Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs

Alexander Rucker
Stanford University

Tushar Swamy
Stanford University

Muhammad Shahbaz
Stanford University

Kunle Olukotun
Stanford University

ABSTRACT

Meeting Service-Level Objectives (SLOs) for workloads in today's datacenter environments places stringent demands on end-host servers: work conservation, tolerance to varying request service time distributions, high throughput, and CPU efficiency. Beginning with Receive Side Scaling (RSS), various schedulers have been proposed to steer packets to cores while preserving locality. However, these techniques are either too *inflexible* (randomly steering traffic at the NIC) or *slow* (bottlenecked by a central CPU-based scheduler).

In this paper, we present *Elastic RSS* (eRSS), a system that extends traditional RSS by scheduling packets *and* cores using emerging programmable NICs with new abstractions (e.g., map-reduce). Operating at the NIC with minimal intervention from the host CPU, eRSS responds to load shifts at line rate and on a per-packet basis. eRSS also supports distributed packet stealing and fast preemption per-core to improve tail latency under heavy-tailed service time distributions. Our preliminary evaluation shows that eRSS increases CPU efficiency while responding to rapid load changes and meeting real-world tail latency constraints of 100 μs.

CCS CONCEPTS

• **Networks** → **End nodes**; **Packet scheduling**; **Cloud computing**; *Programmable networks*; Traffic engineering algorithms;

KEYWORDS

RSS; programmable NICs; map-reduce; Taurus;

APNet '19, August 17–18, 2019, Beijing, China

ACM ISBN 978-1-4503-7635-8/19/08…$15.00
https://doi.org/10.1145/3343180.3343184

ACM Reference Format:
Alexander Rucker, Tushar Swamy, Muhammad Shahbaz, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *APNet '19: 3rd Asia-Pacific Workshop on Networking, August 17–18, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343180.3343184

1 INTRODUCTION

Modern data centers are highly-multiplexed environments, running myriad workloads sharing a common infrastructure (e.g., servers and switches). These workloads range from latency-sensitive tasks (e.g., in-memory databases [20] and key-value storage [1, 16]) to batch-processing jobs (e.g., deep learning [14] and garbage collection [12]), each with specific Service-Level Objectives (SLOs) [6]. So far, techniques for workload scheduling have largely been optimized for either latency-sensitive tasks with time-critical, small jobs running at fine timescales (typically μs) [11] or batch applications with non-interactive, throughput intensive jobs running at coarse timescales [9], but not both.

Recent proposals, such as Shenango [15] and Shinjuku [11], meet tail latency SLOs and achieve high CPU efficiency while tolerating dispersion in requests' service time distributions by scheduling and preempting cores at extremely short intervals (5 μs). However, Shenango and Shinjuku sacrifice throughput by dedicating a CPU core to fine-grained task scheduling and packet forwarding (Figure 1a); this can not saturate current 10 Gbps links with minimum-size packets and will not scale to future 100 Gbps NICs.

Prior proposals using dedicated hardware, such as IX [4] and ZygOS [17] with RSS-enabled NICs (Figure 1b), were able to handle requests at line rate. But, these approaches were not CPU efficient—they used random queueing, which has poor theoretical performance, and over-provisioned cores to handle bursts. They were also susceptible to long tail latencies due to head-of-line blocking under dispersive service time distributions.

Ideally, any solution for workload scheduling in today's shared data centers must run at line rate and be work conserving, dispersion tolerant, and CPU efficient, as summarized in Table 1. Shenango [15] and Shinjuku [11] showed that a

