Revisiting the Classics: Online RL in the Programmable Dataplane

Kyle A. Simpson [0000-0001-8068-9909], Dimitrios P. Pezaros [0000-0003-0939-378X]
University of Glasgow, Glasgow, Scotland
k.simpson.1@research.gla.ac.uk, Dimitrios.Pezaros@Glasgow.ac.uk

***Abstract*—Data-driven networking is becoming more capable and widely researched, partly driven by the efficacy of *Deep Reinforcement Learning* (DRL) algorithms. Yet the complexity of both DRL inference and learning force these tasks to be pushed *away from the dataplane* to hosts, harming latency-sensitive applications. Online learning of such policies cannot occur in the dataplane, despite being useful techniques when problems evolve or are hard to model.**

We present *OPaL*—On Path Learning—the first work to bring *online reinforcement learning* to the dataplane. OPaL makes online learning possible in constrained SmartNIC hardware by returning to classical RL techniques—avoiding neural networks. Our design allows weak yet highly parallel SmartNIC NPUs to be competitive against commodity x86 hosts, despite having fewer features and slower cores. Compared to hosts, we achieve a 21× reduction in 99.99^{th} tail inference times to 34 µs, and 9.9× improvement in online throughput for real-world policy designs. In-NIC execution eliminates PCIe transfers, and our asynchronous compute model ensures minimal impact on traffic carried by a co-hosted P4 dataplane. OPaL's design scales with additional resources at compile-time to improve upon both decision latency and throughput, and is quickly reconfigurable at runtime compared to reinstalling device firmware.

I. Introduction

Automatic network optimisation, control, and defence are at last becoming commonplace. Adaptive techniques such as *reinforcement learning* (RL) have led the charge in data-driven networking, enhancing automatic traffic optimisation [5, 27], congestion control [40], adaptive routing [12, 48], resource management [26], and packet classification [22]. In RL methods, every change and its effects improve future decisions.

In parallel, P4 [3] and *programmable dataplane* (PDP) hardware [17, 30, 52, 55] have inspired explosive growth and interest in the research community surrounding in-network computation and offloading. The promise of PDP hardware is that we can move the entire monitoring and analysis stack *into the dataplane itself, and have it evolve to incorporate new approaches*. The P4 ecosystem already presents novel, openly-available, and fine-grained traffic measurement techniques [6, 11, 13], and its control plane makes it easy to select which flows or packets are monitored in a live network. As a result, there has been keen interest in executing ML in the dataplane [19, 34, 38, 39, 44, 53] to take advantage of flow or per-packet state that cannot be efficiently processed or extracted at any other location. These works have shown the value of in-network ML: high-throughput, low latency response to network changes. While they can exploit on-device state to provide reactive insight, the missing piece of the puzzle is learning and updating these ML analyses online without deferring to another machine in the network. Training these models online and in-network is an exciting (and challenging) lacuna in the field that *has yet to be addressed*.

It is important to make this feasible; offloading to commodity hosts adds PCIe delays [31, 39], but is required due to the complexity of modern ML. For context, deep neural network training relies on backpropagation, can take vast amounts of offline simulation [2], and needs many minibatches for stability. These induce high costs for compute, storage, or dedicated accelerators to overcome the batching needed to operate at line-rate—while BrainWave [10] can reduce batching (and thus tail latencies) by 32×, inference still takes $O(\text{ms})$ [9]. Moreover, novel DMA techniques such as *GPUDirect* [32] halve but do not eliminate such PCIe transfers. The high latencies *caused* by steering and inference harm learning [47] and runtime application performance [40]. If we can bring online learning *to the dataplane*, then we can take advantage of rich, local state while minimising this impact on the learned policy. This would also make it easy to train and prototype agent designs which can learn *as the environment evolves*, or when there is too little data to model and simulate a problem.

In this paper, we enable *online in-NIC learning* by returning to *classical* RL methods. In particular, we focus on tile coding with algorithms such as Sarsa [43]. While these functions have lower model capacity, they do not require batches of inputs to learn in a stable way, negating the memory needed to store experience replays, and have simple update and inference logic. In addition, they have shown promise in other network use cases [25, 37]. Using fixed-point arithmetic, we solve the lack of floating-point support in PDP hardware *and* enable new optimisations. Moreover, the P4 dataplane can offer runtime control over which flows are monitored. We also design our solution to operate close to the P4 pipeline to access per-packet state, but outside of the main packet path to prevent packet stalls.

We show that online RL can be brought to the dataplane by such methods, where it can act on locally extracted state and is made *more efficient by dataplane hardware*. In particular, we exploit how SmartNIC devices often expose general-purpose compute [23] to provide path-adjacent, on-chip RL in the dataplane (fig. 1). As many of these devices have engineering and development histories which predate P4, general compute beyond P4's limits [46] is surprisingly common. By executing on spare compute units, we prevent packet stalling and offer quick runtime reconfigurability. This paper contributes:

- *OPaL*: a general-purpose in-NIC RL agent which scales with allocated device resources to meet latency or throughput demands of network control (§III),

- *ParSa*, a wait-free, parallel RL algorithm to accelerate tile coded policy inference and updates (algorithm 1),
- In-depth evaluation of how OPaL affects carried dataplane traffic, performs under different policy sizes, and improves on host deployment with a 15× latency reduction compared to commodity hardware (21× for 99.99$^{\text{th}}$ tail latencies) and an order of magnitude improvement in online throughput (§IV).
- A description of how OPaL would integrate with state-of-the-art PDP applications to perform fully in-NIC, automated DDoS mitigation (§V).

II. Preliminaries: towards in-NIC RL

A. Programmable hardware capabilities

While P4 [3] has led to great interest in network programmability, it requires similar behaviour between device classes—this is encoded by the *Programmable Switch Architecture* (PSA) [46]. However, many compatible devices long predate these developments. Many-core SoC-based Netronome [30], NetFPGA SUME [16, 55], and other SmartNICs [33, 52] allow arbitrary programs to be specified and executed. Currently, low port-density devices like these are most likely to have general-purpose compute and high degrees of parallelism [23], as they are designed for high-performance offloading and middlebox development.

B. Reinforcement Learning

Reinforcement learning (RL) trains an *agent* to choose an optimal sequence of *actions* from any *state* in pursuit of a given task [43]. Like most ML methods RL uses gradients to update the parameters used to approximate a function, aided by *reward measurements* from the environment. RL's MDP structure allows online learning of a state-action map in a model-free way, and can step through local minima when needed compared to ML. In networking, this allows for learning from on-device state or handling rapidly evolving problems. RL algorithms are agnostic to the policy approximation used so long as they are differentiable, and are computationally simple. E.g., Sarsa [43, pp. 217–221] requires only additions and multiplication to learn a policy online.

C. Tile-coded policy approximation

Returning to classical RL, we examine the linear function approximation of *tile coding* [43, pp. 217–221]. A policy is represented by sets of tiles, each covering one or more dimensions of the input state with several overlapping tilings (offset stepwise to provide generalisation). A state vector produces a single 'hit' in each tiling, all of which then correspond to a list of action preference lists—where computing a hit requires division by a known tile width. Inference is simply summing over all such lists to obtain a final preference list, selecting the highest-value action. Learning uses this same list after the next decision is made, adjusting the value of the previous action using a *temporal-difference value* (δ_t) computed by Sarsa. Crucially, this internal representation has no data dependencies between any tile calculations for an input.

