iPipe: A Framework for Building Distributed Applications on Multicore SoC SmartNICs

Ming Liu
Univ. of Washington

Tianyi Cui
Univ. of Washington

Henry Schuh
Univ. of Washington

Arvind Krishnamurthy
Univ. of Washington

Simon Peter
Univ. of Texas, 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 in an efficient way to maximize the offloading benefits, especially for distributed applications. Towards this end, we characterize four such commodity SmartNICs and summarize the offloading performance implications from four angles: traffic control, computing units, onboard DRAM, and host communication.

Based on our characterization, we build iPipe, an actor-based framework for developing distributed applications that is able to use a SmartNIC's computing power efficiently. The central piece of iPipe is a hybrid scheduler combining FCFS and DRR-based processor sharing that 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, along with 23.0/28.0 μs latency savings.

1 Introduction

Multicore SoC SmartNICs have emerged for the datacenter, aiming to mitigate the gap between increasing network bandwidth and stagnating CPU computing power [15, 16, 22]. In the last two years, major network hardware vendors have released different SmartNIC products, such as Mellanox's Bluefield [46], Broadcom's Stingray [10], Marvell(Cavium)'s LiquidIO [45], Huawei's IN5500 [27], and Netronome's Agilio [51]. They not only target acceleration of traditional networking/storage protocol processing (e.g., OVS/RoCE/iWARP/TCP offloading, traffic monitoring, firewall, etc.), but also bring a new computing substrate into the data center to expand the server computing capacity at a lower cost: SmartNICs usually enclose simple microarchitecture computing cores that are cheap and cost-effective.

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

There have been some recent research efforts that offload networking functions onto FPGA-based SmartNICs (e.g., ClickNP [41], AzureCloud [23]). They take a conventional domain-specific acceleration approach that consolidates as much of the 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 categorize the Multicore SoC SmartNIC into four architectural components – traffic control, computing units, onboard memory, host communication – and use microbenchmarks to understand their performance implications. The experiments identify the resource constraints that we have to be cognizant of, illustrate the utility of a SmartNIC's hardware acceleration units, and provide guidance on how to efficiently utilize the SmartNIC resources.

We design and implement the iPipe framework based on our characterization observations. iPipe introduces an actor programming model for distributed application development.

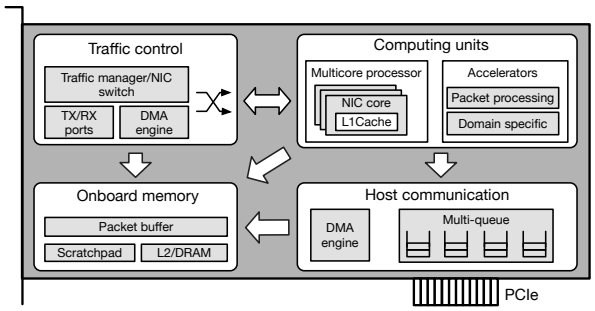


Figure 1: Architectural block diagram for a Multicore SoC SmartNIC.

Each actor has 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 the use of FCFS and DRR-based processor sharing, in order to tolerate tasks with variable execution costs and maximize the SmartNIC computing resource utilization. iPipe allows multiple actors to coexist safely on the SmartNIC, protecting against actor state corruption and denial-of-service attacks.

We prototype iPipe and build three applications (i.e., a real-time data analytics engine, a distributed transaction processing system, and a replicated key-value store) using commodity 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 three 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 the necessary background and presents our 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 is comprised of four major parts (shown in Figure 1): (1) computing units, which includes a general purpose ARM/MIPS multicore processor, along with packet processing (e.g., deep packet inspection, packet buffer management) and domain specific accelerators (e.g., encryption/decryption, hashing, pattern matching, compression); (2) onboard memory, enclosing fast scratchpad and slower L2/DRAM; (3) traffic control module that transfers packets between TX/RX ports and the packet buffer and an internal traffic manager or NIC switch that provides packets to NIC cores with low synchronization overheads; (4) host communication DMA engines.

Table 1 lists the HW/SW specifications of four COTS Multicore SoC SmartNICs evaluated in this paper. They present different design tradeoffs with regards to performance, programmability, and flexibility. The first two LiquidIOII SmartNICs enclose an OCTEON [11] 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 highend ARM Cortex-A72 [7] processor and hosts a full-fledged operating system. It offers lower barrier for application development, and one can use traditional Linux/DPDK/RDMA stacks to communicate with local and external traffic. The Bluefield card even has NVDIMM support for fault tolerance. The link speed for today's Multicore SoC SmartNIC is 10/25 GbE and 100GbE ones will be available for both enterprises and data centers in the next few years. Generally, a SmartNIC is a bit more expensive than a traditional dumb NIC. For example, a 10/25GbE SmartNIC costs 100~200$ and 300~1000$ [8], respectively, while a data center 10/25GbE NIC costs 100~200$.

