# Mind the Gap: A Case for Informed Request Scheduling at the NIC

Jack Tigar Humphries
Stanford University and Google

Kostis Kaffes
Stanford University

David Mazières
Stanford University

Christos Kozyrakis
Stanford University and Google

## ABSTRACT

Recent research in high-throughput networked systems has established the need for centralized and preemptive request scheduling in order to achieve good hardware utilization and low tail latency for a wide variety of workloads. However, this approach is expensive to scale as it requires an increasing number of CPU cores dedicated to scheduling. Moreover, passing every request through a scheduling core introduces latency for inter-core communication and reduces the effectiveness of data preloading and caching optimizations.

In this paper, we advocate in favor of pushing request-to-core scheduling back into the NIC. Instead of the simple request distribution of receive-side-scaling (RSS) in current NICs, we suggest implementing preemptive request scheduling by passing to the NIC up-to-date information about core availability and execution status of active requests. We present a prototype implementation on a commercial SmartNIC that indeed shows performance benefits for different workload scenarios. The prototype allows us to also observe bottlenecks that largely come from artifacts of existing NIC hardware. We propose research towards addressing these limitations in order to achieve low overhead, low latency, and highly efficient request scheduling.

**ACM Reference Format:**
Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3365609.3365856


*HotNets '19, November 13–15, 2019, Princeton, NJ, USA*

ACM ISBN 978-1-4503-7020-2/19/11…$15.00
https://doi.org/10.1145/3365609.3365856

## 1 INTRODUCTION

Web search, e-commerce sites, and the various cloud services have high fanout [18]. A single user request can trigger hundreds or even thousands of requests to multiple back-end servers with different request service time characteristics, such as key-value stores (KVS) [2], databases [13, 39], and AI inference programs [16, 41]. To reduce user-experienced latency, it is necessary to minimize the tail latency of these back-end requests.

To address this need, researchers have developed specialized dataplane operating systems that shed the software bloat by bypassing the Linux kernel and avoiding inter-thread communication and synchronization [15, 22, 27, 32, 34]. These systems rely on Receive Side Scaling (RSS) [12] to randomly distribute incoming requests to polling CPU cores—i.e., they offload scheduling to NIC hardware and the CPU is not involved in scheduling decisions. This approach has allowed systems to scale to high request rates and achieve a near optimal combination of high throughput with low tail latency, but only on homogeneous workloads like KVS requests.

Workloads with highly-variable execution times such as search engines [26], function-as-a-service frameworks [21], and databases [13, 39], have proven far more challenging. Achieving low tail latency under such high-dispersion workloads requires preemption to avoid getting fast requests stuck behind slow ones [40]. Careful scheduling across cores is also needed to ensure work conservation. Recent work has shown that the benefits of using preemption and cross-core load balancing are significant enough to justify dedicating a CPU core to scheduling and dispatching [23, 31]. Unfortunately, this approach scales poorly. Each scheduling core can handle 5M requests per second, or 2.5 Gbps and 41 Gbps of Ethernet traffic if we assume 64 B and 1 KiB requests, respectively [23, 31]. Hence, scheduling can be quite expensive for the 100 Gbps and 200 Gbps networking technologies currently in deployment in datacenters. Dedicated dispatcher cores also introduce latency overheads for two events per packet (one on the scheduling core and one on a worker core) and the necessary coordination across cores. On multi-socket systems, the situation is worse if the worker chosen by the dispatcher is not on the socket whose last-level cache had the packet pre-loaded with Direct Data I/O (DDIO) [6].

If DDIO is extended to higher-level caches, a reasonable approach for chips with close to 100 cores, the latency can be high even for single socket systems.

Current systems thus have to pick between two suboptimal solutions: either schedule quickly and cheaply at the NIC, without knowledge about idle cores and how long each active request has been running on busy cores; or introduce latency and resource overheads by pushing request scheduling to a CPU core where software can make good scheduling decisions with visibility into core availability. Both approaches look increasingly unattractive as bandwidth, CPU core counts, and workload variability increase in datacenters.

We argue for an alternative approach to request scheduling, where the software running on the CPU informs the NIC about core and active request status, allowing the NIC to make fast and effective scheduling decisions as soon as requests arrive. With this approach, we don't need to expend CPU resources for scheduling. Requests can be pre-loaded into the right socket and cache with DDIO and the NIC can directly interrupt CPU cores when their running task needs to be preempted. To validate the feasibility of this scheduling approach, we designed a system that offloads the preemptive scheduling logic of the Shinjuku system [23] to a commercial SmartNIC. Specifically, *Shinjuku-Offload* implements the request queuing, request selection, core selection, and task preemption logic of the Shinjuku scheduler using the ARM cores of a Broadcom Stingray SmartNIC. We show that Shinjuku-Offload outperforms the original (purely CPU-based) Shinjuku for many scenarios, as it does not have to sacrifice CPU cores for scheduling. However, we also identify performance bottlenecks that are artifacts of the hardware architecture and software interfaces of existing NIC devices. We argue that this problem is well-suited to higher-performance hardware implementations and we propose the research steps that can lead to such systems. Essentially, we suggest that, in addition to accelerating dataplane processing such as network protocols and encryption, NICs should also accelerate control plane operations such as scheduling.