III. Design and Implementation

We present our design for an in-NIC, task-independent, online RL system—*OPaL (On Path Learning)*. At a high level, OPaL

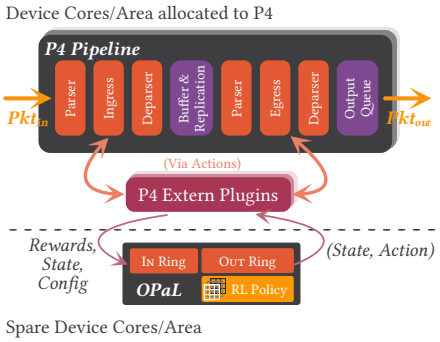


Figure 1. OPaL brings low-latency, online reinforcement learning directly to the dataplane. SoC- and NetFPGA-based SmartNIC devices expose spare compute—making in-situ, asynchronous processing and learning possible alongside P4 dataplanes. Classical RL policy methods are the key to making this feasible.

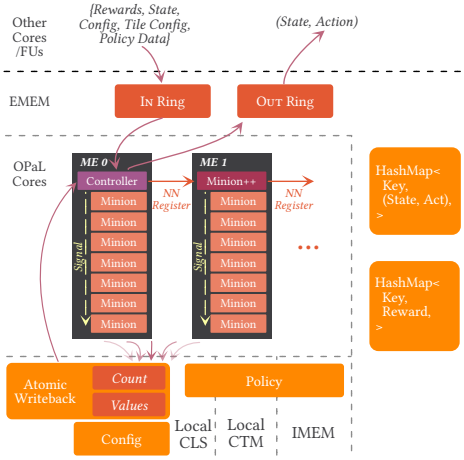


Figure 2. OPaL scales to fit device capacity according to either latency or throughput needs—*CoOp* is the optimal online design. A single *controller* delegates RL inference and updates to many *minions*, who complete independent subtasks.

uses auxiliary compute exposed by SmartNIC devices to offer low-latency online learning, scaling with on-chip resources at build time. As the allocation of cores/chip area is set ahead of time by a framework or system administrator, OPaL(-*CoOp*) agents enumerate themselves at runtime, during initialisation.

Figure 1 outlines our design on Netronome SmartNIC hardware in pursuit of this goal: unused device resources beyond the P4-PSA spec are used to drive asynchronous environmental control. We explain relevant NFP architectural details later in §IV-A. OPaL communicates with the packet pipeline of a P4 dataplane via `extern` plugins using In (state, configuration) and Out (action) messages (fig. 1). Internally, OPaL either has all its cores act independently, or cooperate to solve each task (fig. 2)—with different latency-throughput benefits. We open-source our firmware and control programs for the benefit of the community [36].

A. Challenges, solutions, and insights

Network and execution latency: State measurement, policy inference, and action installation take time, including network latency and serialisation to MAT-friendly formats [45]. In the midst of these tasks, system state evolves over time—adding noise to the state-action mapping being learned, harming learning and accuracy [47]. To solve this, we *co-locate all RL agent functions on PDP hardware*.

Lack of floating-point: PDP hardware, being designed solely

for efficient packet processing, lacks floating-point arithmetic support even in more general purpose NPU SmartNICs. Luckily, the embedded ML literature offers many low-precision floating-point formats [24, 42, 49, 51] and fixed-point representations [35, 54]. The latter set requires only integer arithmetic, which allows us to express and update policy parameters in-NIC. Thus, we use *quantised fixed-point (Qm.n) representations of action values*.

Costs of *online* function approximation: NIC-suitable inference schemes must still be trained offline, even though their data formats eschew floating-point—existing PDP ML converts a *pre-trained* model into a suitable representation, such as a binary neural network or chain of MATs. Additionally, DNN training relies on backpropagation, can require memorizing a sizeable replay buffer for RL, and needs many mini-batches of data for stable training. To solve this, we employ techniques from more classic RL literature: *tile-coded policies*, which have trivial gradients, and *one-step temporal-difference RL algorithms such as Sarsa [43]*.

Insights: We observe that tile coded inference is a map-reduce problem, where each tile hit accesses separate data which are then aggregated together, enabling us to exploit SmartNICs' parallelism. Crucially, updates can be performed locklessly without aggregation. For instance, a policy with m tiling sets (each having a set of input dimensions), with n overlapping tilings then creates $m \times n$ separate tasks. What is key is that fixed-point numbers also enable the use of atomic arithmetic, and thus our scheme admits a novel *wait-free Sarsa* RL algorithm. Moreover, individual integer operations are cheaper than floating-point, and we may tweak policy memory cost at compile time by choosing the desired integer size.

B. System Model

OPaL is a task-independent framework for in-network, online training and execution of *any RL agent design* using classical methods. OPaL is agnostic to the meaning of state vectors it receives as inputs and the actions it produces, which are employed by other functional units or the dataplane. However, in-NIC/in-network execution specifically benefits packet-, flow-, and network-level control tasks.

OPaL runs on one or more cores of a SmartNIC to convert fixed-point state measurements from the environment into a stream of actions using a stored policy. As an example, this might map flow performance measurements into queue priorities, or to compute and apply a rate limit to preserve quality of service. These dedicated cores process requests, compute actions, and update the policy in real time using reward measures. This policy can be trained from scratch entirely on the NIC, acting as a fully online RL agent. An input state vector *always* induces an action, and may update the policy using either an included reward, or one retrieved from memory based on the input state. This allows for simultaneous control and learning of independent systems by the same agent (i.e., optimising several flows with their own reward measures, such as DDoS mitigation in an AS where each next-hop AS might have their own 'health' metric). To protect traffic throughput and allow effective deployment in as many environments as possible, OPaL places RL execution on-chip, *but off the main packet path*, communicating and running parallel to the main P4 dataplane. As shown in fig. 1, this asynchrony allows coexistence with P4 programs, and imposes minimal impact on carried traffic for both bump-in-the-wire deployments and at end-points. For instance, the default P4 pipeline on Netronome has several cores go unused (similarly to spare area on an FPGA design), making this paradigm possible.

C. Action and Update Computation

OPaL applies the insights of Travnik *et al*. [47] to minimise action latency; an action is computed, sent out into the environment, and only then is the policy updated. Using one of the below strategies, a state vector is tile coded, converted into action probabilities, and an action is chosen. This is then written out to the environment as in §III-D. If online learning is enabled, OPaL then checks an internal hashmap for a prior state-action pair matching the current trace, and if this is found then the policy is updated. Updates are computed using *single-step semi-gradient Sarsa* [43, pp. 217–221], though modification to support other single-step methods would be trivial. The new state-action pair is then written into storage. OPaL can be configured to select values of the input state vector as keys for state and reward storage. Two firmware models govern how these tasks are carried out:

***CoOp* (fig. 2 and algorithm 1)** Threads cooperate to process state vectors, minimising latency. *Minion* threads have a fixed list of work items, while a *controller* thread sends compute/update commands before awaiting worker completion. Work items are disjoint, requiring no policy locks. *State vectors* are stored for update computation.

***Ind* (fig. omitted)** Separate threads listen for new states, and each works sequentially. Computing an action list requires a *read lock* on the policy. If an update occurs, the core requests a *write lock* before updating, limiting online throughput. *Tile lists* are stored for update computation.

Each offers a different point of optimisation; *Ind* maximises throughput if updates are disabled, while *CoOp* is designed to minimise decision latency and needs no locks to update the policy (increasing *online learning* throughput). These correspond to raw inference and learning performance, respectively. Latency and throughput have different effects on RL agents according to their target problem. Higher RL throughput is a necessity for per-flow/packet applications, which can require high decision rates even after combining state measurements, such as flow control or DDoS prevention. Equally, lower latency affords an agent finer-grained control and learning of a problem, able to react sooner to new information (e.g., in a routing optimisation problem).