SmartNICs have packet communication paths similar to that of normal NICs, except that the computing units can touch and modify the packet contents as they traverse the NIC. For the transmit path (where a host server sends out traffic via a SmartNIC), 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 (which is located in NIC memory). The traffic manager generates a work item (including the address of the packet) and delivers it to the NIC core. After some processing, the NIC core sends the packet out to the TX port directly via the DMA engine. The receiving side (where a host server receives traffic from the SmartNIC) is similar but in the reverse order. Some SmartNICs (e.g., Bluefield and Stingray) don't integrate a hardware packet buffer management engine and will use a software driver instead. They also replaces the traffic manager with a NIC programmable switch for installation of flow forwarding and steering rules so that host-only traffic can bypass the NIC processor (e.g, *ASAP*^{2} [47]).

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 boxes as host servers for both the client and server and an Arista

SmartNIC model	Vendor	Processor	BW	L1	L2	DRAM	Deployed SW	Nstack	To/From host
LiquidIOII CN2350 [45]	Marvell	cnMIPS 12 core, 1.2GHz	2× 10GbE	32KB	4MB	4GB	Firmware	Raw packet	Native DMA
LiquidIOII CN2360 [45]	Marvell	cnMIPS 16 core, 1.5GHz	2× 25GbE	32KB	4MB	4GB	Firmware	Raw packet	Native DMA
BlueField 1M332A [46]	Mellanox	ARM A72 8 core, 0.8GHz	2× 25GbE	32K	1MB	16GB	Full OS	Linux/DPDK/RDMA	RDMA
Stingray PS225 [10]	Broadcom	ARM A72 8 core, 3.0GHz	2× 25GbE	32K	16MB	8GB	Full OS	Linux/DPDK/RDMA	RDMA

Table 1: Specifications of 4 studied COTS (commercial off-the-shelf) Multicore SoC SmartNICs. BW = bandwidth. Nstack = networking stack.

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 slot at the server side. 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 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 a desired packet interval. We report end-to-end performance metrics (e.g., latency/throughput), as well as 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 two tasks: packet forwarding (through TX/RX ports) and packet feeding to the NIC computing cores. Here, we use an ECHO server that entirely runs on a SmartNIC to understand: (1) how many NIC cores are sufficient to saturate the link speed for different packet sizes and how much computing capacity is left for other "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 NICs 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 takes 3/2/1/1 cores. Stingray uses fewer cores due to its much higher core frequency (3.0GHz v.s. 1.20GHz). This indicates that packet forwarding is not free, which is the default execution tax of a SmartNIC. Figure 4 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 don't 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. Such results can also be observed from the Stingray card. This means that the hardware traffic manager is effective at providing a shared queue abstraction with little synchronization overhead for the packet buffer management.

Design implications: ***I1:*** Not only is packet forwarding not free, but 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 and adaptively make the offloading decisions. ***I2:*** Hardware support reduces synchronization overheads and enables scheduling paradigms that involve multiple workers pulling 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.1). 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[1] or high MPKI 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 that are less likely to benefit from the complex microarchitecture on the host, and 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 the execution completion and also incurs cache misses (i.e., higher MPKI) in feeding data to the accelerator. Batching can amortize invocation costs but result in tying up the NIC core for longer periods. Other SmartNICs (e.g., BlueField and Stingray) display similar characteristics.

SmartNICs also usually enclose special accelerators for packet processing. Take the LiquidIOII ones (CN2350/CN2360) for example. It has packet input (PKI) and packet output units (PKO) for moving data between MAC and packet buffer and a hardware managed packet buffer along with fast packet indexing. When compared with two host-side kernel-bypass networking stacks (DPDK/RDMA), even with the polling

[1]Note that the cnMIPS OCTEON [11] is a 2-way processor and the ideal IPC is 2.

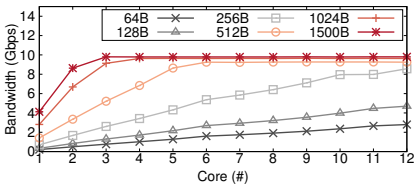


Figure 2: SmartNIC bandwidth varying the number of NIC cores for the 10GbE LiquidIOII CN2350.

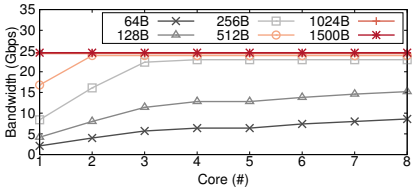


Figure 3: SmartNIC bandwidth varying the number of NIC cores for the 25GbE Stingray PS225.

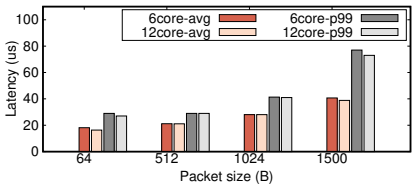


Figure 4: Average/p99 latency when achieving the max throughput on the 10GbE LiquidIOII CN2350.

	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 1 cacheline to different memory hierarchies on 4 SmartNICs and the Intel server. The cacheline for LiquidIOII ones is 128B while the rest is 64B. The performance of LiquidIOII CN2350 and CN2360 is similar.

model, for SEND, it shows 4.6X/4.2X speedups on average across all cases (shown in Figure 5), respectively.