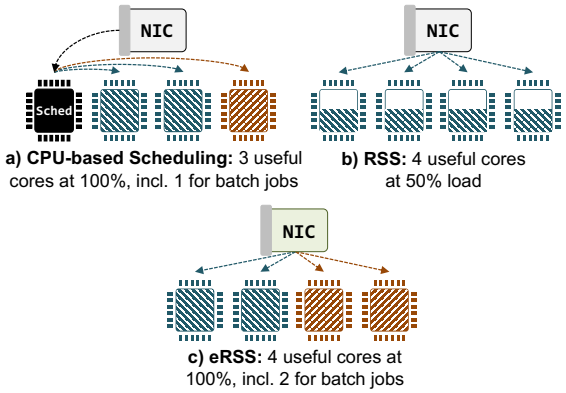

Figure 1: eRSS achieves better core utilization than RSS-only techniques without dedicating a CPU core for packet forwarding.

scheduler can be both work conserving and CPU efficient, even under dispersive request distributions, if it schedules packets and cores together at a fine-grained timescale. We argue that schedulers can provide these advantages and sustain higher throughput by co-scheduling packets and cores directly at the NIC.

Modern SmartNICs (e.g., Mellanox Bluefield [2]) can theoretically schedule packets using on-chip ARM cores; however, these cores are designed for lightweight management tasks and are too slow. PISA-based NICs with a match-action pipeline [8], on the other hand, would require consistent updates to flow tables [11] using Programmed IO (PIO) over a PCIe bus, resulting in scheduling latencies of approximately 100 μs [15]. However, emerging programmable NICs, like Taurus and its map-reduce abstraction [19], alleviate many of these concerns.

In this paper, we present *Elastic RSS* (eRSS), a system that simultaneously allocates cores and schedules packets at line-rate in the NIC. Our implementation of eRSS uses the map-reduce primitives introduced by Taurus [19] for packet scheduling. The map operation calculates the weighted consistent-hashing distance to each core for a packet, and the reduce stage finds the closest allocated core. By scheduling at the NIC (Figure 1c), we avoid both the need to dedicate a core for scheduling and the resulting packet-forwarding bottleneck. We introduce a core-scheduling algorithm that responds to rapid load shifts: if a burst arrives, the NIC detects it within microseconds and allocates the necessary cores. The host CPU performs more complex but infrequent tasks, such as estimating the throughput of a core, scheduling background tasks, and periodically updating the NIC's shadow counters. We evaluate this algorithm and show that it decreases core utilization while maintaining latency, even for sudden bursts.

We start by discussing the recent related work on low-latency schedulers and programmable NICs.

Scheme	Packet Stealing	Dispersion Tolerance	CPU Efficiency	Throughput (64B Pkt)
IX [4]	no	none	none	line-rate
ZygOS [17]	yes	low	none	line-rate
Shinjuku [11]	yes	high	none	< 10 Gbps
Shenango [15]	yes	low	high	< 6 Gbps
eRSS	yes	high	high	line-rate

Table 1: eRSS vs. existing μs-scale approaches.

2 BACKGROUND & RELATED WORK

Low-latency task scheduling. Table 1 compares state-of-the-art schedulers with eRSS and highlights the dichotomy between CPU-efficient, dispersion-tolerant stacks and high-throughput ones. For example, IX [4], a data-plane operating system, uses adaptive batching to schedule requests at high throughput. The NIC, using flow-consistent hashing (i.e., RSS), distributes packets to cores, which then process these packets in a distributed, First Come First Serve (d-FCFS) manner. Each core polls its packet queue to reduce jitter (e.g., interrupt variability) and decrease tail latency. However, IX over-provisions cores to respond to bursts and has higher latencies under heavy-tailed request distributions due to its run-to-completion execution model and use of RSS.

ZygOS [17] improves upon IX by letting cores steal packets from other cores. Packet stealing improves tail latency for requests that suffer head-of-line blocking at cores serving earlier requests. ZygOS implements a work conserving scheduler via packet stealing; it adds specialized queues to each core that other cores can query using interprocessor interrupts. Unlike IX, ZygOS approximates a centralized FCFS (c-FCFS) model, where all cores serve a single queue. Still, despite these improvements, both ZygOS and IX perform poorly when variance among service times is high. This is because FCFS leads to high tail latencies for short requests when cores are busy servicing longer requests.