## 2 BACKGROUND AND CHALLENGES

This section reviews request scheduling approaches. Our goal is to identify common problems and to propose a new scheduling approach to eliminate them.

### 2.1 How is request scheduling performed in existing systems?

**MICA** [27] optimizes network request handling, parallel data accesses, and data structure design for small key-value store accesses. It uses Intel's Flow Director [7] to steer requests to cores based on the key they access. It can achieve throughput as high as 70 MRPS, even for skewed workloads.

**IX** [15] is a dataplane operating system that uses RSS [12] to hash packet 5-tuples and then assign packets to worker cores based on the hash. All network packet and application request processing is done on individual worker cores and runs to completion. Hardware virtualization extensions enforce isolation between the data plane and control plane. By eliminating inter-core communication and using adaptive batching, IX achieves low tail latency for high throughput.

**ZygOS** [34], similarly to IX, uses RSS to assign packets to cores, but also supports work-stealing. Cores that are idle can steal packets from task queues that belong to other cores. This design results in improved tail latency for workloads with limited dispersion.

**Shinjuku** [23] has a centralized queue maintained by a single dispatcher that assigns requests to idle cores. Requests that take too long to finish are preempted by the dispatcher using a low-overhead interrupt mechanism. Shinjuku avoids head-of-line blocking and provides high throughput and low tail latency even for highly-variable workloads.

**RPCValet** [17] is a custom architecture that makes scheduling decisions to minimize $\mu$second-scale tail latency by putting the NIC "close" to the cores. RPCValet integrates a network interface on each core and, similar to Shinjuku, maintains a centralized task queue. Due to this integration, the system has fine-grained knowledge of the load on each core, which may not be possible if the network interface is separated from the cores by a slower PCIe bus. Thus, the scheduler can account for short delays, such as TLB misses.

### 2.2 Fundamental Scheduling Problems

All the aforementioned systems suffer from issues that make them unsuitable for a large subset of existing workloads:

**1. Load Imbalance.** All RSS-based systems suffer from inherent load imbalances between cores. IX [15] and MICA [27] require a large number of concurrent connections to keep per-core queues balanced even for homogeneous tasks [23]. ZygOS [34] alleviates the imbalances by allowing work stealing between per-core queues. Shinjuku [23] and RPCValet [17] eliminate imbalance entirely by implementing a global queue in software and hardware, respectively.

**2. Lack of Preemption.** Theory tells us that low tail latency for highly-variable workloads requires processor sharing [40], a scheduling policy where all requests receive a fine-grained, fair fraction of the available processing capacity. Preemption is necessary to approximate such a policy in a real system. Request variability may be present due to diverse request types in an application, varying handling times for the same request type [19], multiple co-located applications from different latency classes, and periodic background tasks such as garbage collection. Even servers with a large number of cores are not immune to poor tail latency effects from dispersive workloads since both short and long requests may comprise large percentages of the workload (especially if co-located applications belong to different latency classes),

a workload comprised mainly of short requests could see a burst of long requests [19], and not all of the cores in the server may be used for request handling anyway.

Without preemption, short requests will get stuck behind long requests and the tail latency of the short requests will explode. Therefore, due to their lack of preemptive scheduling, ZygOS [34] and RPCValet [17], along with IX [15] and MICA [27], demonstrate high tail latency for highly-variable request service time distributions [23]. Shinjuku's use of low-overhead interrupts for preemptive scheduling allows it to co-schedule workloads ranging from the microsecond to the millisecond scale [23].

**3. Limited Scalability.** Centralized systems such as Shinjuku [23] suffer from limited scalability. The dispatcher can only scale to 5M requests (i.e., about 11 worker cores for a fixed request distribution of 1 $\mu$s). This is a small number of workers. A modern datacenter server has tens or hundreds of cores, so multiple dispatchers need to be instantiated. RSS can be used to route packets from the NIC to different dispatchers, but this can again result in load imbalance. Moreover, one physical core is dedicated to each dispatcher in the system. If the dispatcher can only scale to 11 workers, 1/12 = 8.33% of execution resources is wasted for packet parsing and scheduling. RPCValet [17] takes a better approach and has a hardware implementation of a global queue, but lacks preemption and configurability.