Design implications: ***I3:*** The offloading framework should be able to handle tasks with a wide range of execution latencies and simultaneously ensure that the NIC's packet forwarding is not adversely impacted. ***I4:*** One should take advantage of the available accelerators on the SmartNIC and perform batched execution for domain specific ones 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. A SmartNIC (like LiquidIOII) usually has hardware based packet buffer management. Some SmartNICs (like Bluefield and Stingray) don't have a dedicated packet buffer region. (3) L2 cache, which is 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 (as 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 4 SmartNICs and compare it with the host server. The results in Table 2 illustrate 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 a performance comparable to the host.

Design implications: ***I5:*** When the application working set exceeds the L2 cache size of the 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 one used by all of our SmartNICs). Its performance is usually impacted by many runtime factors. With respect to latency, DMA engine queueing delay, PCIe request size and its response ordering, PCIe completion word delivery, and host DRAM access costs will all slow down PCIe packet delivery [24, 32, 52]. With respect to throughput, PCIe is limited by transaction layer packet (TLP) overheads (i.e., 20-28 bytes for header and addressing), the maximum number of credits used for flow control, the queue size in DMA engines, and PCIe tags used for identifying unique DMA reads.

Generally, a DMA engine provides two kinds of primitives: blocking accesses, which wait for the DMA completion word from the engine, and non-blocking ones, which allow the processing core to continue executing after sending the DMA commands into the command queue. Figures 6 and 7 show our performance characterizations of the 10GbE LiquidIO CN2350. Non-blocking operations insert a DMA instruction word into the queue and don't wait for completion. Hence, its read/write latency and throughput are independent of packet size, which outperform the blocking primitives. 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. This indicates that one should take advantage of the DMA scatter and gather technique.

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, which resembles the DMA blocking operations. We observe similar results as the LiquidIOII ones, but it requires much larger payloads to amortize the verbs software processing cost.

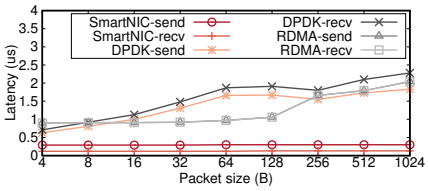


Figure 5: Send/recv latency on the 10GbE LiquidIOII CN2350, compared with RDMA/DPDK ones measured on the host.

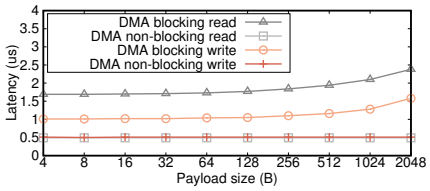


Figure 6: Per-core blocking/non-blocking DMA read/write latency when increasing payload size.

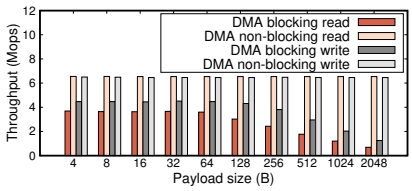


Figure 7: Per-core blocking/non-blocking DMA read/write throughput when increasing payload size.

SmartNICs can assist in intelligent packet steering. Multiqueue is a de-facto standard for today's modern NICs. By providing multiple transmit and receive queues, different host CPU cores can process packets simultaneously without coordination, achieving higher throughput. Normal NICs usually apply a hashing function to map network flows into different queues, referred as RSS [48] (receiver side scaling) or Intel flow director [28]. SmartNICs can push this benefit even further – by looking into the application header, one can perform application level dispatching. The performance benefits of this feature over host-side dispatching can be significant (as shown in Appendix C.1).

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). In addition, SmartNICs can assist host processing by performing intelligent application-layer dispatching.

3 iPipe framework

This section describes the design and implementation of our iPipe framework. We use the insights 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, plenty of accelerators, etc.), but one should use them in an efficient way. 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, that 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, 26, 58] for application development. iPipe uses the actor-based model, instead of say dataflow or thread-based models for the following reasons. First, unlike a thread-based model, actors interact with each other not through shared memory but with explicit messages. Given the communication latencies that we observe between the NIC and the host, explicit messaging is more appropriate in our setting. Second, the actor model is able to support computing heterogeneity and hardware parallelism automatically. While dataflow models also provide such support, the actor-based model allows for non-deterministic and irregular communication patterns that arise in complex distributed applications. Finally, actors have well-defined associated states and can be migrated between the NIC and the host dynamically. This allows us to have dynamic control over the use of SmartNIC computing capabilities as we adapt to traffic characteristics (which is necessary given our characterization experiments).

An actor is a computation agent that performs two kinds of operations based on incoming type of messages: (1) trigger its execution handlers and manipulate its private state; (2) interact with other actors by sending messages asynchronously. Actors don't share memory. 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 address for all actors. The structure outlined above represents a streamlined and lightweight actor design.

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) for actor development. 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, as well as support for 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 them below.