D. Agent-Environment Communication

OPaL uses *multiple-producer/multiple-consumer* (MPMC) channels to communicate with other elements on the NIC; be they P4 programs on the packet path, or other on-chip analysis modules. Through these, a system *pushes* state vectors, reward measures, and setup packets as inputs, and *pulls* a stream of state-action pairs as outputs. This allows decisions to be made asynchronously, preventing packet stalling. As such, OPaL can receive input from P4 `externs` or other, dedicated off-path flow measurement applications in the same manner. We use platform-specific IPC

(EMEM ring buffers) to achieve this, using a shared freelist of buffers for packet payloads. Each message takes a median 126–140 ns communication time (local–remote), comparable to message channels in Rust and Go on commodity hardware.

E. Intra-Agent Communication

Optimising for latency requires meticulous care in how work is passed out and aggregated. This is truer still when moving from the moderately fine-grained control of classical RL (~1 ms) to its logical limit (tens of µs). Marshalling and data mutexes incur significant overheads, but on-chip execution and the ParSa algorithm allow us to sidestep these via lockless atomic aggregation. Moreover, adjacent cores often have special-purpose shared registers or share a small fast cache to accelerate communication. Our implementation exploits the locality of cores in the NFP. Policy tasks are passed between cores using direct *next neighbour* registers, signalling all child threads in response. Such on-chip signals cost just ~20 ns per relayed message. This can be factored into the design of additional async off-path functional units (similar to Li *et al*. [20]) on platforms like NetFPGA. To aggregate, each core performs atomic adds to a shared preference list and an acknowledgement counter checked by the master thread, implementing our wait-free *ParSa* algorithm (algorithm 1).

F. Reconfigurability

OPaL allows policy design and parameters to be changed at runtime using at most two control packets. Design changes are used at the end of learning (moving from online to offline), or when trying to train a new policy for another task from the same vantage point. Parameter changes allow an online agent to become more (or less) adaptive to new data (i.e., after detecting a changepoint in traffic). This extends to policy data, which may be imported from a pre-trained model and exported via PCIe to the host machine. Some aspects must be chosen at compile time; bit depth, *CoOp*/*Ind*, and maximum policy sizes. Choosing a bit depth of 16 bit or 8 bit halves/quarters policy memory costs, allowing complex problems to be modelled using more dimensions or fine-grained tiles. In our implementation, configuration packets are carried over UDP and signalled to the dataplane using a reserved DSCP value [1, 20]. While this simplifies parser generation, it also allows for configuration to be received from only trusted hosts (over the dataplane if needed) via P4 rules. Our control library and evaluation frameworks are written in Rust.

G. Work Allocation

Due to the lack of dynamic memory allocation on PDP hardware, and to simplify value lookups, policies cannot be stored sparsely. Tiling space requirements then scale exponentially with dimension count, so higher-dimension tilings must be placed in larger, slower memory regions. As a result policy parameters are split across such regions, giving different access and compute costs to different tasks. We use a simple first-fit scheduling algorithm run in OPaL, placing the largest work item into the least loaded thread of the least loaded core. Each work item is a separate *tiling* over a list of dimensions. The cost of any work item (from its dimension count and memory location) was empirically measured offline, and we weigh the total cost per core based on the number of minion threads available. This weighting specifically accounts for the controller thread on the first core. Work allocations are recomputed when policy configuration is installed or changed. Naturally, for n tilings and m threads this procedure is $O(n \log m)$: two find/update min operations into binary heaps per tiling, storing $m/8$ and ≤ 8 costs respectively. Although we omit relevant plots for space, we observe that this offers 1.33× and 1.11× speedup over naïve and stride-modulo schedules.

Algorithm 1: ParSa—*Par*allel *Sa*rsa

```
/* Given message passing
   mechanisms scatter and recv, quantised
   arithmetic functions Q_mul and TileCode, and
   omitting schedule/config/precache updates. */
/* cfg.α, cfg.γ are hyperparameters affecting
   the significance of each update and the
   degree of forward-planning, respectively.  */
enum Par { Act(state), Upd(delta, action, state) };
const cfg, policy = /* ... */;
let values: [AtomicI32; cfg.n_actions] = {0};
let acks: AtomicI32 = 0;
fn ParSa id, schedule
    if id==0 then
        forall state_pkt in IN do
            Ctl(state_pkt);
    else
        while true do
            Minion(schedule[id−1], recv());
fn Ctl state
    values, acks = {0}, scatter(Par::Act(state));
    acquire slot for OUT, copy state into slot;
    await acks == cfg.n_minions;
    let action = argmax(values);
    write action into OUT slot, enqueue;
    if cfg.online then
        let ((l_state,
          l_act, l_val), found_s) = cfg.lookup_state_from_key(state);
        let (reward, found_r) = cfg.lookup_reward_from_key(state);
        if found_s && found_r then
            let δ_t = reward+Q_mul(cfg.γ, values[action]) − l_val;
            δ_t = Q_mul(cfg.α, δ_t);
            acks = 0, scatter(Par::Upd(δ_t, l_act, l_state));
            await acks == cfg.n_minions;
        cfg.store_state(state, action, values[action]);
fn Minion tasks, msg
    switch msg do
        case Par::Act(s) do
            forall task in tasks do
                let hit = TileCode(s, task);
                for i in [0..cfg.n_actions) do
                    values[i].atomic_add(policy[hit][i]);
        case Par::Upd(δ, a, s) do
            forall task in tasks do
                let hit = TileCode(s, task);
                policy[hit][a] += δ;
    acks.atomic_add(1);
```

H. Limitations

Direct rule installation into P4 tables from the SmartNIC is not generally possible. To achieve line-rate performance, platforms like NFP use accelerated datastructures (e.g., DCFL [45]) computed over the *entire rule set*. Even through `externs`,

directly adding new rules is neither feasible nor safe. We instead suggest that `externs` or datapath stages which apply RL actions to packets should maintain a small store of state-action pairs, and periodically send these back to the controller for batch installation. Parallelisation also adds per-task overheads which require a minimum number of workers to improve on a serial approach—we measure this crossover point in §IV-D

IV. Evaluation

We investigate the performance of OPaL compared to classical RL techniques executed on commodity hosts, with *CoOp* offering a 15–21 × speedup in median–99.99$^{\text{th}}$ state-action latency and 9.9 × greater online learning throughput. Crucially, in-NIC execution offers tight tail latency bounds compared to host-based approaches. We report on how OPaL scales as additional device resources are added, noting that both in-NIC designs outperform commodity hosts *using just one core* in latency and online throughput. Furthermore, *Ind* provides higher per-core offline throughput than host-based approaches, even though our measured hosts exhibit higher clock speeds. Finally, we show that OPaL has minimal impact on dataplane cross-traffic carried by its parent device.

A. Netronome Platform Fundamentals

NFPs achieve scalable packet processing through parallelism. The chip is composed of *microengines* (MEs), grouped into *islands* of 4 or 12 MEs. Each ME has 4–8 *contexts* (threads) which share a code store. Beyond registers, the platform has an explicit memory hierarchy in size and access cost: LMEM (ME) < CLS (Island) < CTM < IMEM (Chip) < EMEM.

B. Experimental Setup

Testing machines were as follows, with 32 GiB RAM:

MidServer Intel Xeon Bronze 3204 (6×1.9 GHz),
HighServer Intel Xeon Silver 4208 (8×2.1 GHz),
Collector Intel Core i7-6700K (4×4.2 GHz).