Shinjuku [11] approximates centralized Processor Sharing (c-PS) using preemption, which outperforms FCFS for heavy-tailed and highly-dispersive workloads [21]. Using selective preemption, Shinjuku implements c-PS for large packets while operating as a (more efficient) c-FCFS system for average and small requests. However, allocating a dedicated CPU core to packet forwarding, scheduling jobs, and preempting tasks at μs-scale limits Shinjuku's maximum throughput (Table 1). Moreover, because Shinjuku must always allocate enough cores for bursts, it has poor CPU utilization.

Shenango [15], on the other hand, improves CPU efficiency while achieving tail latencies comparable to ZygOS. Like Shinjuku, Shenango dedicates a core, but uses it to frequently schedule CPU cores instead of preempting long-running requests. Allocating cores on a 5 μs granularity

improves efficiency and enables Shenango to rapidly respond to bursts using minimal CPU cores. However, because Shenango dedicates a core to packet forwarding, it can not scale to many-core systems with line rates over 40 Gbps.

Using emerging programmable data planes, eRSS supports Shenango's fine-grained core allocation natively and Shinjuku's c-PS scheduling through a light-weight runtime; this provides low tail latency and CPU-efficiency at line rate.

Emerging NIC data planes. ARM-based SmartNICs [2] and FPGA- and PISA-based data planes [7, 8] can accelerate simple workloads (like key-value stores), but are either too slow or too inflexible to run complex tasks like core scheduling. Furthermore, the slowdown of Moore's Law and rapid increase in network traffic requires domain-specific designs to meet SLOs for modern workloads. Taurus [19] is an emerging NIC platform that extends a PISA data plane (i.e., match-action tables [5]) with a Coarse-Grained Reconfigurable Architecture (CGRA); Taurus increases compute density and provides map-reduce parallelism per-packet. We model eRSS as operating within Taurus's constraints, using match-action tables for workload estimation and map-reduce parallelism for consistent hashing (Figure 2).

3 DESIGN

In eRSS, we divide processing across two timescales: fine-grained, per-packet processing at the NIC (§3.1) and coarse-grained state management at the host CPU (§3.2). eRSS's NIC pipeline (Figure 2) decides how many cores to allocate for each *scaling group*, which could be an application or a VM; it also routes each packet to a core. The NIC's management core implements a control loop using shadow counters to estimate CPU queue depth and periodically load balance packets among cores within a scaling group. Finally, a software manager on the host CPU estimates request service times, allocates slack cores between scaling groups, and updates NIC shadow counters on a coarser timescale (e.g., 100 μs).

3.1 Fine-Grained Per-Packet Processing

Algorithm 1 shows the steps each packet passes through in eRSS's NIC pipeline. We first assign an incoming packet to a scaling group (Line 1), which is uniquely identified by the packet header (e.g., 5-tuple). eRSS makes decisions for each scaling group independently, including workload estimation, core allocation, and packet steering.

Workload estimation. After identifying a scaling group, eRSS next estimates the associated workload (Line 2). eRSS implements two workload estimators to provision an adequate number of cores for each scaling group. The first is *instantaneous workload* (in B/s), an exponentially decaying counter that estimates throughput over a short time interval (e.g., 5 μs). The second is *residual workload* (in B), a counter

Algorithm 1 eRSS's NIC execution pipeline

State: workload W_g, residual-bytes b_g, last-update t'_g, cores c_g, weights w_g^v, queues q_g^v, virtual-core vC
Constants: hash-centroids H^v, residual-delay $d_i = 5\mu s$, core-delay $d_d = 500ns$, residual-thresh $T_i = 100kB$ and $T_d = 1$ cyc, slack $S = 0.75$
Inputs: time t, packet p, throughput t_g, packet-hash h, max-cores M_g, v2p-mapping v_g
Outputs: physical-cores