3.2 iPipe Actor Scheduler

iPipe schedules the actor execution among SmartNIC/host cores. The scheduler allocates actor execution tasks to computing cores and specifies a custom scheduling discipline for different tasks. 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 compromising the NIC's primary task of conveying traffic. Recall that 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 would suffer from temporary load imbalances, thus leading to worse tail latencies. Fortunately, hardware traffic managers on SmartNICs provide support for a shared queue abstraction with low synchronization overhead (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 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 for 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 [55]. However, in our setting, we also care about the tail response time; even if the application can tolerate it, a high response latency in our setting means that the NIC isn't performing its basic duty of forwarding traffic in a timely manner. If we were to consider minimizing the tail response time, then First Come

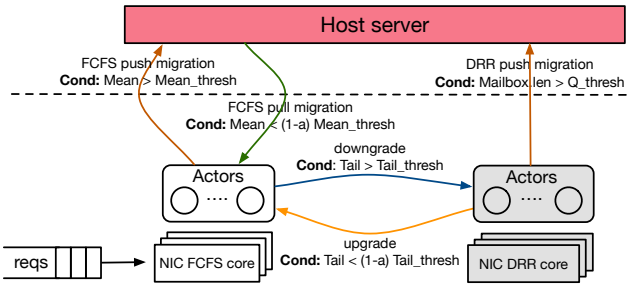


Figure 8: An overview of iPipe scheduler on the SmartNIC. Cond is the operation triggered condition.

First Served (FCFS) is considered optimal when task size distribution has low variance [59] but has been shown to perform poorly when the task size distribution has high dispersion or is heavy-tailed [4]. In contrast, Processor Sharing is considered optimal for high variance distributions [61].

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 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.

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 greater variance in service time to a DRR scheduler. The scheduler uses DRR for high variance tasks as DRR is an efficient approximation of Processor Sharing in a non-preemptible setting [57]. Further, the scheduler places as much computation as possible on the SmartNIC and migrates actors when the NIC cannot promptly handle incoming bandwidth. To assist in 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 greater 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 indicates a queue build up at the SmartNIC and one should migrate the actor that contributes the most to the overloading to the host processor (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 issues a pull request to the host server to migrate the actor that will incur the least load back to the SmartNIC. We use a dedicated core of the FCFS group (core 0) 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 among all cores' CPU usage. Finally, we use measurements from our characterization study to set the thresholds *mean_thresh* and *tail_thresh*. We consider the MTU packet

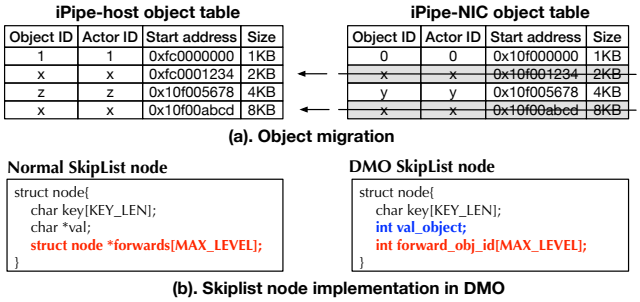


Figure 9: iPipe distributed memory objects.

size at which the SmartNIC is able to 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). This means 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 one core for DRR, and the scheduler will migrate a core to DRR. A similar condition is used 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 before, when there is persistent queueing and the mean response time is above a threshold, the scheduler will move an actor to the host side. We pick the actor with the highest load (i.e., average execution latency scaled by frequency of invocation) for migration. We perform a cold migration mechanism in four phases as described in details in the Appendix B.3.

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 outline 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 and others

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 9-a) on both sides and utilizes the local memory manager to allocate/deallocate 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.

When using DMOs to design a data structure, one has to use the object ID for indexing instead of pointers. This 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 the skiplist based memtable via DMO. As shown in Figure 9-b, a traditional skiplist node includes a key string, a value string, and a set of forwarding pointers. With DMO, the key field is the same. Value and forwarding pointers are replaced by object IDs. When traversing, one will use the object ID to get the start address of the object, cast the type, and then read/write its contents.

Scratchpad. Our characterization experiments 2.2.4 have shown that the scratchpad memory provide the fastest performance but has very limited resources. Instead of exposing this to applications and managing them, we decide to keep this memory resource internally and use it for storing the iPipe bookkeeping information.

3.4 Security Isolation

iPipe allows multiple actors to concurrently execute on a SmartNIC. There are two attacks that iPipe should protect against: (1) actor state corruption, where a malicious actor manipulates other actors' states; (2) denial-of-service, where an actor hangs on the SmartNIC core and violates the service availability of other actors. We primarily describe how to protect against these attacks on the Cavium LiquidIOII, as one can apply similar techniques to other SmartNICs.

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. In this case, we use a physical partition approach. During the initialization phase, iPipe creates large equal size chunks of memory regions for each registered actor (where its size is $\frac{Mem_{total}-Mem_{fimrware}-Mem_{runtime}}{Num_{actor}}$). iPipe runtime maintains the mapping between actor ID, its base address, and size. During execution, an actor can only allocate/reclaim/access its objects within its region. 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., executing an infinite loop), violating actor availability. We apply a timeout mechanism to address this issue. LiquidIOII CN2350/CN2360 SmartNICs include a hardware timer with 16 timer rings. We give each core a dedicated timer. When an actor is executed, 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.