OPaL was evaluated on server blades (*Mid/HighServer*), each hosting a single Netronome Agilio LX 40GbE (NFP-6480, 1.2 GHz). These servers ran Ubuntu 18.04.5 LTS (4.15.0-140-generic). We additionally use a more powerful consumer-grade machine (*Collector*) for estimating host performance when offloaded to a network function, running Ubuntu 18.04.4 LTS (4.15.0-96-generic). Host execution occurs on the CPU using a numpy-based Sarsa implementation. Control programs were built using rustc 1.52.1. We run OPaL on a 4-ME island of the NFP-6480 (32 contexts) using 32 bit, 16 bit and 8 bit arithmetic. This is the largest cluster of cores which is not in use by a P4 pipeline. All OPaL timing measurements were repeated over 10 000 state packets (preceded by 1000 warmup packets), retrieving item processing times over PCIe from which throughput was derived. Host throughput and latency measures were observed over 10 trials of 10 s (with 5 s warmup/cooldown times). We differ this from the NFP as hosts need to run numpy-based agents in parallel as separate processes; this also allows us to investigate the effects of oversubscription. Policy sizes are set to those of a real-world DDoS control application [37]: 20-dim state vectors, a bias tile and 16 full tiling sets (7×1-dim, 8×2-dim, 1×4-dim), 8 tilings per set, 6 tiles per dimension, and 10 actions. For context, such input would contain per-flow state (e.g., IATs, rates) combined with the last action taken (2-dim tilings) and loads along the ingress-egress path (4-dim). In *CoOp*, this creates 129 tasks across 31 workers. We choose a larger action count to investigate the performance of more complex agents.

C. Experiments

Inference and learning: We compare how long it takes for OPaL to compute actions and policy updates, and report on its throughput against a floating point (numpy-based) implementation of Sarsa on commodity hosts. This lets us demonstrate the performance differences between *Ind* and *CoOp*, particularly in how *Ind*'s (and hosts') required policy locks impact throughput. We compare online learning performance with offline in these cases. Online performance marks the number of decisions that can be made per second (and associated latency) when training a policy. Offline performance is crucial for pushing a trained, known-good policy to agents with an expected higher raw decision throughput. State-action latency is a shared property of both cases; the main impact on throughput arises from the update step.

We then vary the amount of worker threads to show how OPaL scales to fit available compute resources on a device. This is important for planning in an intelligent dataplane—particularly when cohabiting with other dataplane programs—and has effects on ahead-of-time work scheduling which we examine later. This also demonstrates the number of cores needed to achieve a given latency or throughput bound on a real-world policy. Moreover, to demonstrate how these costs vary in larger policies, we vary the total number of dimensions in each tiling.
End-to-end RL latency: We compare the key RL latencies we discuss in §III-A across 3 scenarios: completely in-NIC (OPaL), offloading RL decisions to a SmartNIC's controller machine, and offloading to a *virtual Network Function* (vNF).
Coexistence with the dataplane: While varying the rate of RL updates performed by *CoOp* (32 bit) from 0–16 000 actions/s, we measure packet loss and latencies of cross traffic carried over a co-hosted P4 pipeline. This allows us to quantify whether on-chip (out-of-path) execution impacts ordinary dataplane behaviour indirectly: e.g., EMEM cache evictions or hidden resource contention. We test an NFP in *MidServer* using Pktgen-DPDK [50], connecting *HighServer* as the traffic source over a 40 Gbit/s cable. We perform loss tests using 7/1 Tx/Rx queues at 100 % send rate for 10 bursts of 30 s, and perform latency tests using 1/1 Tx/Rx queue at 10 % send rate for 200 000 measurements (sampling at 2000 Hz for 10×10 s). This maximises throughput in the former case, relying on NIC counters for loss detection. The latter minimises host resource contention to observe accurate latencies, observe enough samples to detect subtle (aggregate) latency effects, and eliminate *host* receive drops. DPDK was setup using 4×1 GiB hugepages. Sent traffic comprised fixed-size 64–1518 B packets [4]. CPU clock scaling was disabled on *HighServer*.
Resource requirements: Using the policy size defined above, we investigate how the memory requirements imposed by OPaL vary with the number of dedicated MEs, over and above a base P4 forwarding plane. We report resource use for 32 bit *Ind* and

Table I
LATENCIES AND COMPUTATION TIMES FOR OPAL VERSUS COMMODITY HOSTS. ON-DEVICE EXECUTION IS CRUCIAL IN LOWERING LATENCIES *and* REDUCING TAIL LATENCIES. LOWER IS BETTER, WITH THE BEST MARKED *in bold*.

Datatype	Machine/FW	State-Action Latency (µs)			State-Update Time (µs)		
		Median	99th	99.99th	Median	99th	99.99th
Float	Collector	515.94	606.06	725.03	606.06	636.82	833.99
	MidServer	1069.07	1125.1	1508.0	1260.04	1605.99	1719.864
Int32	OPaL-*Ind*	185.133	185.533	186.213	230.840	231.347	232.227
	OPaL-*CoOp*	**34.347**	**34.520**	**34.573**	**62.000**	**62.440**	**63.120**

Table II
ACTION AND UPDATE THROUGHPUTS FOR OPAL VERSUS COMMODITY HOSTS. MOST DESIGNS CANNOT SCALE ONLINE PERFORMANCE WITH ADDITIONAL CORES. HIGHER IS BETTER, WITH THE BEST MARKED *in bold*.

Datatype	Machine/FW	Workers	Throughput (k actions/s)		Throughput/core (k actions/s)	
			Offline	Online	Offline	Online
Float	Collector	4	7.673(49)	1.627(31)	1.918(12)	—
	MidServer	6	5.584(30)	0.791(12)	0.931(5)	—
Int32	OPaL-*Ind*	32	**172.875(229)**	4.333(5)	**5.402(7)**	—
	OPaL-*CoOp*	32	29.166(173)	**16.141(73)**	0.911(5)	**0.504(2)**

CoOp agents, with hash tables sized to 4096 state-action pairs and 16 separate reward values. This captures the relative cardinality of network RL traces to rewards; many input flows will map to one or few reward values (i.e., DDoS attack size estimation per egress-AS, queue occupancy per output port in AQM).

Deployability: By timing agent setup and compile times, we describe the runtime costs needed for an administrator to repurpose an installed agent in a live network.

D. Results and Discussion

Inference and learning: Table I shows how OPaL compares in latency with a numpy-based RL policy.[1] When using 4 MEs of the NFP-6480, *CoOp* achieves sub-35 µs median latency, with 99th and 99.99th percentile latencies less than 1 µs worse (15× and 21× speedups over *Collector*). Importantly, *Ind* achieves lower median latencies (2.79×) *and* update times (2.63×) than a dedicated *Collector using only a single core or functional unit*. Crucially, in-NIC execution gives far tighter bounds on tail latency compared to host offloading. 99.99th percentile state-action latencies exceed the median by 0.58 % and 0.66 % for *Ind* and *CoOp*, while *host* tail latencies are at least 40.53 % greater. We show their cumulative distributions in detail (fig. 5), noting how just one additional CPU-intensive task—potentially automated system updates or another traffic processing task—impacts tail latencies further (*Float(Over)*).