**4. Inter-thread communication overhead.** Any system that schedules on the host requires inter-thread communication to move the requests between cores. The overhead associated with such communication can severely limit a system's performance. For example, the high work-stealing rate needed for highly-variable workloads and the high overhead of work stealing render ZygOS unusable [23, 34]. Shinjuku [23] requires inter-thread communication between the networking subsystem, the dispatcher, and worker cores. We measure that this communication causes 2 $\mu$s of additional tail latency for requests that require minimal application work compared to when all processing is performed by one thread. This latency overhead makes it suboptimal for the ultra low latency workloads supported by MICA [27].

## 2.3 How and where should we do scheduling?

As demonstrated by Shinjuku [23], for a scheduling architecture to be as general as possible and cover all types of workloads, it must be centralized and preemptive. Unfortunately, Shinjuku also demonstrated that any software implementation of these two key properties has limited scalability and high latency overhead. The systems that can scale to very high request counts are the ones that delegate scheduling to the hardware, whether that is RSS for IX [15], Flow Director for MICA [27], or new specialized hardware for RPCValet [17]. The question now is whether it is possible to implement centralized preemptive scheduling in hardware. RPCValet proposed a way to implement centralized scheduling, but the implementation is too invasive and requires significant modifications to existing processors. Another option is to create a new hardware accelerator that receives packets from the NIC and schedules application-level requests to cores. Such a design would add an extra hop in the requests' path and therefore add additional latency.

We argue that *the correct place to do request scheduling is the NIC*, before requests arrive at the host, consume any resources, and suffer additional latency. SmartNICs can implement scheduling as they have programmable pipelines [10], FPGAs [9], and even general purpose cores [4]. Existing frameworks leverage these capabilities both for packet processing and request scheduling. FlexNIC [24] uses a match-action (M+A) pipeline to modify incoming packets and either send responses via the network or steer packets to specific CPU cores. In the latter case, only the needed parts of the packet are sent via DMA to the host and packet steering is specified by the M+A rules, such as a key-based hash in a key-value store. Developers use a custom programming language based on P4 to specify the M+A rules. Floem [33] provides a queue-based API that allows programmers to more easily offload parts of an application to the NIC.

While these frameworks can accommodate the scheduling policies used by MICA and IX and achieve high scalability for homogeneous workloads like KVS, they lack one key abstraction necessary for centralized preemptive scheduling. *Host cores need to provide feedback to the SmartNIC* at a fine granularity. More specifically, they have to indicate whether they are busy or ready to receive more work.

# 3 CASE STUDY: SHINJUKU ON A SMARTNIC

The optimal place to do scheduling is on the SmartNIC as it has a centralized view of all packets entering and leaving the system. We first describe what an ideal SmartNIC tailored for request scheduling would look like. While such a SmartNIC does not yet exist, we approximate it by implementing centralized preemptive scheduling on an existing SoC-based SmartNIC. We offload Shinjuku's networking subsystem and dispatcher from an x86 host server CPU to a Broadcom Stingray SmartNIC [4]. In §4 and §5, we evaluate and describe the limitations of this hardware.

## 3.1 What is an ideal SmartNIC?

SmartNICs with programmable pipelines [10], FPGAs [9], or ASICs would be ideal as they process packets and schedule requests at line rate. They also have on-chip memory to store scheduling state and can dynamically steer packets to cores. The one thing all such SmartNICs lack is a high throughput, low latency communication path and shared coherent memory with the host CPU. The host cores would use that coherent memory to continuously provide feedback

at fine granularity to the NIC and drive its functionality. For example, if the SmartNIC is used for scheduling, this feedback would include data such as the instantaneous per-core load or performance counter data used to predict the state of each core's caches and provide good scheduling affinity.

## 3.2 SmartNIC Requirements

The ideal SmartNIC does not yet exist. However, we can offload the Shinjuku networking subsystem and dispatcher (described in §2.1) to existing SmartNICs, as long as they support the functionality below. Any system we build with existing SmartNICs is suboptimal. The limitations are discussed in §5.1.

**1. The SmartNIC steers packets to specific host server cores.** The SmartNIC should be able to address requests to specific cores and those cores should be able to receive the requests without coordinating with other cores.

**2. The host server CPU provides load feedback to the SmartNIC.** The host server CPU must be able to communicate with the SmartNIC at fine granularity, whether via shared memory, network packets, or some other mechanism.

**3. The SmartNIC stores system state and task queues.** The state and task queues can be stored in onboard SRAM. The SmartNIC must store these data so it can make smart scheduling decisions.

**4. The SmartNIC interrupts specific host server cores.** The SmartNIC must be able to interrupt host server cores to implement preemptive scheduling.