3.5 Host/NIC communication

We use a message passing mechanism to communicate between 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. 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 list.

4 Applications built with iPipe

We implement three distributed applications using 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 key system components: a *consensus protocol*, and a *key-value data store*. We use the traditional Multi-Paxos algorithm [37] to achieve consensus among multiple replicas. Each replica maintains an ordered log for every Paxos instance. There is a distinguished leader that receives client requests and performs consensus coordination using Paxos prepare/accept/learning messages. In the common case, consensus for a log instance can be achieved with a single round of accept messages, and the consensus value can be disseminated using an additional round (learning phase). Each node of a replicated state machine can then execute the sequence of commands in the ordered log to implement the desired replicated service. When the leader fails, replicas will run a two-phase Paxos leader election (which determines the next

leader), choose the next available log instance, and learn accepted value from other replicas if its own log has gaps. Typically, the Multi-Paxos protocol can be expressed as a sequence of messages that are generated and processed based on the state of the RSM log.

For the key-value store, we take the log-structure merge tree (LSM) that is widely used for many KV systems (such as Google's Bigtable [13], LevelDB [39], Cassandra [5]). An LSM tree accumulates recent updates in memory and serves reads of recently updates values from 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., SkipList) and *SSTables*, collections of data items sorted by their keys and organized into a series of levels. Each level has a size limit on its SSTables, and this limit grows at an exponential rate with the level number. 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 one 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 stationary on the host because they have to interact with persistent storage.

Distributed Transactions. We build a distributed transactions system that uses optimistic concurrency control and two-phase commit for distributed atomic commit, following the design used by other systems [31, 62]. Note that we choose to not add a replication layer as we try to eliminate the application function overlap with our replicated key-value store. The application includes a coordinator and participants that run a transaction protocol. 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 and then sends a reply to the client with the result; 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.

In iPipe, we implement the coordinator and participant as actors running on the NIC. The key storage abstractions required to implement the protocol are the coordinator log [18] and the data store, which we realize using a traditional extensible hashtable [25]. Both of these are realized using distributed shared objects. We also cache responses from outstanding transactions. 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 [33] and extended its functionality. All data tuples are passed through three workers: *filter*, *counter*, and *ranker*. The filter applies a pattern matching module [17] 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 the software managed cache for statistics. Ranker is implemented using a distributed shared object, and we consolidate all top-n data tuples into one object. 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)
- When compared with another SmartNIC programming system (i.e., Floem [54]), what are the design trade-offs in terms of performance and programmability? (§5.5)
- Can we use iPipe to build other applications (e.g., network functions)? How does it perform? (§5.6)

5.1 Experimental methodology

We use the same testbed as our characterization experiments 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 [12]. On the host side, we use pthreads for iPipe execution and allocate 1GB pinned hugepages for the message ring. Each runtime thread periodically polls requests from the channel and performs actor execution. It transits to idle C-states when there is no more work. The iPipe runtime spreads across the NIC firmware and host system with 10683 LOCs and 4497 LOCs, respectively. To show the effectiveness of the actor scheduler, we also present results for the Stingray card.

Programmers use the C langugage to build applications (which are compiled with SmartNIC/host GNU tool chains). Our three workloads, real-time analytics (RTA), distributed transactions (DT), replicated key-value store (RKV), built with iPipe have 1583 LOCs, 2225 LOCs, and 2133 LOCs, respectively, and we compare 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 request based on the Twitter tweets [38]. The number of data tuples in each request vary based on the packet size. For DT, each request is a multi-key read-write transaction including 2 reads and 1 writes (used in previous work [31]). For RKV, we generate the <key,value> pair in each packet, with the following characteristics: 16B key, 95% read/5%write, zipf distribution (skewness=0.99), and 1 million keys (used in previous work[42, 53]). 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 normal Intel NICs in the case of DPDK. The RTA application runs a 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 10 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.

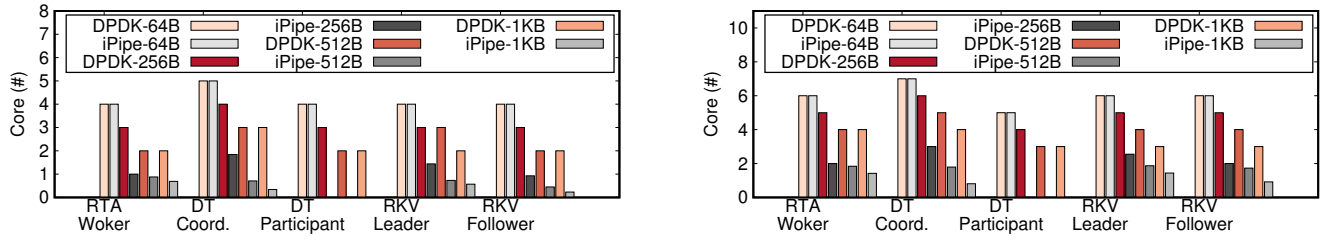
5.3 Latency versus Throughput

We next examine the latency reduction and per-core throughput increase provided by iPipe and find that SmartNIC offloading provides considerable benefits. Figures 11 and 12 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 fraction 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 use a fewer number of host CPU cores. 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 both the coordinator and the participants mainly run on the SmartNIC processor and 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 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