Table II compares OPaL's throughput against hosts. We set the worker count on host machines equal to their number of physical cores—moving beyond this would hamper tail latencies by an order of magnitude. To make the comparison fair in the context of many-core CPU environments, we include per-core throughput. *Ind* achieves 2.82× higher offline throughput per core than commodity *Collector*, in spite of the NFP-6480 having a considerably slower clock speed (0.29×). Due to the *abundance* of such weaker chips, in-NIC RL is able to deliver much higher throughput. As anticipated, *CoOp* is key in achieving serviceable throughput in an online learning agent, 9.9× that of a dedicated collector machine.

[1]For brevity, we omit integer numpy results—against a float implementation, median action latencies are 14.6 % worse, with 7.9 % longer update times.

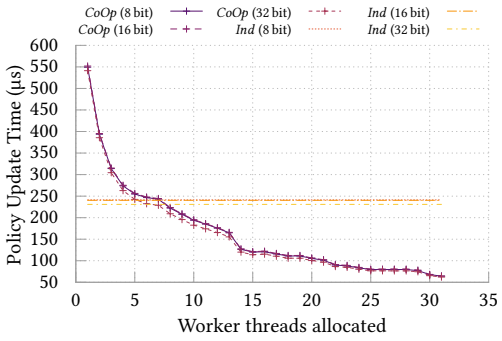


Figure 3. *CoOp*'s online learning performance improves with additional cores. This requires 8 workers to offer greater online throughput than single-threaded in-NIC RL. Sharper performance increases occur when a new physical core is added (7–8) or the scheduler works around a bottleneck (13–14).

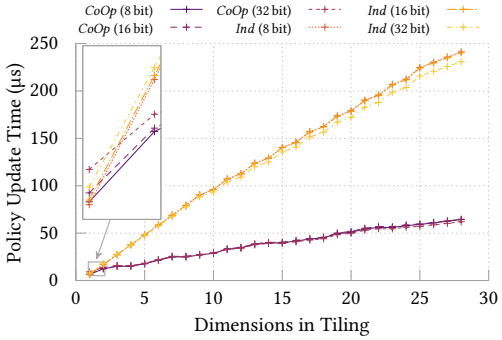
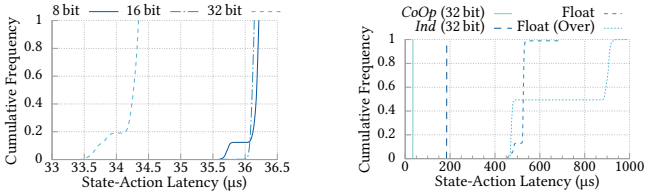


Figure 4. *CoOp* processes updates faster than *Ind*—thus has higher online performance—on almost all policy sizes. Lower bit depths are effective on simple policies. State-action latency scales similarly.



(a) OPaL's *CoOp* design achieves consistent, tight latency bounds.

(b) Tail latencies suffer in hosts—particularly when oversubscribed.

Figure 5. Cumulative state-action latency plots for OPaL versus hosts.

By limiting workers, we show how *CoOp*'s policy update time (thus online throughput—fig. 3) scales with available cores. While *CoOp* always outperforms hosts, we observe there are two distinct crossover points which must be met to overcome *Ind*; 8 workers for online throughput, and 3 for latency (plot omitted). Some artefacts of our environment and design are visible; the addition of new physical cores is more significant than contexts, and some schedule bottlenecks are visible. Most importantly, *CoOp*'s resource demand is tunable at compile time to meet the training rate or action latency required by a task. Figure 4 shows how policy complexity affects update cost, scaling from a bias tile up to the full DDoS policy size. *CoOp* always produces an action in less time than *Ind*, but requires at least one state-based tile to excel in online learning. We note that this is a trivial case, as using *only* a bias tile returns a single preference list regardless of input state.

8 bit and 16 bit agents underperform against 32 bit, except for smaller policies (zoomed portion of fig. 4)—even though OPaL is optimised to access policy data in batches. As the native register

width on the NFP is 32 bit, the compiler emits extra instructions around ALU operations to correctly load and store values. This explains what we see in larger policies, as higher dimension tilings require more arithmetic, and most of the I/O comes *after* computing each hit tile, causing ALU use to dominate. This also explains why the crossover point differs for online (fig. 4, 10 dims) and offline (plot omitted, 3 dims) agents: state management falls into the serial portion of the online algorithm. To overcome this, we investigated bit-stuffing values into a single word during writeback (as the platform offers both 32 bit and 64 bit atomic addition). This is analogous to SIMD through use of padding bits, but we found that manipulating tiles into the correct format added 10 % overhead.

End-to-end RL latency: To determine state-action latencies, we take inference times from table I for host and in-NIC processing, and add the packet RTT to the inference site:

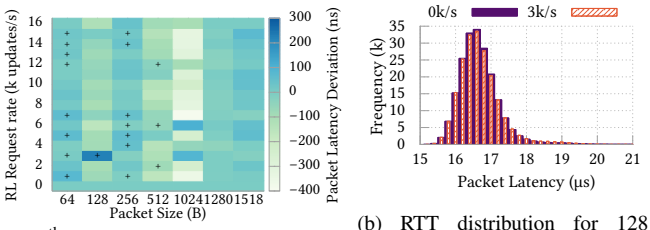
In-NIC. As described in §III-D, EMEM rings have a median one-way delay cross-island of 140 ns, giving a *median* 34.63 µs *end-to-end inference latency*.

Dedicated Collector. Employing DPDK, such hosts add one-way PCIe packet delays of 0.9–2.3 µs [31]. A UDP packet carrying 20 elements of state in OPaL is 128 B, so costs 1 µs to forward, and the reply state-action pair invokes a slightly higher cost. *This gives an end-to-end inference latency of* 517.9 µs.

vNF Offload. Cziva and Pezaros [7] show that more lightweight vNF frameworks like GNF [7] and ClickOS [28] add 45–55 µs *additional* RTT latency above PCIe costs. *This gives an end-to-end inference latency of* 572.9 µs.

Thus, in-NIC RL offers 14.96× and 16.54× lower latency over collector and vNF deployments respectively. We contrast these against deep RL for network tasks, which can take 3 ms [40]—2 orders of magnitude above OPaL with identically sized inputs.

Coexistence with the dataplane: Our setup met 40 Gbit/s for packet sizes ≥256 B. For frames of 64 B and 128 B, input traffic rates were 17.4 Gbit/s and 32.9 Gbit/s respectively (33.9 Mpps and 32.2 Mpps). Passing this traffic over the NFP device running OPaL, no packet losses occurred at any rate of RL actions. We show the effect of RL workloads on the RTTs of cross traffic via fig. 6. As observed RTTs were not normally distributed, we employed a one-tailed *Mann-Whitney U test*, marking population increases in latency ($p < 0.05$) with a "+". Statistically significant increases concentrate around smaller packet sizes; all (bar one) of these affected 99th percentile latencies by under 0.38 % (≤78 ns). This slight effect can be explained by increased pressure on the NFP's *Command Push-Pull* (CPP) bus, which handles cross-island accesses to memory and other resources. OPaL uses the CPP bus through its IN/OUT EMEM rings and last-tier policy accesses. This also explains the sensitivity of 256 B packets to OPaL—the NFP segments packets, storing metadata (e.g., MAC prepend) and the first bytes of a packet in a 256 B CTM block and parking their payloads in EMEM. 256 B packets overshoot this due to metadata, causing small I/O accesses at a high rate for packets sized around this cutoff. An anomaly is 128 B packets at 3000 RL updates per second, causing a 222 ns (1.18 %) increase. We believe the inbound



(a) 99th %tile cross-traffic RTT changes. (b) RTT distribution for 128 B packets at 0 and 3000 actions/s.