## 3.3 Which SmartNIC did we use?

The Broadcom Stingray SmartNIC (PS225) [4], a recently released SmartNIC, meets the requirements in §3.2. The Stingray contains an internal SoC equipped with a dual port 10GbE network interface, 8 ARMv8 A72 64-bit cores, 4 GB DDR4 RAM, and 12 GB flash storage. The ARM CPU is connected internally to the NIC and the NIC itself is connected to the host server via PCIe x8. The NIC presents network interfaces, each with a unique MAC address, to both the host server CPU and the ARM CPU. When a packet arrives, it is steered to the proper CPU based on the MAC address in the Ethernet header. When the ARM CPU wants to send a byte to a host CPU (or vice versa), it constructs a network packet, places the byte in the payload, sets the MAC address of the host CPU's interface in the Ethernet header, and sends the packet via the NIC. The ARM CPU to host CPU communication latency is 2.56 $\mu$s. It is not possible to implement lower-overhead communication as the ARM CPU and the host CPU do not share physical memory and are physically and logically separated by the NIC.

## 3.4 System Design and Implementation

The Shinjuku networking subsystem and dispatcher run on the ARM cores in the Broadcom Stingray SmartNIC and the workers run on the x86 server host cores. Figure 1 shows the path a packet travels through the system. ❶ A packet is received by the SmartNIC and processed by the networking subsystem. ❷ The networking subsystem passes the application request to the dispatcher. ❸ The dispatcher hands off the request to the worker through the Stingray. ❹ If the worker doesn't finish the request in the time slice, the worker is preempted. ❺ The worker notifies the dispatcher that it has either finished processing the request or received an interrupt. If the worker finished the request, the worker also sends a response to the client.

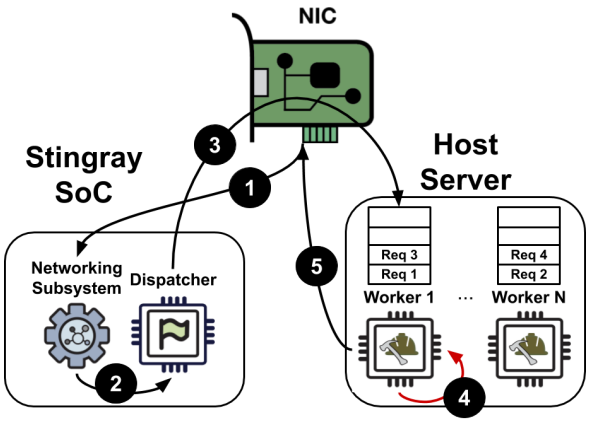


**Figure 1: Shinjuku-Offload system design.**

*3.4.1 Networking Subsystem and Dispatcher.* The networking subsystem and dispatcher run as a DPDK [1] process on the SmartNIC ARM CPU with each component pinned to separate hardware threads. The networking subsystem polls on a physical network interface and parses UDP packets. The networker places application-level requests into shared memory. The dispatcher receives requests from the networker and places them into a FIFO task queue. The dispatcher assigns the request at the front of the queue to an available worker, and sends the request to the worker via a network packet. The worker will later notify the dispatcher that a request has been finished or preempted. If the request has finished, no further action is needed from the dispatcher. If the request has been preempted, the dispatcher adds the request to the end of the task queue. Once the request reaches the front of the queue again, it can be assigned to any worker, not necessarily the worker that handled it first.

Due to the high overhead of constructing and sending packets, the dispatcher's functionality is split across three ARM cores. One core is dedicated to managing the task queue, enqueuing new and preempted requests along with dequeuing requests and assigning them to idle workers. A second core is dedicated to placing the dequeued requests into packets and sending the packets to workers. A third core is dedicated to polling for response packets from workers and

parsing the responses. These three cores communicate via shared memory.

*3.4.2 Dispatcher-Worker Communication.* The dispatcher and the workers communicate with each other by sending UDP packets via the NIC. Each worker polls its own network interface, as does the dispatcher. SR-IOV is used to create enough virtual network interfaces such that there is one virtual interface per worker. To send a packet to a worker, the dispatcher sets the MAC address of the worker's network interface in the packet's Ethernet header. The worker sets the MAC address of the dispatcher's network interface in the Ethernet header of response packets.

*3.4.3 Workers.* The workers run on the x86 host server CPU. Each worker runs in a DPDK thread within a single Dune [14] process. Each worker polls on its own network interface until it receives a request from the dispatcher. Upon receipt of a request, the worker spawns a new context and executes the request (or reuses a context if the request had previously been preempted). If the request finishes, the worker sends a response to the client. If the request is preempted by the local APIC timer (see §3.4.4), the worker notifies the dispatcher of the preemption and saves the work it has done so far (e.g., stack and register contents) in host DRAM.

