Intelligent NIC Queue Management in the Dragonet Network Stack

Kornilios Kourtis[†] Pravin Shinde[‡] Antoine Kaufmann[§] Timothy Roscoe[‡]

[†]*IBM Research*[*] [‡]*ETH Zurich* [§]*University of Washington*[*]

Abstract

Recent network adaptors are equipped with multiple transmit and receive hardware queues combined with a wide variety of filtering and demultiplexing functionality. Contemporary network stacks depend on this functionality for high performance and strong isolation, but face a challenge: how to allocate a limited set of queues and filters to the flows of a given workload. The problem is made worse by the great variation in filter semantics between different adaptors.

In this paper we propose a solution to this problem using the Dragonet network stack. Dragonet takes a specification of the networking hardware functionality, and applies OS policies at runtime to allocate queues and filters to applications, deriving performance benefits in a way not possible with conventional OS protocol stacks. We describe a prototype of Dragonet which can use both Intel's DataPlane Development Kit and Solarflare's OpenOnload protocol framework, and present experimental results using microbenchmarks and memcached that show how Dragonet's queue management can be used for achieving high scalability and quality of service (QoS) guarantees as network flows come and go, independently of NIC hardware.

1 Introduction

The network bandwidth and latency demands of modern server applications impose two requirements on host network stacks: maximizing the system's raw performance, and efficiently sharing resources for which multiple applications and network flows compete.

The first ensures that each application can use the hardware (and perhaps more importantly the energy to drive the hardware) at its full potential. This alone, however, is not enough: consolidating multiple services on machines is also required to fully utilize the hardware. Given the diversity and fluidity of modern workloads, dynamically sharing hardware resources offers a way to maximize utilization and avoid overcommitting hardware.

Much recent work addresses the problem of maximizing raw performance in host networking stacks [7, 13, 14, 20]. This is generally achieved by ensuring that each packet is processed on a single core, distributing data structures to avoid contention, and eliminating OS overheads by techniques such as user-space networking and batching. Taking these techniques to their logical conclusion, other work proposes fully decoupling the data plane from the control plane at the OS level [1, 21]. All these approaches depend heavily on the use of NIC hardware queues to achieve isolation and high performance.

Efficiently allocating NIC queues across the network flows in a system is critical to performance and can enable higher degrees of service consolidation. In contemporary systems, however, there is no obvious, rigorous way to formulate, let alone solve, this problem: every NIC is different, and offers not merely different numbers of filters but different semantics and limits for the filters and directors used to demultiplex incoming packets [26].

We address this challenge with Dragonet, a network stack that allocates NIC queues to flows in a variety of different NIC implementations based on generic OS policies. We argue that the right place to implement this allocation decision is not in the NIC driver (as happens today), and certainly not in the NIC itself. Instead, NIC queue allocation should be performed in a hardware-independent way by the OS, and specifically by the network stack. Hence, Dragonet explicitly controls how NIC queues are allocated to network flows.

We do not aim to provide a solution for a specific queue allocation problem. These problems can change significantly over time as application demands and NIC hardware change. Instead, we are interested in creating the proper abstractions to formulate the problem, and building a network stack that can systemically solve it.

[*]Work completed while at ETH Zurich

Our contributions are as follows:

- We present a new model for NIC hardware that represents the capabilities of the NIC along with its configuration space for steering packets into queues. Our models are reactive dataflow graphs called Physical Resource Graphs (PRGs); nodes are annotated with semantic information about how the NIC operates.
- We demonstrate the value of the model in Dragonet (§3), a network stack that selects NIC configurations that satisfy generic OS policies. Instead of hard-coding policies, we express them using cost functions. Using the PRG abstraction and information about the network state, Dragonet explores the NIC's configuration space for a configuration that minimizes the cost function.
- We evaluate Dragonet in §4 using a UDP echo server and memcached, two modern, high-performance, NICs (the Intel i82599 and the Solarflare SFC9020), and two policies: load balancing and performance isolation for a set of given network flows.
- We show that proper NIC queues allocation significantly improves performance and enables performance isolation. Furthermore, we show that Dragonet finds good NIC configuration solutions with reasonable overhead.

We start with a discussion of our motivation and background for this work.

2 Motivation and Background

The work in this paper is motivated by combined trends in processors (and associated system software) and networking hardware.

2.1 Network hardware

We make a twofold argument. Firstly, exploiting NIC hardware queues is essential for keeping up with increasing network speeds in the host network stack. Secondly, doing so requires dealing with complex and hardware-specific NIC configuration.

