.8 core savings). Among these three applications, DT participant saves the most since it is able to run all its actors on the SmartNIC, followed by the DT coordinator, RTA worker, RKV follower, and RKV leader.

5.3 Latency versus Throughput

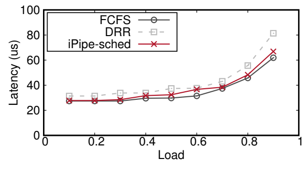
We next examine the latency reduction and per-core throughput increase provided by iPipe and find that SmartNIC offloading provides considerable benefits. Figures 14 and 15 report the results comparing DPDK and iPipe versions of the applications, when we configure the system to achieve the highest possible throughput with the minimal number of cores. When calculating the per-core throughput of three applications, we use the CPU usage of RTA worker, DT coordinator, and RKV leader to account for fractional



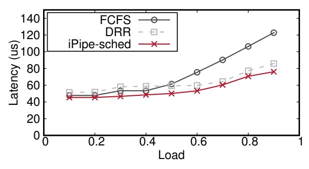
(a) Low dispersion on 10GbE LiquidIOII CN2350.



(b) High dispersion on 10GbE LiquidIOII CN2350.



(c) Low dispersion on 25GbE Stingray.



(d) High dispersion on 25GbE Stingray.

Figure 16: P99 tail latency at different networking loads for 10GbE LiquidIOII CN2350 and 25GbE Stingray. We consider both low and high dispersion distributions for request execution costs.

core usage. First, under 10GbE SmarNICs, applications (RTA, DT, and RKV) built with iPipe outperform the DPDK ones by 2.3X, 4.3X, and 4.2X, respectively, as iPipe allows applications to offload some of the computation to the SmartNIC. The benefits diminish a little under the 25GbE setup (with 2.2X, 2.9X, and 2.2X improvements) since actors running on the host CPU receive more requests and require more CPU power. Second, at low to medium request rates, NIC-side offloading reduces request execution latency by 5.7μs, 23.0μs, 8.7μs for 10GbE and 5.4μs, 28.0μs, 12.5μs for 25GbE, respectively. Even though the SmartNIC has only a wimpy processor, the iPipe scheduler keeps the lightweight, fast-path tasks on the NIC and moves the heavyweight, slow ones to the host. As a result, PCIe transaction savings, fast networking primitives, and hardware-accelerated buffer management can help reduce the fast path execution latency. DT benefits the most as (1) both the coordinator and the participants mainly run on the SmartNIC processor; (2) the host CPU is only involved for the logging activity.

P99 tail latency. We measured the tail latency (P99) when achieving 90% of the maximum throughput for the two link speeds. For the three applications, iPipe reduces tail latency by 7.3μs, 11.6μs, 7.5μs for 10GbE and by 3.4μs, 10.9μs, 12.8μs for 25GbE. This reduction is not only due to fast packet processing (discussed above), but also because iPipe's NIC-side runtime guarantees that there is no significant queue build up.

5.4 iPipe actor scheduler

We evaluate the effectiveness of iPipe's scheduler, comparing it with standalone FCFS and DRR schedulers under two different request cost distributions: one is exponential with low dispersion; the other one is bimodal-2 with high dispersion. We choose two SmartNICs (i.e., 10GbE LiquidIOII CN2350 and 25GbE Stingray) representing the cases where the scheduling runtime uses firmware hardware threads and OS pthreads, respectively. The workload generator is built using packet traces obtained from our three real-world applications, and it issues requests assuming a Poisson process. We measure the latency from the client. The mean service times of the exponential distribution on the two SmartNICs (i.e., LiquidIOII and Stingray) is 32μs and 27μs, while b1/b2 of the bimodal-2 distribution is 35μs/60μs and 25μs/55μs.