*3.4.4 Preemption.* Workers are preempted if they do not finish executing a request within the time slice (e.g., 10 $\mu$s).

The Stingray could interrupt CPU cores by sending network packets, but given the communication latency of 2.56 $\mu$s, this would not be efficient. Instead, upon receipt of a request, workers set a timer that will deliver an interrupt once the time slice expires. Setting the Linux timer and receiving a timer signal is expensive. To reduce that cost, the Dune kernel module maps the local APIC's timer registers into guest physical address space so that workers can set the timer directly. The timer interrupt is delivered as a low-overhead posted interrupt [23]. This approach reduces the cost of setting timers from 610 cycles to 40 (93%) and of receiving timer interrupts from 4193 cycles to 1272 (70%).

While this interrupt mechanism is more efficient than having the NIC send packet-based interrupts when the system is under heavy load, a downside is that a worker is interrupted even if there is no additional work waiting at the NIC. The NIC should have a low-latency path (faster than 2.56 $\mu$s) through which it can deliver interrupts. If an interrupt takes 2.56 $\mu$s to be delivered, then between when the dispatcher sends the interrupt and when the worker receives it, the worker could finish the task and move onto the next task, causing the next task to be unnecessarily preempted.

*3.4.5 Queuing Optimization.* Given the communication latency between the Stingray ARM CPU and the host server CPU, how can the dispatcher ensure that a pending request is waiting in a worker's RX queue when the worker is preempted or finishes a request, so that the worker is always busy?

Recall that each worker polls on a network interface. The dispatcher ensures that at least one request is waiting in the worker's network RX queue while the worker is executing a request (i.e., there are at least two outstanding requests at the worker). When the worker finishes a request, the request is discarded; when the worker preempts a request, the worker sends the request back to the dispatcher and the dispatcher adds the request to the end of the centralized task queue. When the worker wants to execute another request, it pulls out the next request that the dispatcher stashed in the worker's network interface RX queue and begins work immediately. As shown in Figure 3, we improved system throughput by 250% when keeping five outstanding requests at all workers at any time.

## 4 EVALUATION

We compare the vanilla Shinjuku system to the Shinjuku system with the networking subsystem and dispatcher offloaded to the Stingray SmartNIC. We will refer to the latter system as Shinjuku-Offload. Both systems run on a server with two 2.3 GHz Intel E5-2658 processors and 128 GB DRAM. The server runs Ubuntu 16.0.4 LTS with the Linux 4.4.0 kernel and the Stingray SmartNIC runs Poke 2.5 with the Linux 4.14.65 kernel. Shinjuku uses an Intel 82599ES 10Gb NIC and Shinjuku-Offload uses the Broadcom Stingray SmartNIC, described in §3.3. We use an open loop load generator similar to mutilate [25] that transmits requests over UDP. The client machines each include two Intel Xeon E5-2630 CPUs at 2.3GHz and Intel 82599ES 10Gb NICs. We refer to the 99th percentile latency as the *tail latency*.

### 4.1 Synthetic Workload

We use a synthetic workload to compare the performance of both systems. The requests contain fake work that keeps the server busy for a specific amount of time. These requests allow us to emulate different workload distributions.

The first workload, shown in Figure 2 has a bimodal service time distribution where 99.5% of requests have a 5 $\mu$s service time and 0.5% of requests have a 100 $\mu$s service time. The preemption time slice is 10 $\mu$s. Shinjuku pins the networking subsystem and the dispatcher to separate hyperthreads on the same physical core and pins 3 workers to their own hyperthreads on 3 physical cores. Shinjuku-Offload pins the networking subsystem and dispatcher to their own threads on the ARM CPU and pins 4 workers to their own hyperthreads on 4 physical cores. Just like Shinjuku, Shinjuku-Offload handles highly dispersive workloads while maintaining low tail latencies.

We show the benefit of our Shinjuku-Offload queuing optimization in Figure 3. We use fixed 1 $\mu$s service times and vary the number of outstanding requests per worker. We

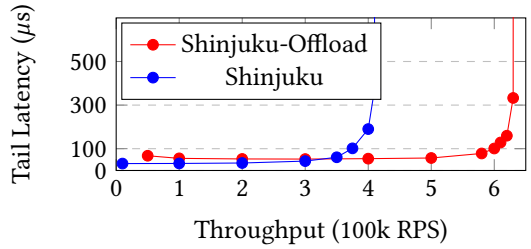


**Figure 2: 99.5% 5 µs, 0.5% 100 µs bimodal distribution. Shinjuku has 3 workers and Shinjuku-Offload has 4 (up to 4 outstanding requests). The preemption time slice is 10 µs.**