1: $g \leftarrow$ Get-Scaling-Group(p)
2: $W_g \leftarrow W_g + \text{Size}(p), \quad b_g \leftarrow b_g + \text{Size}(p)$
3: **if** $(W_g \geq t_g * c_g * S \vee$
4: $\quad (b_g \geq T_d \wedge t - t'_g \geq d_i))$ **then**
5: $\quad c_g \leftarrow c_g + 1, \quad t'_g \leftarrow t$
6: $\quad$ Interrupt(c_g)
7: **else if** $(W_g \leq t_g * (c_g - 1) * S * 0.9 \wedge$
8: $\quad b_g < c_g * t_g * T_d \wedge$
9: $\quad$ Time() $- t'_g \geq d_d \wedge$
10: $\quad c_g > 0)$ **then**
11: $\quad$ Signal-In-Band(c_g)
12: $\quad c_g \leftarrow c_g - 1, \quad t'_g \leftarrow t$
13: $(vC, best) \leftarrow (0, \infty)$
14: **for all** $i \in [0, c_g]$ **do**
15: $\quad$ **if** $|H^i - h| * W_g^i < best$ **then**
16: $\quad\quad (vC, best) \leftarrow (i, |H^i - h| * W_g^i)$
17: $q_g[vC] \leftarrow q_g[vC] + \text{Size}(p)$
18: **return** $v_g[vC]$

decaying linearly and proportionally to the number of currently allocated cores. The residual counter ensures that there are enough cores left running to drain the queues, even if the instantaneous throughput drops rapidly or can not react to a load spike (e.g., a scaling group runs out of cores). The software runtime estimates *throughput per core*, necessary to determine the number of cores needed for an incoming packet stream. The decaying counters are implemented as registers in a match-action pipeline stage, which the control plane periodically decreases by a constant factor.

Core allocation. The next stage in the eRSS NIC pipeline determines the number of cores needed for a scaling group, i.e., whether too many, too few, or the correct number of cores are currently allocated. If the instantaneous workload is greater than some fraction (e.g., 75%) of the estimated throughput of all cores, eRSS allocates one more core (Line 3). eRSS can also increase the core count if the residual workload is above a certain threshold (Line 4). eRSS limits increments based on residual workload to one every 5 μs—otherwise, a brief excursion above the threshold would result in quickly reaching the maximum core count.

eRSS deallocates cores only when the instantaneous workload falls below 90% of the threshold for one fewer than the number of allocated cores (Line 7) *and* the residual workload