The speed of networking hardware continues to increase; 40Gb/s adaptors (both Ethernet and Infiniband) are becoming commoditized in the datacenter market, and 100Gb/s Ethernet adaptors are available. The data rates that computers (at least when viewed as components of a datacenter) are expected to source and sink are growing.

At the same time, the speed of individual cores is not increasing, due to physical limits on clock frequency, supply voltage, and heat dissipation. As with other areas of data processing, the only solution to handling higher network data rates in end systems is via parallelism across cores, and this requires multiplexing and demultiplexing flows in hardware, before they reach software running on general-purpose cores.

Fortunately, all modern NICs support multiple send and receive queues, and include filtering hardware which can demultiplex incoming packets to different queues, typically ensuring that all packets of the same "flow" (suitably defined) end up in the same serial queue.

Multiple send and receive queues in NICs predate the multicore era: their original purpose was to reduce CPU load by offloading packet demultiplexing. This also had the useful property of providing a mechanism for performance isolation between flows without expensive support from the CPU scheduler. However, modern NIC functionality is highly sophisticated, and varies considerably between different vendors and different price points.

In this paper we also focus only on the receive path, where received packets are demultiplexed by hardware and directed to the appropriate receive queue. This is not to dismiss the transmit case, which can be surprisingly complex when quality-of-service concerns are taken into account, but managing receive queues as a system resource is a sufficiently complex problem in itself. Moreover, in the receive case the packet steering is performed by the NIC, rendering attempts to utilize NIC queues more challenging than on the send side.

2.1.1 Configuring queue filters in modern NICs

The primary challenge in exploiting NIC queues is configuration complexity: NICs offer a rich and diverse set of programmable filters for steering packets to hardware queues. To show this, we provide a simplified discussion of filters in the network adaptors we use in this paper.

The Intel i82599 [11] exports 128 send and 128 receive queues and supports:

1. *5-tuple filters*: 128 filters that match packets based on five fields: <protocol, source IP, destination IP, source port, destination port>, each of which can be masked so that it is not considered in packet matching.
2. *Flow director filters*: These are similar to 5-tuple filters, but offer increased flexibility at the cost of additional memory and latency (they are stored in the receive-side buffer space and implemented as a hash table with linked list chains). Flow director filters can operate in two modes: "perfect match", which supports 8192 filters and matches on fields, and "signature", which supports 32768 filters and the matches on a hash-based signature of the fields. Flow-director filters support global fine-grained masking, enabling range matching.
3. *Ethertype filters*: these filters match packets based on the Ethertype field (although they are not to be used for IP packets) and can be used for protocols

such as Fibre Channel over Ethernet (FCoE).

4. a *SYN filter* for directing TCP SYN packets to a given queue. This can be used, for example, to handle denial-of-service (DoS) attacks.
5. *FCoE redirection filters* for steering Fibre Channel over Ethernet packets based on FC protocol fields.
6. *MAC address filters* for steering packets into queue pools, typically assigned to virtual machines.

Finally, the i82599 also supports Receive Side Scaling [8, 18] in which packet fields are used to generate a hash value used to index a 128-entry table with 4-bit values indicating the destination queue.

In contrast, the Solarflare SFC9020 [28] NIC supports 1024 send and 1024 receive queues, 512 filters based on MAC destination and two kinds of 8192 filters each for TCP and UDP: a full matching in which the entire 5-tuple is considered, and a wildcard mode in which only destination IP and port are used. If a packet is matched by multiple filters, the more specific filter is selected. Moreover, each filter can use RSS to distribute packets across multiple queues (two different hash functions are supported).

2.2 System software

We now examine the evolution of network stacks and make two observations: modern network stacks indeed depend increasingly on NIC hardware to achieve good performance, but there is currently no solution that provides support for generic OS policies and deals with the complexity realities of modern NIC hardware.

2.2.1 RSS: Queue allocation in the NIC

Modern network stacks (and commodity OSes in general) have evolved incrementally from designs based on simple NICs feeding unicore CPUs. As multicore machines became dominant and the scalability of the software stack became a serious concern, significant efforts were made to adapt these designs to exploit multiple cores.

However, the difficulty of adapting such OSes while maintaining compatibility with existing hardware has limited the extent to which such stacks can evolve. This in turn has strongly influenced hardware trends.