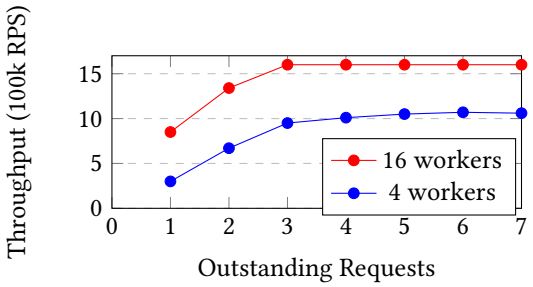


**Figure 3: Fixed 1 µs service time. Shinjuku-Offload.**

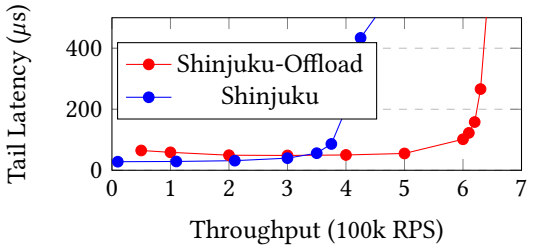


**Figure 4: Fixed 5 µs service time. Shinjuku has 3 workers and Shinjuku-Offload has 4 (up to 4 outstanding requests).**

run Shinjuku-Offload with 4 workers and 16 workers. As we increase the outstanding requests from 1 to 5 requests in the case of 4 workers, throughput increases by 250%, and levels out at 5 requests. As we increase the outstanding requests from 1 to 3 requests in the case of 16 workers, throughput increases by 88%, and levels out at 3 requests.

We have observed that tail latency increases as the number of outstanding requests gets larger, so it is best to set it to 5.

The second workload, shown in Figure 4 has a fixed 5 µs service time. Shinjuku has 3 workers and Shinjuku-Offload has 4 workers. We turned off preemption for the fixed workloads. Shinjuku-Offload outperforms Shinjuku as Shinjuku-Offload has an extra worker, since its networking subsystem and dispatcher are running on the SmartNIC.

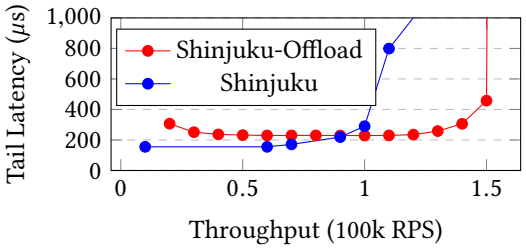


**Figure 5: Fixed 100 µs service time. Shinjuku has 15 workers and Shinjuku-Offload has 16 (up to 2 outstanding requests).**

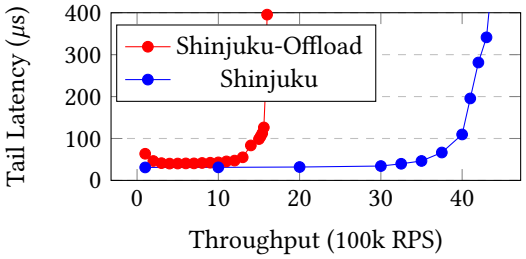


**Figure 6: Fixed 1 µs service time. Shinjuku has 15 workers and Shinjuku-Offload has 16 (up to 5 outstanding requests).**

The next workload, shown in Figure 5 has a fixed request service time of 100 µs. Shinjuku has 15 workers and Shinjuku-Offload has 16 workers. Shinjuku-Offload outperforms Shinjuku for a large number of workers when the request service time is large.

The final workload, shown in Figure 6 has a fixed request service time of 1 µs. As before, Shinjuku has 15 workers and Shinjuku-Offload has 16 workers. However, Shinjuku greatly outperforms Shinjuku-Offload. From the point of saturation in Figure 5 to the point of saturation in Figure 6, the Shinjuku-Offload workers spend 110% more time waiting for work from the dispatcher. The Shinjuku-Offload dispatcher is a bottleneck since (1) it runs on the slower ARM CPU and (2) there is much higher communication overhead in Shinjuku-Offload. The dispatcher must send *packets* to communicate with workers rather than write to a cache line, as in Shinjuku.

## 5 DISCUSSION

### 5.1 SoC-based SmartNIC Limitations

The limitations of SoC-based SmartNICs are on clear display in Figure 6 as the Shinjuku dispatcher outperforms the Shinjuku-Offload dispatcher. The SmartNIC dispatcher is the bottleneck. Little more can be done in software to improve the throughput; rather, the bottleneck is caused by the hardware. The suggestions below will eliminate the bottleneck and can all be implemented in hardware.