Figure 6. Effects on tail latency of cross-traffic caused by off-path RL compute. Statistically significant increases in population latency are concentrated on smaller packet sizes, and are typically sub-78 ns.

Table III
NFP MEMORY USE DUE TO OPAL USING 1 AND 4 MES (32 BIT). CLS AND CTM ARE SHARED BETWEEN ALL PROGRAMS ON THE SAME ISLAND, EMEM AND IMEM ARE SHARED BETWEEN ALL PROGRAMS ON A NIC.

Firmware	EMEM		EMEM Cache		IMEM		i5.CLS		i5.CTM	
	MiB	%	KiB	%	KiB	%	KiB	%	KiB	%
Base P4	6776.67	88.24	268.52	2.91	858.28	10.48	0.00	0.00	0.00	0.00
Ind(1)	6780.21	88.28	2541.08	27.57	1263.28	15.42	24.75	38.67	94.25	36.82
Ind(4)	6780.22	88.28	2545.33	27.62	1263.28	15.42	51.18	79.97	107.00	41.80
CoOp(1)	6779.12	88.27	1773.59	19.24	1263.28	15.42	22.41	35.01	90.00	35.16
CoOp(4)	6779.12	88.27	1769.84	19.20	1263.28	15.42	52.16	81.49	90.00	35.16

request rate is weakly synchronised with traffic, causing bursty accesses to the CPP bus. We expect that FPGA designs can avoid this by having dedicated IN/OUT mechanisms for an OPaL agent.

Resource requirements: Table III shows how OPaL consumes memory as it scales to additional cores, compared with a simple P4 forwarding application. As one program is installed per ME, these results represent the minimum and maximum resource use on a single island (i.e., without replacing P4 workers). We observe negligible costs on shared EMEM (~4 MiB) from state and reward hash tables. The most significant costs arise due to policy data (405 KiB shared IMEM, 90 KiB local CTM, 15 KiB local CLS), which can be halved or quartered using 16 bit and 8 bit quantisation. This is a high upfront cost on per-island resources (CLS/CTM)—OPaL leaves resources for other off-path dataplane applications, but is fairest from 3 cores onwards.

Deployability: Setup of OPaL uses two packet types: *setup*, which contains policy shape and learning parameters, and *tiling*, which provides a list of indices for tiling sets. Handling these packets took a mean 27.03 µs and 16.69 µs respectively on *Ind*, allowing an agent to be swapped from online to offline painlessly, i.e., after convergence. *CoOp* exhibits similar costs, however the scheduler causes policy *structure* changes to take 422.63 µs for a max-size policy—we found that this scales as described earlier ($O(n \log m)$). Policy *data* changes require no additional work, resolving purely to `memcpy`s. Firmware installation (i.e., changing *Ind* to *CoOp* or bit depth) took a mean 38.83 s. Compiling and linking OPaL and the P4 toolchain took 35 s, while changing only OPaL parameters required 25 s. These results show that OPaL can be easily adapted by network administrators once in place, and illustrates an advantage of SoC-based SmartNICs.

V. POTENTIAL INTEGRATIONS

A. In-Network DDoS Defence

Classical RL has seen recent use in real-time, adaptive DDoS mitigation [37]. Simpson *et al.*'s *Guarded* agent design uses a mixture of network and per-flow state to monitor how flows re-

spond to bandwidth changes and packet loss. Actions move flows up or down in punishment levels. To implement and improve upon this work using OPaL, we would place its RL agents on SmartNICs at AS edge nodes—a bump-in-the-wire deployment.

Inputs: Low-latency, pure-P4 solutions to extract and record per-flow TCP state directly in the dataplane such as Dapper [11] and Sonata [13] are well-studied. We propose placing such monitors in the P4 dataplane, existing on-chip alongside OPaL. The required global state (load measures from network paths) must still come from elsewhere in the network; this is now the element at highest risk of becoming stale, but the least likely to vary significantly in response to individual actions. We posit that INDDoS [8], which estimates DDoS victim cardinality, would be an effective reward function source.

Integrating OPaL: Before flow monitoring, this solution polls OPaL's OUT *Ring*—these actions would be placed into a local hash table *and* exported to the controller to be batch-inserted as P4 rules. Packet ingress timestamps would be used to emulate the TRS scheduler used by anti-DDoS agents for rate-controlled work, selecting state vectors for OPaL. The tight bounds on OPaL's execution time make it easy to calculate the maximum number of decisions which can be made per deadline. Reward values would then be separately inserted by a modified INDDoS table.

It can take at least one RTT for meaningful changes to occur in a flow's behaviour (O(ms) in a transit AS/ISP). Accordingly, this use case benefits most from an increase in *throughput* using *Ind*. Higher throughput means that network flows are more likely to be judged in *every* timestep—changes in flow behaviour are more likely to be acted on and learned from. Flows exceeding maximum throughput simply cause it take longer *in expectation* for a flow to be reassessed. As shown in §IV, OPaL far exceeds the throughput of host offloading, making in-NIC execution ideal. The control plane can then dynamically narrow down or expand the set of flows to be monitored.

B. Network Deployment Considerations

In a network, a subset of OPaL nodes could be *CoOp* agents, training online, while most other nodes run *Ind* to meet throughput guarantees. The control plane would then combine and distribute these improved policies between offline agents. This can be taken further, using policy deltas to enable transfer learning for more complex models such as neural networks.

VI. Related Work

In-network ML: Taurus [44] proposes efficient line-rate inference using a configurable grid of map-reduce units in the packet pipeline (implementing e.g., LSTMs and SVMs). On CGRA hardware, they achieve sub-µs extra latency. *IIsy* [53] shows how *classical ML inference* (SVMs, Naïve Bayes, etc.) can be converted into match-action tables compatible with *any* P4 deployment. They achieve mean 2.62 µs extra latency on NetFPGA.

A recent line of research is the use of *Binarised Neural Networks* (BNNs) [14, 18, 29] for line-rate packet classification. *BaNaNa SPLIT* [34, 38] shows this as an offload mechanism for fully-connected layers. In-network packet tagging and classification by pre-trained BNNs is shown by *N3IC* [39], achieving packet inference in 45 µs on the NFP, and 0.3 µs on NetFPGA for 256 bit inputs. Comparatively, OPaL-*CoOp* can process an identically-sized input in a median 13.83 µs. Our work handles larger inputs (640 bit) at lower latencies (34 µs), and offers online learning. We expect that a NetFPGA implementation of OPaL would enjoy a similar factor of speedup. Langlet [19] has shown the viability of NN inference using 64 bit quantisation on the NFP, using *in-path* compute rather than our asynchronous model. Inference latency on small networks can be as high as 500 µs on line rate traffic, emphasising the value of path-adjacent compute.

We stress that none of these approaches (or that we have seen) tackle *online learning and control* in-network—we believe OPaL has broken new ground in this regard.

In-network ML *acceleration*: Optimisation of distributed neural network training is an area where in-network compute has been key for general NNs [21] and RL-specific procedures [20], using NetFPGAs to implement floating point adders. In-NIC processing allows gradient packets to be aggregated *in-network*, overcoming incast behaviour and host bottlenecks.

RL for network control: *NeuroCuts* [22] uses deep RL to train an agent which can build efficient decision tree packet classifiers for use in constrained environments (e.g., network hardware). Deep RL techniques have been used for QUIC congestion control optimisation [40]. A key facet of this work is the need for asynchronous RL in networks, where pauses for DNN-based inference can significantly harm throughput.