Figure 16 shows the P99 tail latency as we increase the network load for four different cases. For the low dispersion one, iPipe's scheduler behaves similar to FCFS but outperforms DRR. Under 0.9 networking load, iPipe can reduce 9.6% and 21.7% of DRR's tail latency for LiquidIOII and Stingray, respectively. For the high dispersion one, iPipe's scheduler is able to tolerate the request execution variation and serve short tasks in time, outperforming the other two. For example, when the networking load is 0.9, iPipe can reduce 68.7% (61.4%) and 10.9% (12.9%) of the tail latency for FCFS and DRR cases on LiquidIOII (Stingray). In this case, our approximate tail latency threshold (measured via $\mu+3\sigma$) for LiquidIOII and Stingray is 52.8μs and 44.6μ, respectively. Thus, requests with latencies higher than the threshold are processed on the iPipe DRR cores.

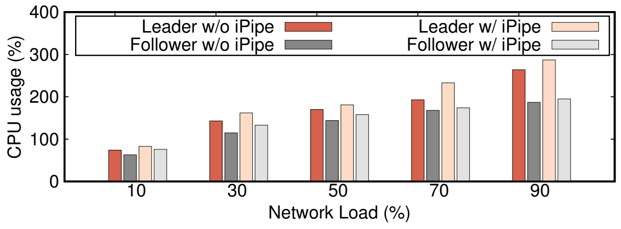


Figure 17: Host CPU usage of RKV leader and follower with and without iPipe under different network loads for a 10GbE. Packet size is 512B.

5.5 iPipe framework overheads

We evaluate the CPU overhead of the iPipe framework by comparing two host-only implementations of RKV (one with iPipe and one without iPipe). To make a fair comparison, when running these two versions, we pin the communication thread to the same core using the same epoll interface. We generate 512B requests and gradually increase the networking load. Figure 17 reports the host CPU utilization of the RKV leader and the follower when achieving the same throughput. On average, iPipe consumes 12.3% and 10.8% more CPU cycles for RKV leader and follower, respectively. Overall, iPipe brings in three kinds of overheads: message handling, DMO address translation when accessing objects, and the cost of the iPipe scheduler that orchestrates traffic and maintains statistics of execution costs. Since the message handling is not unique to iPipe, the other two parts dominate the above-measured overheads.

5.6 Comparison with Floem

Floem [53] is a programming system aimed at easing the programming effort for SmartNIC offloading. It applies a data-flow language to express packet processing and proposes a few programming abstractions, such as logic queue and per-packet state. iPipe also has related concepts, such as message rings and packet metadata. However, compared with iPipe, the key difference is that the language runtime of Floem does not use the SmartNIC's computing power efficiently. First, the offloaded elements (computations) on Floem are stationary, no matter what the incoming traffic is. Note, however, that under high network traffic load comprising of small packets, Multicore SoC SmartNICs have no room for application computation (as shown in Section 2.2.2). In iPipe, under such settings, we will migrate the computation to the host side. Second, the common computation elements of Floem mainly comprise of simple tasks (like hashing, steering, or bypassing). Complex ones are performed on the host side. This approach misses the opportunity of using the cheap computing parallelism and domain-specific accelerators on the SmartNIC. In contrast, iPipe, can be used to manage and offload complex operations, and the runtime will dynamically schedule them in the right place.

We take the real-time analytics (RTA) workload, and compare its Floem and iPipe implementations. With the same experimental setup, Floem-RTA achieves at most 1.6Gbps/core (in the best case), while iPipe-RTA can achieve 2.9Gbps. As described above, this is because iPipe can offload the entire actor computation while Floem utilizes a NIC-side bypass queue to mitigate the multiplexing overhead. For the small packet size case (i.e., 64B), iPipe-RTA delivers 0.6Gbps/core, outperforming Floem by 88.3%, since iPipe moves all the actors to the host and uses all NIC cores for packet forwarding, while Floem's static policy persists with performing the computations on the SmartNIC.