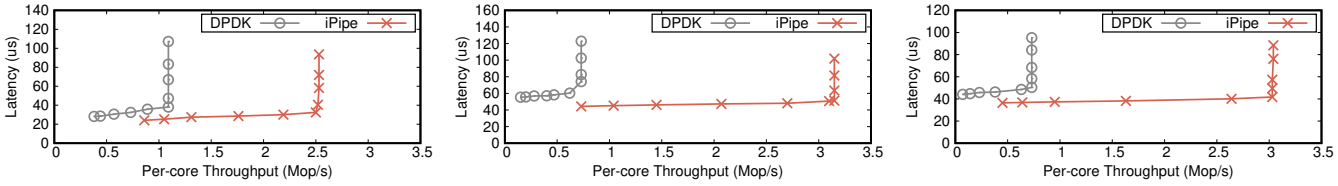
We evaluate the effectiveness of iPipe's scheduler, comparing it with standalone FCFS and DRR schedulers under



(a) 10GbE w/ LiquidIOII CN2350.

(b) 25GbE w/ LiquidIOII CN2360.

Figure 10: Host used CPU cores compared between DPDK and iPipe on three different applications varying packet sizes for a 10GbE/25GbE network.

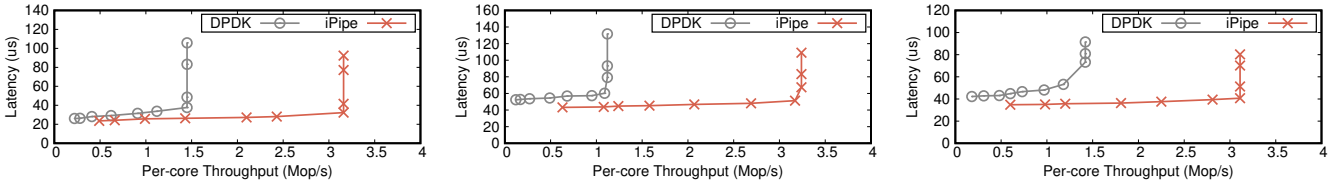


(a) RTA.

(b) DT.

(c) RKV.

Figure 11: Latency versus per-core throughput for three applications under 10GbE, compared between DPDK and iPipe cases. Packet size is 512B.

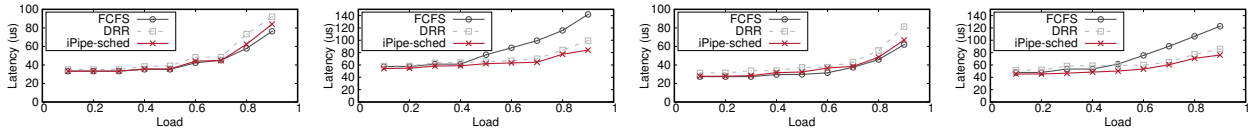


(a) RTA.

(b) DT.

(c) RKV.

Figure 12: Latency versus per-core throughput for three applications under 25GbE, compared between DPDK and iPipe cases. Packet size is 512B.



(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 13: P99 tail latency varies with the networking load for low and high dispersion request distribution for 10GbE LiquidIOII CN2350 and 25GbE Stingray cards.

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 actor scheduler runs as firmware hardware threads and OS pthreads, respectively. The workload generator is built using packet traces obtained from our three real world applications and issues requests assuming a Poisson process. We measure the latency from the client side. 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 13 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 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).

5.5 Comparison with Floem

Floem [54] 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 couple of programming abstractions, such as logic queue, per-packet state, etc. iPipe also has similar designs (like message rings, packet metadata). However, compared with iPipe, the key difference is that the language runtime of Floem doesn't use the SmartNIC computing power in an efficient way. First, the offloaded elements (computation) on Floem is stationary, no matter what the incoming traffic is. However, we have shown that, when the packet size is small and networking load is high (Section 2.2.2), such Multicore SoC SmartNICs have no room for application computation. In iPipe, 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 (even though they can be expressed) are performed on the host side. In iPipe, we have shown that complex operations can also be offloaded, and our 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 side and uses all NIC cores for packet forwarding, while Floem still uses the NIC-side for offloading. In sum, we believe iPipe can be an efficient backend for Floem.

5.6 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. However, a FPGA based solution achieves 1.23~1.6μs. For the IPsec gateway, 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. Such 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, one can achieve comparable performance as FPGA based ones for network functions.

6 Related work

SmartNIC acceleration. In addition to Floem [54], ClickNP [41] is another framework using FPGA-based SmartNICs for network functions. It uses the Click [36] dataflow programming model and statically allocates a regular dataflow graph model during configuration, whereas iPipe is able to 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. KV-Direct [40] 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. HotCocoa [6] proposes a set of hardware abstractions to offload the entire congestion control algorithm to a SmartNIC.