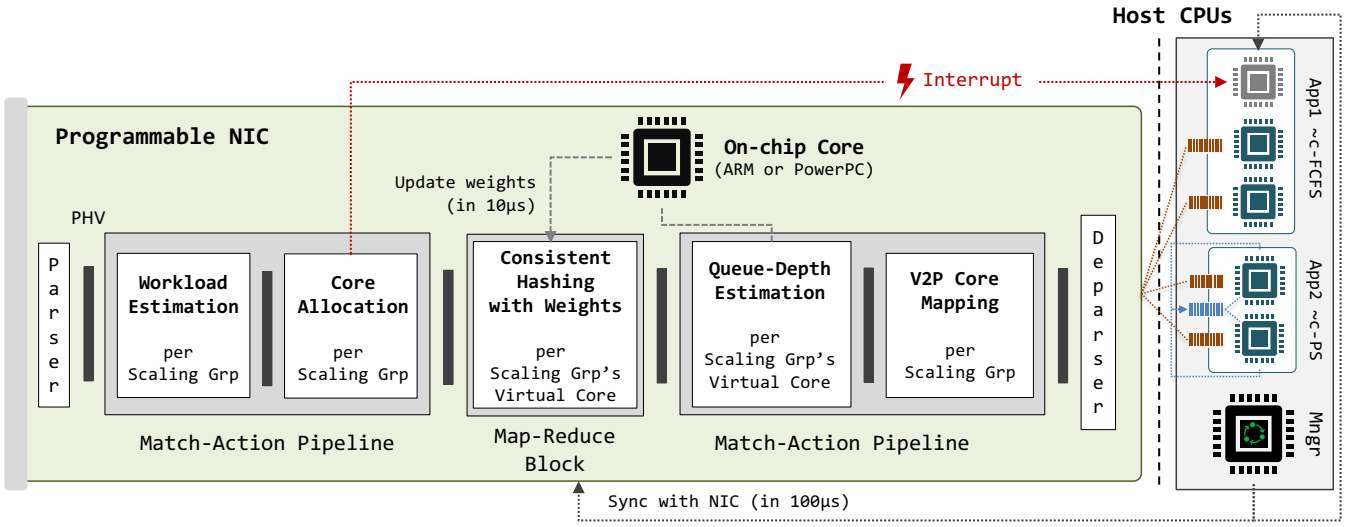

Figure 2: eRSS's NIC pipeline performs core and packet scheduling for each incoming packet. The on-chip core updates weights at approximately 10 μs intervals, and the software manager syncs runtime state (e.g., the true queue depths and per-scaling group core mapping) with the NIC at 100 μs intervals.

is negligible (Line 8). Different increase and decrease thresholds provide hysteresis; eRSS also deallocates cores only once every 500 ns to avoid undershooting the ideal number of cores. eRSS tracks the currently allocated cores using an up-down counter register in Taurus's match-action pipeline for each scaling group.

Notifications via interrupts. Periodically, the software manager determines which slack cores to assign for each scaling group and initializes a sleeping process on each. eRSS notifies the software manager whenever the core count is updated for a given scaling group. When allocating a core, eRSS interrupts the target core and wakes up the sleeping process to begin handling packets (Line 6). In addition to their low latency, interrupts can preempt an unmodified OS task, allowing eRSS to operate alongside existing software. When deallocating a core, eRSS prepends a special header to the last packet sent to that core. This ensures that the software manager deallocates the core only when it has served all pending packets (Line 11). After eRSS yields control, the OS can run other processes (e.g., batch jobs) on the core using its own scheduler (e.g., CFS [13]).

Consistent hashing & virtual–physical core mapping. In eRSS, virtual cores within a scaling group are assigned using a dense, zero-based addressing scheme to simplify allocation: at any given time, virtual cores in the range [0, max] are active. A map operation calculates the weighted distance from the packet's 5-tuple hash to each virtual core, following the scheme described in [18]. We use distances on a three-dimensional torus instead of a ring; this provides more adjacency for load-shifting. The reduce operation then selects the best core: the core with minimum distance to the hash among the allocated virtual cores (i.e., those in [0, max]). This yields a virtual core, which indexes a per-scaling group table to yield a physical core (Line 18).

Updating weights using the on-chip core. eRSS estimates per-virtual core queue depths following each packet (Line 17); periodically, the on-chip core uses the queue depths to update the weights in the consistent hashing stage. These weights help distribute traffic across cores in the presence of both load imbalances for each scaling group and load shifts from adding and removing cores. The on-chip core forms a negative feedback loop when updating the weights, directing traffic from heavily loaded cores to lightly loaded ones. The updates happen relatively infrequently (e.g., every 10 μs), and not on a per-packet basis, to prevent changing allocation decisions based on spurious variations in traffic (i.e., noise).

3.2 Coarse-Grained State Management

eRSS's software manager (Figure 2) synchronizes true per-core queue depths with the NIC to update both shadow and residual workload counters. It also maintains and updates an estimation of packet throughput per scaling group, which allows the NIC to infer a rough mapping of incoming packets to execution time. This accounts for long-term drift in application characteristics and moves complexity from hardware to software. In our preliminary evaluation, we model the per-packet workload as a linear function of packet size, but a more complex system would use additional packet characteristics (e.g., application headers) to predict workload.