**1. The hardware needs to support line-rate scheduling.** The ARM cores are too slow to schedule requests at line

rate, and any general-purpose CPU would likely be unable to maintain line rate. Scheduling work is so simple and parallel that an FPGA or ASIC is a better fit. While an FPGA [20, 29] or ASIC could be installed in the SmartNIC, it needs to incorporate load feedback from the host. Loom [38] and PIEO [37] explore packet scheduling in hardware, but these approaches focus on scheduling *outgoing* packets on *one* wire. They do not incorporate fine-grained load feedback from the host or schedule for multiple destinations (i.e., multiple cores rather than one wire). Elastic RSS [35] is a customized version of hardware-based RSS that provisions cores for applications on the $\mu$s scale and incorporates fine-grained load feedback, but only scheduling parameters can be changed in the implementation—the scheduling policy itself is fixed upfront. ADM [36]—a more flexible hardware-software design that passes lightweight messages to cores without traversing cache hierarchies—could be used to incorporate this feedback and support line-rate scheduling.

**2. There should be low communication overhead between the dispatcher and workers.** The dispatcher and workers send packets through the NIC to communicate. It takes 2.56 $\mu$s to both construct a packet and have it travel through the NIC one-way. The latency is hidden by the queuing optimization, but the dispatcher cannot do as fine-grained scheduling, causing higher tail latency [17].

This inefficient communication interface is required by the Stingray hardware and there is no lower-latency path through which the SmartNIC cores and host can communicate. The host CPU and the SmartNIC cores are physically separated by the NIC itself, so the SmartNIC cores cannot directly initiate low-overhead PCIe transactions on the host.

Ideally, the SmartNIC cores would share memory with the host server CPU and have a faster communication path. RPCValet [17] recognizes this and closely integrates the network interface with cores, though their proposal requires a departure from existing server designs. Conversely, giving SmartNIC cores direct access to the host PCIe bus would be beneficial, though PCIe transactions alone are not enough. CXL [5] can provide a path between the host CPU and the SmartNIC with high throughput, low latency, and most importantly, coherent shared memory (which PCIe does not provide). With CXL, the SmartNIC writes its scheduling decisions directly to host memory where polling workers see them. When workers finish, they set a completion flag and the SmartNIC snoops on the resulting coherence traffic [5].

OpenCAPI [11], CCIX [3], and Gen-Z [8] are other high throughput, low latency technologies for connecting devices and sharing memory that can improve performance. The lowest latency available through all of these mechanisms—likely a few hundred nanoseconds to a microsecond for a one-way trip—is the lowest foreseeable communication latency in a modern system barring deeper NIC–CPU integration.

**3. The SmartNIC should directly send interrupts to the host server CPU in order to accommodate many different scheduling designs.** The SmartNIC can send packets to trigger interrupts, but there is 2.56 $\mu$s of latency from when we start constructing an interrupt message to when the interrupt handler runs. We use the local APIC timer instead. However, using a local timer relies on the queuing optimization to ensure a request is waiting for the worker when it is preempted, which causes higher tail latency. Furthermore, in order to use the local timer efficiently, we need to virtualize our process using Dune, which adds design complexity and virtualization overhead.

**4. The SmartNIC should not add considerable complexity to the system.** SmartNICs have their own architecture, which may have little in common with the host architecture. Furthermore, it is not immediately clear what type of load information should be sent from the host to the SmartNIC. To ensure the additional complexity does not outweigh the performance benefits, we should develop libraries and tools that make it easy to specify scheduling functions for the SmartNIC and for the OS and applications to send information to the SmartNIC. Floem [33] and iPipe [28] simplify the offload of applications to SmartNICs, though they do not target the low latency scheduling and communication functions in our design, so this is a future research direction.

## 5.2 Additional Use Cases

**DDIO for high-level caches.** Intel Direct Data I/O (DDIO) technology [6] allows NICs to place packets directly into the CPU's LLC rather than main memory, delivering increased bandwidth, lower latency, and reduced power consumption. To avoid cache pollution, direct packet placement to a high-level cache is not allowed. However, Shinjuku's scheduling algorithm guarantees that at most one request is in-flight at any time on each core and Shinjuku-Offload may be able to have fewer outstanding requests at each core with CXL.

Thus, a NIC that uses this algorithm can place network packets even into the L1 cache without danger of filling it.

**Congestion Control.** Recent research proposes the co-design of congestion control with OS scheduling [30]. The network's goal is not to deliver packets as fast as possible but rather just in time for processing. Such a congestion control scheme requires fine-grained data from both the network and the host cores and thus would benefit from our proposal.