5.7 Network functions on iPipe

The focus of iPipe is to accelerate distributed applications with significant complexity in program logic and maintained state. For network functions with easily expressed states (or even stateless ones) that have sufficient parallelism, FPGA-based SmartNICs are an appropriate fit. We now consider how well iPipe running on multicore SmartNICs can approximate FPGA-based SmartNICs for such workloads. We built two network functions with iPipe (i.e., Firewall and IPSec gateway) and evaluated them on the 10/25GbE LiquidIOII cards. For the firewall, we use a software-based TCAM implementation matching wildcard rules. Under 8K rules and 1KB packet size, the average packet processing latency ranges from 3.65μs to 19.41μs as we increase the networking load. These latencies are higher than an FPGA based solution (i.e., 1.23~1.6μs reported in [38]). We also implemented an IPSec datapath that processes IPSec packets with AES-256-CTR encryption and SHA-1 authentication. We take advantage of the crypto engines to accelerate packet processing. For 1KB packets, iPipe achieves 8.6Gbps and 22.9Gbps bandwidth on the 10/25 GbE SmartNIC cards, respectively. These results are comparable to the ClickNP ones (i.e., 37.8Gbps under 40GbE link speed). In other words, if one can use the accelerators on a Multicore SoC SmartNIC for implementing the network functions, one can achieve performance comparable to FPGA based ones.

6 Related work

SmartNIC acceleration. In addition to Floem [53], ClickNP [38] is another framework using FPGA-based SmartNICs for network functions. It uses the Click [33] dataflow programming model and statically allocates a regular dataflow graph model during configuration, whereas iPipe can move computations based on runtime workload (e.g., request execution latency, incoming traffic). There are a few other studies that use SmartNICs for application acceleration. For example, KV-Direct [37] is an in-memory key-value store system, which runs key-value operations on the FPGA and uses the host memory as a storage pool.

In-network computations. Recent RMT switches [6] and SmartNICs enable programmability along the packet data plane. Researchers have proposed the use of in-network computation to offload compute operations from endhosts into these network devices. For example, IncBricks [41] is an in-network caching fabric with some basic computing primitives. NetCache [25] 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 [55] conducts data aggregation along the network path on programmable switches.

RDMA-based datacenter applications. Recent years have seen 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 [16, 27, 45], DSM (distributed shared memory) system [16, 46], database and transactional system [12, 17, 29, 63]. Generally, RDMA provides fast data access capabilities but 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 SmartNICs. It does, however, borrow some techniques approaches from related RDMA projects (e.g., lazy updates for the send/receive rings in FaRM [16]).

Microsecond-scale scheduler. Researchers have proposed schedulers to reduce the tail latency of μs-scale tasks. ZygOS [54] builds a work-conserving scheduler that applies the d-FCFS queueing discipline and enables low-overhead task stealing. Shinjuku [26] addresses a similar problem (i.e., handling tasks with variable service times) as the iPipe scheduler. It provides a fast preemptive scheduling mechanism by mapping the local APIC address into the guest physical address space. However, triggering an interrupt is not only a costly operation on the SmartNIC, it is also disabled on SmartNICs that are firmware-based (such as LiquidIOII ones). Instead, we explore a hybrid scheduling discipline that runs heavy-weight actors on DRR cores and executes light-weight actors on FCFS cores.

7 Conclusion

This paper makes a case for offloading distributed applications onto a Multicore SoC SmartNICs. We conduct a detailed performance characterization on different commodity Multicore SoC SmartNICs and build the iPipe framework based on experimental observations. We then develop three applications using iPipe and prototype them on these SmartNICs. Our evaluations show that by offloading computation to a SmartNIC, one can achieve considerable host CPU and latency savings. This work does not raise any ethical issues.

8 Acknowledgments

This work is supported in part by NSF grants CNS-1616774, CNS-1714508, and 1751231. We would like to thank the anonymous reviewers and our shepherd, Bruce Maggs, for their comments and feedback.

References

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.*

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM.*

[3] Venkat Anantharam. 1999. Scheduling strategies and long-range dependence. *Queueing systems* 33, 1-3 (1999), 73–89.

[4] Apache. 2017. The Apache Cassandra Database. http://cassandra.apache.org. (2017).

[5] ARM. 2019. ARM Cortex-A72 Multi-core Processor. https://developer.arm.com/products/processors/cortex-a/cortex-a72. (2019).