In-network computations. Recent RMT switches [9] and SmartNICs enable programmability along the packet data plane. Researchers have proposed the use of in-network computation, where one can offload compute operations from endhosts into these network devices in order to reduce datacenter traffic and improve application performance. For example, IncBricks [44] is an in-network caching fabric with some basic computing primitives. NetCache [29] is another in-network caching design, which uses a packet-processing pipeline on a Barefoot Tofino switch to detect, index, store, invalidate, and serve key-value items. DAIET [3] conducts data aggregation (for MapReduce and TensorFlow) along the network path using programmable switches.

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 [19, 30, 49], DSM system [19, 50], database and transactional system [14, 20, 31, 60]. 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 [19]).

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.

References

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[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 (SIGCOMM '13)*. ACM, New York, NY, USA, 435–446. https://doi.org/10.1145/2486001.2486031

[3] Sapio Amedeo, Abdelaziz Ibrahim, Aldilaijan Abdulla, Canini Marco, and Kalnis Panos. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. (2017).

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

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

[6] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. 2017. HotCocoa: Hardware Congestion Control Abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. ACM, New York, NY, USA, 108–114. https://doi.org/10.1145/3152434.3152457

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

[8] Authors. 2019. Private conversation with SmartNIC vendors. unpublished. (2019).

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

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

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

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

[13] 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 (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data

[14] 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*. ACM, 26.

[15] 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).

[16] 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).

[17] Russ Cox. 2019. Implementing Regular Expressions. https://swtch.com/~rsc/regexp/. (2019).

[18] F Cristian et al. 1990. Coordinator Log Transaction Execution Protocol. (1990).

[19] 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*. 401–414.

[20] 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*. ACM, 54–70.

[21] 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 (NSDI 16)*. Santa Clara, CA, 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

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

[23] 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 (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

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

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

[26] 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)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245. http://dl.acm.org/citation.cfm?id=1624775.1624804

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

[28] Intel. 2019. Intel Flow Director. https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director. (2019).

[29] 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. ACM, 121–136.

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

[31] 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)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[32] Anuj Kalia Michael Kaminsky and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*. 437.

[33] 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 (ASPLOS '16)*. ACM, New York, NY, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[34] 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 (SIGCOMM '18)*. ACM, New York, NY, USA, 297–312. https://doi.org/10.1145/3230543.3230572

[35] 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 (EuroSys '15)*. ACM, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/2741948.2741969

[36] 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.

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

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

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

[40] 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*. ACM, 137–152.

[41] 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*. ACM, 1–14.

[42] 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 (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 429–444. http://dl.acm.org/citation.cfm?id=2616448.2616488

[43] 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*. IEEE, 356–362.

[44] 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*. ACM, 795–809.

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

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

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

[48] Microsoft. 2019. Receiver Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. (2019).

[49] 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*. 103–114.

[50] 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*. 291–305.

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

[52] 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 (SIGCOMM '18)*. ACM, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[53] 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 (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[54] 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 (OSDI 18)*. USENIX Association, Carlsbad, CA, 663–679. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

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

[56] 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)*. USENIX Association, Renton, WA, 1–16. https://www.usenix.org/conference/nsdi18/presentation/sharma

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

[58] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-typed actors for java. In *European Conference on Object-Oriented Programming*. Springer, 104–128.

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

[60] 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*. ACM, 87–104.

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

[62] 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 (SOSP '15)*. ACM, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

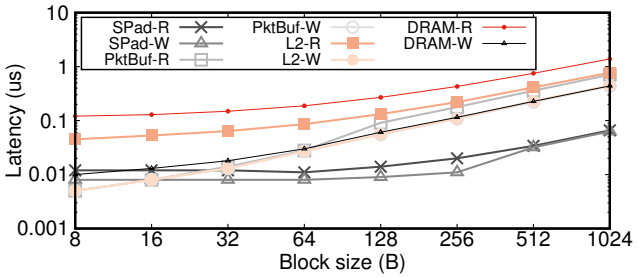


Figure 14: Read/Write latency to different onboard memory varying block size for LiquidIOII CN2350. Y is log scale.

Appendix A More characterization results

This section presents more characterization results of four Multicore SoC SmartNICs that are not included in the main paper.

A.1 Performance of microbenchmarks and accelerators

Table 3 summarize the system and microarchietcture results.

A.2 Locking primitives

Multicore SoC SmartNICs provide various locking primitives so that applications can take advantage of the parallel execution power without building complex lock-free/wait-free data structures. Usually, there are three types of locking primitives: spinlock, read-write lock, and packet-linked lock. The first two are common. The last one is a special kind of locking mechanism existed on SmartNICs, which is supported by the hardware-assisted scheduler. The NIC core enters/leaves a critical region by switching the packet metadata (i.e., tag type on LiquidIOII). When a core stays in the critical session, all other cores are unable to pull incoming traffic. This is a coarse grained global locking approach. To characterize the locking overhead, we use an Echo server and perform lock/unlock operations for per-packet computation. We use client observed throughput to measure the overhead for locking. We find that on LiquidIOII CN2350/CN2360, **1 RW read lock < 1 spinlock < 1 RW write lock < packet-linked lock < N × read-write/spinlock (where $N \geq 2$)**. One should carefully choose the right lock mechanism based on the size of critical session.