Additionally, the software manager allocates cores to scaling groups. In eRSS, cores are independently scheduled for each scaling group; therefore, the same core cannot simultaneously be available to two scaling groups at the NIC. Instead, on a coarse time scale (100 μs), the software manager determines the maximum number of slack cores for each scaling group. It will reclaim slack cores from scaling groups not allocating all their cores and provide them to other scaling groups such that a greater fraction of their cores are allocated. Doing so ensures that each scaling group has headroom to adapt to traffic variations—if a group uses all of its cores within a 100 μs interval, it will receive more cores at the next one. However, a software manager can also enforce scheduling decisions to guarantee SLOs, such as provisioning a minimum number of cores per application.

3.3 Runtime Support & Optimizations

A variety of software techniques, such as packet stealing [11] and preemption [15], exist to avoid head-of-line blocking and approximate c-FCFS or c-PS scheduling. Because eRSS is a hardware solution, it does not dictate the use of a specific software runtime: packet stealing and preemption are infrequent, and need not run at line rate. Furthermore, the software runtime can implement *distributed* packet stealing and preemption to mitigate the need for a centralized scheduler [15]. Rapid preemption can happen on a per-core basis to emulate processor sharing; packets are first sent to a core using eRSS. Then, for each scaling group, long-running requests are preempted and sent to a shared queue (Figure 2, App2 ~c-PS), from which all cores periodically draw packets. For simplicity, in our preliminary evaluation, we do not model preemption or packet stealing.

4 PRELIMINARY EVALUATION

Experiment setup. We built a simulator running a synthetic system model to evaluate eRSS, including request arrival times, background workloads, and realistic flow and packet sizes. Our simulator tracks a fixed number of concurrent flows, with flow and packet sizes drawn from empirically measured distributions [3, 10]. Furthermore, the inter-arrival time between packets uses a Poisson distribution, with λ scaled to reach a target throughput. To model dispersion, the request service times in our simulator are kept proportional to the sizes of the processed packets (e.g., 4000 cycles for a 1500 B packet). We also add 25% processing overhead when moving flows between cores due to cache misses and the need for packet stealing.

Core scheduling. To evaluate how quickly eRSS adapts to bursts, we generate a sequence of bursts with peak intensities increasing to 35 Gbps (Figure 3a), with a baseline load of 4 Gbps. Figure 3b shows that eRSS allocates cores

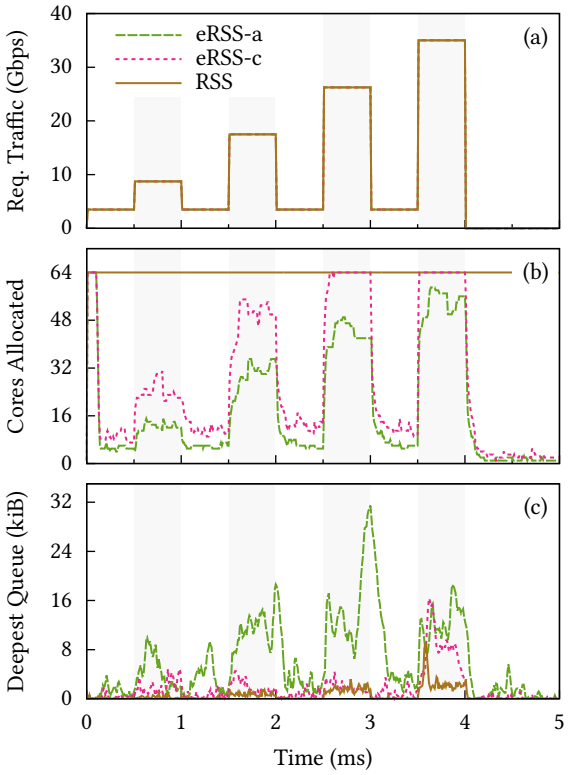

Figure 3: eRSS reacts quickly to traffic variations.

proportionally to the incoming load. Initially, eRSS starts with the maximum number of cores—ensuring that sufficient cores are allocated while eRSS learns the throughput characteristics of the scaling group.