PDP design for asynchronous compute: *PANIC* [41] places a routing fabric between distinct packet/data processing elements *in a SmartNIC*. Such designs would enable novel asynchronous compute in SmartNICs and switches, for instance offering consistent and easy to use communication between workers versus hard-coded ME relationships. Event-driven versions of P4 have been suggested [15]. Timer events and device state changes would empower in-network RL use cases, signalling timesteps for RL agents or new, effective, fine-grained sources of input state.

VII. Conclusion

We have presented *OPaL*, bringing *online RL* to the dataplane. In-NIC use of classical RL algorithms makes this possible, and enables significant reductions in median–99.99$^{\text{th}}$ inference times and order of magnitude improvements in online learning throughput, with minimal impact on dataplane cross-traffic. In future, we aim to examine the performance of individual applications driven by OPaL—both classical and deep RL-based—and how a NetFPGA implementation can offer further latency and throughput improvements. A promising avenue here would be to investigate constant transfer learning between online OPaL agents and high-throughput offline function approximators such as BNNs.

Acknowledgements: The authors would like to thank Rhys Simpson for his comments and discussions on the soundness of SIMD-like optimisations. They would additionally like to thank Stefanos Sagkriotis, Mircea Iordache-Şică, and Haruna Adoga for their comments and feedback. We thank the anonymous reviewers of SOSR'21 and NOMS'22 for their comments. This work was supported in part by the Engineering and Physical Sciences Research Council [grants EP/N509668/1, EP/N033957/1].

References

[1] Fred Baker, David L. Black, Kathleen Nichols and Steven L. Blake. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. Dec. 1998. DOI: 10.17487/RFC2474. URL: https://rfc-editor.org/rfc/rfc2474.txt.

[2] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski and Susan Zhang. 'Dota 2 with Large Scale Deep Reinforcement Learning'. In: *CoRR* abs/1912.06680 (2019). arXiv: 1912.06680.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese and David Walker. 'P4: programming protocol-independent packet processors'. In: *Computer Communication Review* 44.3 (2014), pp. 87–95. DOI: 10.1145/2656877.2656890.

[4] Scott Bradner and Jim McQuaid. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544. Mar. 1999. DOI: 10.17487/RFC2544. URL: https://rfc-editor.org/rfc/rfc2544.txt.

[5] Li Chen, Justinas Lingys, Kai Chen and Feng Liu. 'AuTO: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization'. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. Ed. by Sergey Gorinsky and János Tapolcai. ACM, 2018, pp. 191–205. DOI: 10.1145/3230543.3230551.

[6] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford and Ori Rottenstreich. 'Catching the Microburst Culprits with Snappy'. In: *Proceedings of the Afternoon Workshop on Self-Driving Networks, SelfDN@SIGCOMM 2018, Budapest, Hungary, August 24, 2018*. ACM, 2018, pp. 22–28. DOI: 10.1145/3229584.3229586.

[7] Richard Cziva and Dimitrios P. Pezaros. 'Container Network Functions: Bringing NFV to the Network Edge'. In: *IEEE Commun. Mag.* 55.6 (2017), pp. 24–31. DOI: 10.1109/MCOM.2017.1601039.

[8] Damu Ding, Marco Savi, Federico Pederzolli, Mauro Campanella and Domenico Siracusa. 'In-Network Volumetric DDoS Victim Identification Using Programmable Commodity Switches'. In: *IEEE Trans. Network and Service Management* (2021). Early access.

[9] Javier Duarte, Philip Harris, Scott Hauck, Burt Holzman, Shih-Chieh Hsu, Sergo Jindariani, Suffian Khan, Benjamin Kreis, Brian Lee, Mia Liu, Vladimir Lončar, Jennifer Ngadiuba, Kevin Pedro, Brandon Perez, Maurizio Pierini, Dylan Rankin, Nhan Tran, Matthew Trahms, Aristeidis Tsaris, Colin Versteeg, Ted W. Way, Dustin Werran and Zhenbin Wu. 'FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing'. In: *Computing and Software for Big Science* 3.1 (Oct. 2019), p. 13. ISSN: 2510-2044. DOI: 10.1007/s41781-019-0027-2.

[10] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung and Doug Burger. 'A Configurable Cloud-Scale DNN Processor for Real-Time AI'. In: *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. Ed. by Murali Annavaram, Timothy Mark Pinkston and Babak Falsafi. IEEE Computer Society, 2018, pp. 1–14. DOI: 10.1109/ISCA.2018.00012.

[11] Mojgan Ghasemi, Theophilus Benson and Jennifer Rexford. 'Dapper: Data Plane Performance Diagnosis of TCP'. In: *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*. ACM, 2017, pp. 61–74. DOI: 10.1145/3050220.3050228.

[12] Tomer Gilad, Neta Rozen Schiff, Philip Brighten Godfrey, Costin Raiciu and Michael Schapira. 'MPCC: online learning multipath transport'. In: *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020*. Ed. by Dongsu Han and Anja Feldmann. ACM, 2020, pp. 121–135. DOI: 10.1145/3386367.3433030.

[13] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford and Walter Willinger. 'Sonata: query-driven streaming network telemetry'. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. Ed. by Sergey Gorinsky and János Tapolcai. ACM, 2018, pp. 357–371. DOI: 10.1145/3230543.3230555.

[14] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv and Yoshua Bengio. 'Binarized Neural Networks'. In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon and Roman Garnett. 2016, pp. 4107–4115.

[15] Stephen Ibanez, Gianni Antichi, Gordon J. Brebner and Nick McKeown. 'Event-Driven Packet Processing'. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 2019, pp. 133–140. ISBN: 978-1-4503-7020-2. DOI: 10.1145/3365609.3365848.

[16] Stephen Ibanez, Gordon J. Brebner, Nick McKeown and Noa Zilberman. 'The P4-¿NetFPGA Workflow for Line-Rate Packet Processing'. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*. Ed. by Kia Bazargan and

Stephen Neuendorffer. ACM, 2019, pp. 1–9. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293924.

[17] Intel. *Explore the Power of Intel Programmable Ethernet Switch Products*. Oct. 2020. URL: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html (visited on 01/02/2021).

[18] Minje Kim and Paris Smaragdis. 'Bitwise Neural Networks'. In: *CoRR* abs/1601.06071 (2016). arXiv: 1601.06071.

[19] Jonatan Langlet. 'Towards Machine Learning Inference in the Data Plane'. Bachelor's Thesis. Karlstad University, June 2019. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-72875 (visited on 04/05/2021).

[20] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander G. Schwing and Jian Huang. 'Accelerating distributed reinforcement learning with in-switch computing'. In: *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Srilatha Bobbie Manne, Hillery C. Hunter and Erik R. Altman. ACM, 2019, pp. 279–291. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322259.

[21] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander G. Schwing, Hadi Esmaeilzadeh and Nam Sung Kim. 'A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks'. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 175–188. ISBN: 978-1-5386-6240-3. DOI: 10.1109/MICRO.2018.00023.

[22] Eric Liang, Hang Zhu, Xin Jin and Ion Stoica. 'Neural packet classification'. In: *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. Ed. by Jianping Wu and Wendy Hall. ACM, 2019, pp. 256–269. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342221.

[23] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter and Karan Gupta. 'Offloading distributed applications onto smartNICs using iPipe'. In: *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. Ed. by Jianping Wu and Wendy Hall. ACM, 2019, pp. 318–333. DOI: 10.1145/3341302.3342079.

[24] Stefan Mach, Fabian Schuiki, Florian Zaruba and Luca Benini. 'FPnew: An Open-Source Multi-Format Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing'. In: *CoRR* abs/2007.01530 (2020). arXiv: 2007.01530.