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. 99–110.

[7] Broadcom. 2019. Broadcom Stingray SmartNIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225. (2019).

[8] Broadcom. 2019. The TruFlow Flow processing engine. https://www.broadcom.com/applications/data-center/cloud-scale-networking. (2019).

[9] Cavium. 2017. Cavium OCTEON Multi-core Processor. http://www.cavium.com/octeon-mips64.html. (2017).

[10] Cavium. 2017. OCTEON Development Kits. http://www.cavium.com/octeon_software_develop_kit.html. (2017).

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation.*

[12] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems.*

[13] Cisco. 2015. 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. 2016. 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] Russ Cox. 2019. Implementing Regular Expressions. https://swtch.com/~rsc/regexp/. (2019).

[16] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation.*

[17] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles.*

[18] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation.*

[19] Daniel Firestone. 2017. Hardware-Accelerated Networks at Scale in the Cloud. https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf. (2017).

[20] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation.*

[21] Alex Goldhammer and John Ayer Jr. 2008. Understanding performance of PCI express systems. *Xilinx WP350, Sept* 4 (2008).

[22] Troy D. Hanson. 2017. Uthash Hashtable. https://troydhanson.github.io/uthash/. (2017).

[23] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73).*

[24] Huawei. 2018. Huawei IN550 SmartNIC. https://e.huawei.com/us/news/it/201810171443. (2018).

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

[26] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).*

[27] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. 295–306.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference.*

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. 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).*

[30] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems.*

[31] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.*

[32] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems.*

[33] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[34] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[36] LevelDB. 2017. LevelDB Key-Value Store. http://leveldb.org. (2017).

[37] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles.*

[38] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference.*

[39] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation.*

[40] Jianxiao Liu, Zonglin Tian, Panbiao Liu, Jiawei Jiang, and Zhao Li. 2016. An approach of semantic web service classification based on Naive Bayes. In *Services Computing (SCC), 2016 IEEE International Conference on.*

[41] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. 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.*

[42] Marvell. 2018. Marvell LiquidIO SmartNICs. https://www.marvell.com/documents/08icqisgkbtn6kstgzh4/. (2018).

[43] Mellanox. 2018. Mellanox BuleField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic. (2018).

[44] Mellanox. 2019. Accelerated Switch and Packet Processing. http://www.mellanox.com/page/asap2?mtag=asap2. (2019).

[45] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store.. In *USENIX Annual Technical Conference.*

[46] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory.. In *USENIX Annual Technical Conference.*

[47] Netronome. 2018. Netronome Agilio SmartNIC. https://www.netronome.com/products/agilio-cx/. (2018).

[48] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.*

[49] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation.*

[50] OFED. 2019. Infiniband Verbs Performance Tests. https://github.com/linux-rdma/perftest. (2019).

[51] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).*

[52] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*.

[53] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*.

[54] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[55] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*.

[56] Linus Schrage. 1968. Letter to the editor—a proof of the optimality of the shortest remaining processing time discipline. *Operations Research* 16, 3 (1968), 687–690.

[57] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.

[58] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* 4, 3 (1996), 375–385.

[59] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-typed actors for java. In *European Conference on Object-Oriented Programming*.

[60] James W Stamos and Flaviu Cristian. 1993. Coordinator log transaction execution protocol. *Distributed and Parallel Databases* 1, 4 (1993), 383–408.

[61] Alexander L Stolyar and Kavita Ramanan. 2001. Largest weighted delay first scheduling: Large deviations and optimality. *Annals of Applied Probability* (2001), 1–48.

[62] SmartNIC Vendors. 2019. Marvell, Private communications. unpublished. (2019).

[63] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*.

[64] Adam Wierman and Bert Zwart. 2012. Is tail-optimal scheduling possible? *Operations research* 60, 5 (2012), 1249–1257.

[65] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*.