We measure eRSS using two configurations: aggressive (eRSS-a) and conservative (eRSS-c). eRSS-a allocates additional cores to ensure that each is running at < 90% capacity; eRSS-c ensures that cores are running at < 75% capacity, which results in more cores being allocated. As the traffic load decreases, eRSS correspondingly deallocates cores: it removes cores rapidly to match the sharp drop in load, leaving enough cores to ensure that all queues are drained.

Figure 3c shows, at every instant, the depth of the deepest queue in our simulated system. We use the deepest queue's depth as a proxy for the tail latency added by queueing, and therefore eRSS. eRSS has relatively higher queue sizes than traditional RSS, because it both uses fewer cores to process packets—traditional RSS uses all the available cores—and adds transient workload imbalances when reallocating cores. When adding and removing cores from the pool of active cores, eRSS redistributes work across all cores, which requires additional time to move data between cores. However, the added short-term imbalance and data movement taper off after a few weight updates, and each core's load stabilizes.

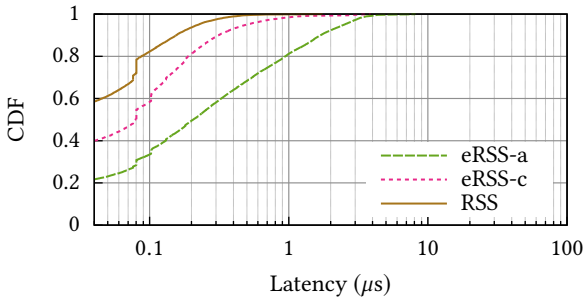

Figure 4: eRSS's 99th-percentile (tail) latencies, modeled as a function of queuing delay.

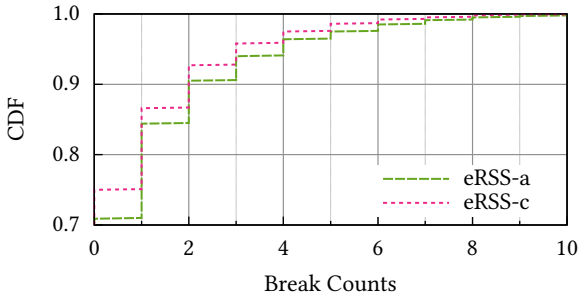

Figure 5: How often eRSS moves flows between cores. Baseline RSS does not move flows.

Tail latency. Modern cloud systems frequently have tail latency targets of 100 μs [15]: if an application is performing beyond its SLO, freeing up unused cores for background work would provide significant cost and energy savings. eRSS can be adapted to maximally conserve resources while *just barely* meeting latency constraints. Although eRSS uses fewer cores, and therefore adds tail latency, the overall impact on tail latency due to queuing (Figure 4) is bounded: our aggressive eRSS-a policy adds only 3.3 μs more latency, a tiny fraction of the total SLO. The conservative eRSS-c strategy also has a far smaller tail latency increase of 0.8 μs, demonstrating eRSS's adaptability.

Runtime characteristics. Finally, packet stealing can lead to poor performance due to reordering if flows are broken too often [11]. Figure 5 shows the number of breaks per flow for eRSS: 75% of flows are never broken, and over 90% of flows are broken at most two times. Throughout this evaluation, the periodic per-core weight updates are made using a large proportional feedback coefficient; this provides fast convergence of consistent hashing at the cost of more frequent updates. If we used a less aggressive strategy, or if there were fewer workload spikes, then flows would be broken less often—rapid changes in core counts require breaking flows to redistribute them.

5 SUMMARY & FUTURE WORK

With eRSS, we demonstrate that emerging NIC architectures with new abstractions (e.g., map-reduce) can enable complex scheduling techniques beyond just basic RSS. eRSS utilizes specialized hardware resources, like match-action stages and a map-reduce unit, to perform continuous workload estimation, core allocation, and packet load-balancing with infrequent corrections from software. The resulting system trades a minor increase in tail latency (while meeting SLOs) for significant gains in CPU efficiency. eRSS builds on prior work to achieve a line-rate packet and core scheduler that is work conserving, dispersion tolerant, and CPU efficient.