[25] Kleanthis Malialis and Daniel Kudenko. 'Distributed response to network intrusions using multiagent reinforcement learning'. In: *Eng. Appl. of AI* 41 (2015), pp. 270–284. DOI: 10.1016/j.engappai.2015.01.013.

[26] Hongzi Mao, Mohammad Alizadeh, Ishai Menache and Srikanth Kandula. 'Resource Management with Deep Reinforcement Learning'. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*. Ed. by Bryan Ford, Alex C. Snoeren and Ellen W. Zegura. ACM, 2016, pp. 50–56. ISBN: 978-1-4503-4661-0. DOI: 10.1145/3005745.3005750.

[27] Hongzi Mao, Ravi Netravali and Mohammad Alizadeh. 'Neural Adaptive Video Streaming with Pensieve'. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 2017, pp. 197–210. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098843.

[28] João Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Andrei Olteanu, Michio Honda, Roberto Bifulco and Felipe Huici. 'ClickOS and the Art of Network Function Virtualization'. In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*. Ed. by Ratul Mahajan and Ion Stoica. USENIX Association, 2014, pp. 459–473.

[29] Daisuke Miyashita, Edward H. Lee and Boris Murmann. 'Convolutional Neural Networks using Logarithmic Data Representation'. In: *CoRR* abs/1603.01025 (2016). arXiv: 1603.01025.

[30] Netronome. *SmartNIC Overview*. 2021. URL: https://www.netronome.com/products/smartnic/overview/ (visited on 01/02/2021).

[31] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo and Andrew W. Moore. 'Understanding PCIe performance for end host networking'. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. Ed. by Sergey Gorinsky and János Tapolcai. ACM, 2018, pp. 327–341. DOI: 10.1145/3230543.3230560.

[32] NVIDIA. *GPUDirect*. Mar. 2021. URL: https://developer.nvidia.com/gpudirect (visited on 14/10/2021).

[33] NVIDIA. *NVIDIA BlueField Data Processing Units*. Apr. 2021. URL: https://www.nvidia.com/en-gb/networking/products/data-processing-unit/ (visited on 11/05/2021).

[34] Davide Sanvito, Giuseppe Siracusano and Roberto Bifulco. 'Can the Network be the AI Accelerator?' In: *Proceedings of the 2018 Morning Workshop on In-Network Computing, NetCompute@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*. Ed. by Xin Jin and Changhoon Kim. ACM, 2018, pp. 20–25. DOI: 10.1145/3229591.3229594.

[35] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports and Peter Richtárik. 'Scaling Distributed Machine Learning with In-Network Aggregation'. In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 785–808.

[36] Kyle A. Simpson. *FelixMcFelix/pdp-rl-paper. Quantifying*

the effects of performing some (or all) work in an RL system in the dataplane. Oct. 2021. URL: https://github.com/FelixMcFelix/pdp-rl-paper (visited on 18/10/2021).

[37] Kyle A. Simpson, Simon Rogers and Dimitrios P. Pezaros. 'Per-Host DDoS Mitigation by Direct-Control Reinforcement Learning'. In: *IEEE Trans. Network and Service Management* 17.1 (2020), pp. 103–117. DOI: 10.1109/TNSM.2019.2960202.

[38] Giuseppe Siracusano and Roberto Bifulco. 'In-network Neural Networks'. In: *CoRR* abs/1801.05731 (2018). arXiv: 1801.05731.

[39] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi and Roberto Bifulco. 'Running Neural Networks on the NIC'. In: *CoRR* abs/2009.02353 (2020). arXiv: 2009.02353.

[40] Viswanath Sivakumar, Tim Rocktäschel, Alexander H. Miller, Heinrich Küttler, Nantas Nardelli, Mike Rabbat, Joelle Pineau and Sebastian Riedel. 'MVFST-RL: An Asynchronous RL Framework for Congestion Control with Delayed Actions'. In: *CoRR* abs/1910.04054 (2019). arXiv: 1910.04054.

[41] Brent Stephens, Aditya Akella and Michael M. Swift. 'Your Programmable NIC Should be a Programmable Switch'. In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018*. ACM, 2018, pp. 36–42. ISBN: 978-1-4503-6120-0. DOI: 10.1145/3286062.3286068.

[42] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang and Kailash Gopalakrishnan. 'Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks'. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox and Roman Garnett. 2019, pp. 4901–4910.

[43] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, Nov. 2018. ISBN: 9780262039246. URL: http://incompleteideas.net/book/the-book-2nd.html.

[44] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz and Kunle Olukotun. 'Taurus: An Intelligent Data Plane'. In: *CoRR* abs/2002.08987 (2020). arXiv: 2002.08987.

[45] David E. Taylor and Jonathan S. Turner. 'Scalable Packet Classification using Distributed Crossproducting of Field Labels'. In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*. IEEE, 2005, pp. 269–280. DOI: 10.1109/INFCOM.2005.1497898.

[46] The P4.org Architecture Working Group. *$P4_{16}$ Portable Switch Architecture (PSA)*. Working Draft. Jan. 2021. URL: https://p4.org/p4-spec/docs/PSA.html (visited on 01/02/2021).

[47] Jaden B. Travnik, Kory Wallace Mathewson, Richard S. Sutton and Patrick M. Pilarski. 'Reactive Reinforcement Learning in Asynchronous Environments'. In: *Front. Robotics and AI* 2018 (2018). DOI: 10.3389/frobt.2018.00079.

[48] Asaf Valadarsky, Michael Schapira, Dafna Shahaf and Aviv Tamar. 'Learning to Route'. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*. Ed. by Sujata Banerjee, Brad Karp and Michael Walfish. ACM, 2017, pp. 185–191. ISBN: 978-1-4503-5569-8. DOI: 10.1145/3152434.3152441.

[49] Shibo Wang and Pankaj Kanwar. *BFloat16: The secret to high performance on Cloud TPUs*. Aug. 2019. URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus (visited on 01/02/2021).

[50] Keith Wiles. *Pktgen-DPDK – DPDK-based packet generator*. 2021. URL: https://github.com/pktgen/Pktgen-DPDK (visited on 29/05/2021).

[51] Shaolin Xie, Scott Davidson, Ikuo Magaki, Moein Khazraee, Luis Vega, Lu Zhang and Michael Bedford Taylor. 'Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds'. In: *ACM SIGOPS Oper. Syst. Rev.* 52.1 (2018), pp. 96–108. DOI: 10.1145/3273982.3273991.

[52] Xilinx. *Xilinx Revolutionizes the Modern Data Center with Software-Defined, Hardware Accelerated Alveo SmartNICs*. Feb. 2021. URL: https://www.xilinx.com/news/press/2021/xilinx-revolutionizes-the-modern-data-center-with-software-defined-hardware-accelerated-alveo-smartnics.html (visited on 11/05/2021).

[53] Zhaoqi Xiong and Noa Zilberman. 'Do Switches Dream of Machine Learning?: Toward In-Network Classification'. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 2019, pp. 25–33. ISBN: 978-1-4503-7020-2. DOI: 10.1145/3365609.3365864.

[54] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan and Yang Liu. 'BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning'. In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. Ed. by Ada Gavrilovska and Erez Zadok. USENIX Association, 2020, pp. 493–506.

[55] Noa Zilberman, Yury Audzevich, G. Adam Covington and Andrew W. Moore. 'NetFPGA SUME: Toward 100 Gbps as Research Commodity'. In: *IEEE Micro* 34.5 (2014), pp. 32–41. DOI: 10.1109/MM.2014.61.