A.3 Memory read/write latency

Figure 14 presents the memory read/write latency varying block size on the LiquidIOII CN2350 SmartNIC. To measure the latency L2 cache, we firstly use firstly use MIPS prefetch instructions to load the data and perform read/write tests. Unsurprisingly, for both reads/writes, scratchpad shows the lowest latency, followed by the packet buffer, L2 cache, local memory, and host memory. Also, we find that for any memory resource, writes outperform reads. This is due ot write back configuration on the memory hierarchy.

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 to build a shim networking stack for the SmartNIC. This stack performs Layer2/Layer3 protocol processing, such as packet encapsulating/decapsulation, fragmentation, 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 procedure

We describe the 4 phase migration as follows:

- Phase 1: 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.
- Phase 2: The actor finishes the execution of its current tasks and changes to the *Ready* state. Note that, for an actor in the DRR group, it finishes executing all the requests in its mailbox.
- Phase 3: 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*.
- Phase 4: The scheduler forwards the buffered requests from the NIC to the host and rewrites their destination addresses. We will label the NIC actor as *Clean*.

When migrating an actor to the host, as shown in Figure 9, 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

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 [56]	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 [40]	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 [54]	Quick sort	1-D array	34.0	1.7	0.1	**3DES**	0.8	0.9	3.4	1.3	1.1
Rate limiter [41]	Leaky bucket	FIFO	8.2	0.7	4.4	**AES**	1.1	0.9	2.7	1.0	0.8
Firewall [41]	Wildcard match	TCAM	3.7	1.3	1.6	**KASUMI**	1.0	0.9	2.7	1.1	0.9
Router [35]	LPM lookup	Trie	2.2	1.3	0.6	**SMS4**	0.8	0.9	3.5	1.4	1.2
Load balancer [21]	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 [43]	Naive Bayes	2-D array	71.0	0.5	15.2	**ZIP**	1.0	0.2	190.9	N/A	N/A
Packet replication [34]	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.

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 15 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-α)Mean_thresh then       ▷ Migration
22:                 iPipe_actor_pull()
23:             end if
24:         end if
25:     end while
26: end procedure
```

Appendix C More evaluations

C.1 SmartNIC as a dispatcher

SmartNICs allows application-specific flow steering into multiple transmit/receive NIC queues. To demonstrate this, we use a sharded key-value store (which uses the Memtable

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-α)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
```

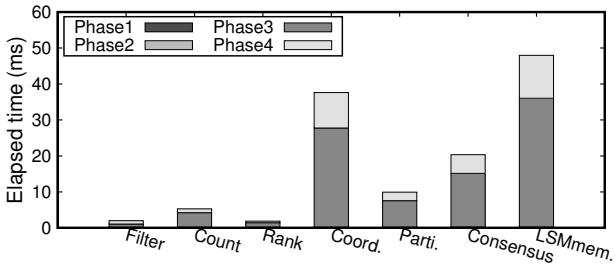


Figure 15: Migration elapsed time breakdown of 8 actors from three applications evaluated with 10GbE CN2350 cards.

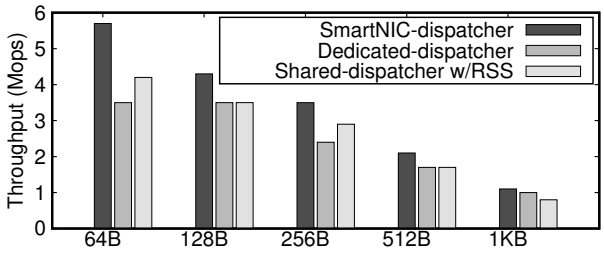


Figure 16: Throughput comparison varying packet size (on 10GbE), compared among SmartNIC, dedicated, and shared dispatchers.

implementation of the RKV) along with our workload generator and compare with three different dispatcher designs. Specifically, we configure the system with 4 host CPU cores (where each core runs one shard), 4 NIC queues, and use the 10GbE LiquidIO CN2350. We compare three scenarios: (1) *SmartNIC dispatcher*, where NIC cores obtain the shard information from the request application header and push request into the appropriate queue. (2) *dedicated dispatcher*, where we use one dedicated host core to poll incoming requests from all NIC queues, and push them to the other three cores appropriately. We divide the key space into three shards in this case. (3) *shared dispatcher*, where we run the dispatcher on all 4 cores and co-locate it with the key-value store. In this case, we also enable RSS. Note that the dedicated and shared dispatchers have a lockless FIFO command queue to buffer requests from other cores.

Figure 16 reports the measured throughput for different packet sizes. On average, the SmartNIC dispatcher outperforms the dedicated and shared dispatchers by 32.9% and 25.8%, respectively. This is because the SmartNIC steering eliminates the inter-core request transfer and command queue operation overheads.