In-NIC machine learning. Advanced Machine Learning (ML) could be used to enhance some of eRSS's current estimators. For example, ML could predict per-packet execution time, replacing the current linear model. The NIC would then independently detect heavy-tailed flow distributions and ensure that sufficient cores are allocated, allowing for less over-provisioning and fewer NIC updates. This would provide shorter stabilization times and minimize short-term load imbalances.

Learned hyper-parameters. Another potential improvement to eRSS is using Reinforcement Learning (RL) to optimize eRSS's hyper-parameters, such as the amount of over-provisioning and the reaction time. These parameters are currently statically optimized for a single scaling group; a more dynamic system could take an SLO for each scaling group as its only parameter. RL would then explore the hyper-parameter space, finding the minimal amount of over-provisioning that meets the SLO. eRSS can thus learn custom policies that achieve better performance than general-purpose solutions.

ACKNOWLEDGMENTS

We thank members of the Pervasive Parallelism Lab, Neeraja Yadwadkar, and the anonymous APNet reviewers for their valuable feedback that helped improve the quality of this paper. This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7865. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government. This research is also supported in part by a Herbert Kunzel Stanford Graduate Fellowship and affiliate members and supporters of the Stanford DAWN project: Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, Teradata, SAP and VMware.

REFERENCES

[1] Memcached. https://memcached.org. Accessed on 04/30/2019.

[2] BlueField SmartNIC Ethernet. https://www.mellanox.com/products/smartnic/, 2018. Accessed on 04/30/2019.

[3] Agilent Technologies. Mixed Packet Size Throughput. https://s3.amazonaws.com/zanran_storage/www.ixiacom.com/ContentPages/109218067.pdf.

[4] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI* (2014).

[5] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM* (2013).

[6] Dean, J., and Barroso, L. A. The Tail at Scale. *Communications of the ACM* (Feb. 2013).

[7] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI* (2018).

[8] Ibanez, S., Brebner, G., McKeown, N., and Zilberman, N. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *ACM/SIGDA FPGA* (2019).

[9] Jeong, E. Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *USENIX NSDI* (2014).

[10] Jurkiewicz, P., Rzym, G., and Boryło, P. Flow Length and Size Distributions in Campus Internet Traffic. *arXiv:1809.03486* (2018).

[11] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive Scheduling for μsecond-Scale Tail Latency. In *USENIX NSDI* (2019).

[12] Maas, M., Harris, T., Asanovic, K., and Kubiatowicz, J. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *USENIX HOTOS* (2015).

[13] Molnar, I. Modular Scheduler Core and Completely Fair Scheduler (CFS). https://lwn.net/Articles/230501/. Accessed on 07/7/2019.

[14] Narayanan, D., Santhanam, K., Phanishayee, A., and Zaharia, M. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NIPS* (2018).

[15] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI* (2019).

[16] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud Storage System. *ACM TOCS* (Aug. 2015).

[17] Prekas, G., Kogias, M., and Bugnion, E. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *ACM SOSP* (2017).

[18] Schindelhauer, C., and Schomaker, G. Weighted Distributed Hash Tables. In *ACM SPAA* (2005).

[19] Swamy, T., Rucker, A., Shahbaz, M., Yadwadkar, N., Zhang, Y., and Olukotun, K. Taurus: An Intelligent Data Plane. https://p4.org/assets/P4WS_2019/p4workshop19-final19v2.pdf. Accessed on 05/1/2019.

[20] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy Transactions in Multicore In-Memory Databases. In *ACM SOSP* (2013).

[21] Wierman, A., and Zwart, B. Is Tail-Optimal Scheduling Possible? *Operations Research* (Sept. 2012).