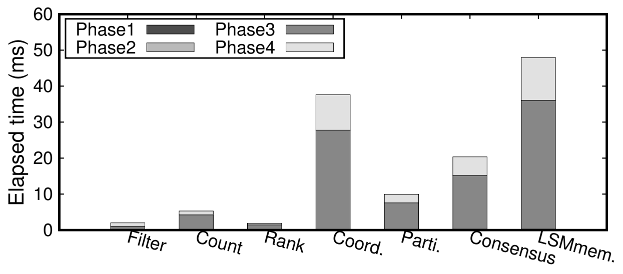


Figure 18: Migration elapsed time breakdown of 8 actors from three applications evaluated with 10GbE CN2350 cards.

Appendix A SmartNIC computing unit characterization

Appendices are supporting material that has not been peer reviewed. Table 3 summarizes the microarchietcture results for LiquidIOII CN2350 multicore processor and accelerators.

Appendix B More details in the iPipe framework

This section describes more details of the iPipe framework that is not included in the main paper.

B.1 iPipe runtime APIs

Table 4 presents the major APIs. Specifically, the actor management APIs are used by our runtime. We provide five calls for managing DMOs. 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 *dmo_free*, iPipe frees the space allocated for the object and deletes the entry from the object table. *dmo_memset, dmo_memcpy, dmo_memmove* resemble memset/memcpy/memmove APIs in glibc, except that it uses the object ID instead of a pointer.

For the networking stack, iPipe takes advantage of packet processing accelerators (if the SmartNIC has) to build a shim customized networking stack for the SmartNIC. This stack performs simple Layer2/Layer3 protocol processing, such as packet encapsulating/decapsulation, checksum verification, etc. When building a packet, it uses the DMA scatter-gather technique to combine the header and payload if they are not colocated. This helps improve the bandwidth utilization, as shown in our characterization (Section 2.2.5).

B.2 iPipe actor scheduling algorithm

Algorithms 1 and 2 show the details of our iPipe hybrid scheduler.

B.3 iPipe actor migration evaluation

When migrating an actor to the host, as shown in Figure 12, 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 DMO works similarly, except that it uses the glibc memory allocator.

We estimate the migration cost (SmartNIC-pushed) by breaking down the time elapsed of four phases 3.2.5. We choose 8 actors from three applications. our experiments are conducted under 90% networking load and we force the actor migration after the warm up (5s). Figure 18 presents our results. First, phase 3 dominates the migration cost (i.e., 67.8% on average of 8 actors) since it requires to move the distributed objects to the host side. For example, the LSM memtable actor has around 32MB object and consumes 35.8ms. Phase 4 ranks the second (i.e., 27.2%) as it pushes buffered requests to the host. Also, it varies based on the networking load. Phase 1 and Phase 2 are two lightweight parts because they only introduce the iPipe runtime locking/unlocking and state manipulation overheads.

Algorithm 1 iPipe FCFS scheduler algorithm

```
1: $wqe$ : contains packet data and metadata
2: $DRR\_queue$ : the runnable queue for the DRR scheduler
3: procedure FCFS_SCHED                          ▷ on each FCFS core
4:     while true do
5:         $wqe = iPipe\_nstack\_recv()$
6:         $actor = iPipe\_dispatcher(wqe)$
7:         if actor.is_DRR then
8:             $actor.mailbox\_push(wqe)$
9:             Continue
10:        end if
11:        $actor.actor\_exe(wqe)$
12:        $actor.bookeeping()$              ▷ Update execution statistics
13:        if T_tail > Tail_thresh then                  ▷ Downgrade
14:            $actor.is\_DRR = 1$
15:            $DRR\_queue.push(actor)$
16:        end if
17:        if core_id is 0 then                    ▷ Management core
18:            if T_mean > Mean_thresh then                ▷ Migration
19:                $iPipe\_actor\_migrate(actor\_chosen)$
20:            end if
21:            if T_mean < (1-$\alpha$)Mean_thresh then       ▷ Migration
22:                $iPipe\_actor\_pull()$
23:            end if
24:        end if
25:    end while
26: end procedure
```