# 6 CONCLUSION

Offloading scheduling to a SmartNIC will improve the scalability and performance of existing network applications with highly dispersive service times. The optimal place to do request scheduling is in the SmartNIC as it has a centralized view of all packets entering and leaving the system. We propose an ideal SmartNIC that schedules packets at line rate, has a high throughput and low latency communication path with the host server, shares coherent memory with the host server, and most importantly, instantly incorporates host load feedback into its scheduling decisions. While this ideal

SmartNIC does not yet exist, we show promising results with a current SmartNIC and suggest improvements.

## 7 ACKNOWLEDGMENTS

We thank Simon Peter, Xi Wang, Amy Ousterhout, and the anonymous HotNets reviewers for their helpful feedback. This work was supported by a gift from Google.

## REFERENCES

[1] [n. d.]. Data plane development kit. http://www.dpdk.org/. ([n. d.]). Last accessed: 2019-06-26.

[2] [n. d.]. Memcached. https://memcached.org/. ([n. d.]). Last accessed: 2019-06-26.

[3] [n. d.]. An Introduction to CCIX https://docs.wixstatic.com/ugd/0c1418_c6d7ec2210ae47f99f58042df0006c3d.pdf. ([n. d.]). Last accessed: 2019-06-26.

[4] [n. d.]. Broadcom Stingray https://www.broadcom.com/products/ethernet-connectivity/smartnic/bcm58800. ([n. d.]). Last accessed: 2019-06-26.

[5] [n. d.]. Compute Express Link https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf. ([n. d.]). Last accessed: 2019-06-26.

[6] [n. d.]. Direct Data I/O https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. ([n. d.]). Last accessed: 2019-06-26.

[7] [n. d.]. Flow Director https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. ([n. d.]). Last accessed: 2019-06-26.

[8] [n. d.]. Gen-Z OVERVIEW http://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview-V1.pdf. ([n. d.]). Last accessed: 2019-06-26.

[9] [n. d.]. Mellanox PB Innova-2 Flex http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. ([n. d.]). Last accessed: 2019-06-26.

[10] [n. d.]. Netronome NFP 4000 https://www.netronome.com/m/documents/PB_NFP-4000.pdf. ([n. d.]). Last accessed: 2019-06-26.

[11] [n. d.]. OpenCAPI https://opencapi.org/wp-content/uploads/2016/09/OpenCAPI-Exhibit-SC17.pdf. ([n. d.]). Last accessed: 2019-06-26.

[12] [n. d.]. Receive Side Scaling http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx. ([n. d.]). Last accessed: 2019-06-26.

[13] [n. d.]. RocksDB http://rocksdb.org/. ([n. d.]). Last accessed: 2019-06-26.

[14] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Hollywood, CA, USA, 335–348. http://dl.acm.org/citation.cfm?id=2387880.2387913

[15] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[17] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPC-Valet: NI-Driven Tail-Aware Balancing of $\mu$s-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, Providence, RI, USA, 35–48. https://doi.org/10.1145/3297858.3304070

[18] Jeffrey Dean and Luiz AndrÃľ Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[19] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 79–94. https://www.usenix.org/conference/nsdi19/presentation/didona

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

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/atc19/presentation/fouladi

[22] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[23] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for $\mu$second-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[24] 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, Atlanta, Georgia, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[25] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/2592798.2592821

[26] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. 2016. Work Stealing for Interactive Services to Meet Target Latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, Barcelona, Spain, Article 14, 13 pages. https://doi.org/10.1145/2851141.2851151

[27] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[28] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications

Onto smartNICs Using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA, 318–333. https://doi.org/10.1145/3341302.3342079

[29] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. 2008. NetFPGA: Reusable Router Architecture for Experimental Research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '08)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/1397718.1397720

[30] Amy Ousterhout, Adam Belay, and Irene Zhang. 2019. Just In Time Delivery: Leveraging Operating Systems Knowledge for Better Datacenter Congestion Control. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/ousterhout

[31] 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)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[32] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.

[33] 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

[34] 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 (SOSP '17)*. ACM, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[35] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019 (APNet '19)*. ACM, New York, NY, USA, 71–77. https://doi.org/10.1145/3343180.3343184

[36] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible Architectural Support for Fine-grain Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/1736020.1736055

[37] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, 367âĂŞ379. https://doi.org/10.1145/3341302.3342090

[38] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 33–46. https://www.usenix.org/conference/nsdi19/presentation/stephens

[39] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, Farminton, Pennsylvania, 18–32. https://doi.org/10.1145/2517349.2522713

[40] Adam Wierman and Bert Zwart. 2012. Is Tail-Optimal Scheduling Possible? *Oper. Res.* 60, 5 (Sept. 2012), 1249–1257. https://doi.org/10.1287/opre.1120.1086

[41] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. 2019. A Case for Managed and Model-less Inference Serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, Bertinoro, Italy, 184–191. https://doi.org/10.1145/3317550.3321443