Applications	Computation	DS	Exe. Lat.(us)	IPC	MPKI	Accelerator	IPC	MPKI	Exe. Lat.(us)		
									bsz=1	bsz=8	bsz=32
Baseline (echo)	N/A	N/A	1.87	1.4	0.6	**CRC**	1.2	2.8	2.6	0.7	0.3
Flow monitor [57]	Count-min sketch	2-D array	3.2	1.4	0.8	**MD5**	0.7	2.6	5.0	3.1	3.0
KV cache [37]	key/value Rr/Wr/Del	Hashtable	3.7	1.2	0.9	**SHA-1**	0.9	2.6	3.5	1.2	0.9
Top ranker [53]	Quick sort	1-D array	34.0	1.7	0.1	**3DES**	0.8	0.9	3.4	1.3	1.1
Rate limiter [38]	Leaky bucket	FIFO	8.2	0.7	4.4	**AES**	1.1	0.9	2.7	1.0	0.8
Firewall [38]	Wildcard match	TCAM	3.7	1.3	1.6	**KASUMI**	1.0	0.9	2.7	1.1	0.9
Router [32]	LPM lookup	Trie	2.2	1.3	0.6	**SMS4**	0.8	0.9	3.5	1.4	1.2
Load balancer [18]	Maglev LB	Permut. table	2.0	1.3	1.3	**SNOW 3G**	1.4	0.5	2.3	0.9	0.8
Packet scheduler [2]	pFabric scheduler	BST tree	12.6	0.5	4.9	**FAU**	1.4	0.6	1.9	1.4	1.0
Flow classifier [40]	Naive Bayes	2-D array	71.0	0.5	15.2	**ZIP**	1.0	0.2	190.9	N/A	N/A
Packet replication [31]	Chain replication	Linklist	1.9	1.4	0.6	**DFA**	1.3	0.2	9.2	7.5	7.3

Table 3: Performance comparison among generic offloaded applications and accelerators for the 10GbE LiquidIOII CN2350. Request size is 1KB for all cases. We report both per-request execution time as well as microarchitectural counters. DS=Data structure. IPC=Instruction per cycle. MPKI=L2 cache misses per kilo-instructions. bsz=Batch size. DFA=Deterministic Finite Automation.

	API	Explanation
Actor	actor_create (*)	create an actor
	actor_register (*)	register an actor into the runtime
	actor_init (*)	initialize an actor private state
	actor_delete (*)	delete the actor from the runtime
	actor_migrate (*)	migrate an actor to host
DMO	dmo_malloc	allocate a dmo obj.
	dmo_free	free a dmo obj.
	dmo_mmset	set space in a dmo with value.
	dmo_mmcpy	copy data from a dmo to a dmo.
	dmo_mmmove	move data from a dmo to a dmo.
	dmo_migrate	migrate a dmo to the other side.
MSG	msg_init	initialize 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 4: iPipe major APIs. There are four categories: actor management (Actor), distributed memory object (DMO), message passing (MSG), and networking stack (Nstack). The Nstack has additional methods for packet manipulation. APIs with * are mainly used by the runtime as opposed to actor code.

Algorithm 2 iPipe DRR scheduler algorithm

```
1:  procedure DRR_SCHED                                    ▷ On each DRR core
2:      while true do
3:          for actor in all DRR_queue do
4:              if actor.mailbox is not empty then
5:                  actor.update_deficit_val()
6:                  if actor.deficit > actor.exe_lat then
7:                      wqe = actor.mailbox_pop()
8:                      actor.actor_exe(wqe)
9:                      actor.bookeeping()
10:                     if T_tail < (1-$\alpha$)Tail_thresh then     ▷ Upgrade
11:                         actor.is_DRR = 0
12:                         DRR_queue.remove(actor)
13:                     end if
14:                 end if
15:                 if actor.mailbox is empty then
16:                     actor.deficit = 0
17:                 end if
18:                 if actor.mailbox.len > Q_thresh then         ▷ Migration
19:                     iPipe_actor_migrate(actor)
20:                 end if
21:             end if
22:         end for
23:     end while
24: end procedure
```