For example, the most common method for utilizing NIC receive queues is Receive-Side Scaling (RSS) [8, 18]. The main goal of RSS is to remove the contention point of a single DMA queue and allow the network stack to execute independently on each core. With RSS, the NIC distributes incoming packets to different queues so that they can be processed by different cores. Packets are steered to queues based on a hash function applied to protocol fields (e.g., on a 4-tuple of IP addresses and TCP ports). Assuming the hash function distributes packets evenly among queues, the protocol processing load is balanced among cores.

The key drawback of RSS, as with any other hard-coding of policy into NIC hardware, is that the OS has little or no control over how queues are allocated to flows.

For example, RSS does not consider application locality. Maximizing network stack performance requires packet processing, including network protocol processing and application processing, to be fully performed on a single core. This increases cache locality, ensuring fast execution, and minimizes memory interconnect traffic, improving scalability. Hence, performant network stacks depend on a NIC configured to deliver packets to the queue handled by the core the receiving application resides.

2.2.2 Queue allocation in the driver

The shortcomings of RSS can be addresses by using more flexible NIC filters, and trying to intelligently allocate queues to flows using a policy baked into the device driver.

An example of this approach is Application Targeted Receive (ATR) [12] (also called "Twenty-Policy" in [20]). This is used by the Linux driver for the Intel i82599, where, transparently to the rest of the OS, the device driver samples *transmitted* packets (at a rate of 1:20 by default) to determine the core on which the application sending packets for a particular flow resides. Based on the samples, the driver then configures the NIC hardware to steer received packets to a queue serviced by that core.

The high-level problem with driver-based approaches is that the NIC driver lacks a global system view (available network flows, current OS policy, etc.) to make good decisions. Instead of using the full information, it will use heuristics based on hard-coded policies that may create more problems than they actually solve.

2.2.3 Queue allocation in contemporary stacks

In some cases, solutions for specific NIC queue management problems have been addressed in the network stack.

For example, Receive Flow Steering (RFS) [8] in the Linux kernel tries to address the poor locality of RSS and steer packets to cores on which the receiving application resides. When using RFS, the network stack keeps track of which core a particular flow was processed on (on calls to `recvmsg()` and `sendmsg()`), and tries to steer packets to the queue assigned to that core. RFS without acceleration, performs the steering in software, where Accelerated RFS uses NIC filters. In the latter case, drivers need to implement the `ndo_rx_flow_steer()` function, used by the stack communicate the desired hardware queue for packets matching a particular flow. The driver is required to poll the stack for expired flows in order to remove

stale filters. Currently, three NIC drivers (for Solarflare, Mellanox, and Cisco silicon) implement this function.

Another example is Affinity-Accept [20], that aims to improve locality for TCP connections. The incoming flows are partitioned into 4096 flow groups by hashing the low 12 bits of the source port, and each group is mapped to a different DMA queue, handled by a different core. The system periodically checks for imbalances and reprograms the NIC by remapping flow groups to different DMA queues (and hence cores).

Both of these methods are not without problems. Accelerated RFS operates on a very simplified view of NIC hardware. As a result, it cannot deal with the physical limits of the NIC (e.g., what happens when the NIC's filter table is full?), and at the same time cannot exploit all NIC hardware features. Affinity-Accept NIC queue management, on the other hand, targets a single NIC (the i82599), and cannot be applied to NICs that do not provide the ability to distribute flows based on the low 12 bits of the source port (e.g., Solarflare's SFC9020).

Perhaps more importantly, both of these techniques specifically target connection locality in a scenario in which all network flows are equal. It is not possible, for example, to utilize NIC queues to provide performance isolation to specific network flows.

2.2.4 Dataplane OSes

Recently, so-called "dataplane OSes" such as Arrakis [21] and IX [1] have proposed radically simplifying the design of the shared OS network stack. In particular, these systems attempt to remove the OS completely from the data path of packets. This is achieved using multiple queues, and adopting a hard allocation of queues to applications.

We believe that this structure will be increasingly compelling for high-performance server OSes in the future. For this reason, we orient our work in this paper more towards such dataplane-based OSes.

The adoption of dataplane-based designs, however, emphasizes the problem of intelligent queue allocation. For example, Arrakis [21] specifies a hardware model for virtualized network I/O, called virtual network interface cards (VNICs). VNICs are expected to multiplex and demultiplex packets based on complex predicate expressions (filters). In contrast to traditional network stacks, the application is assumed to have direct access to the NIC DMA ring and establishing the proper filters (both on the send and the receive path) is not just a performance concern, but the mechanism for establishing protection across applications running on the system.

Real NICs, however, are not perfect: they have limited numbers of queues, limited numbers of filters, limited filtering expressiveness, and, as we touched on in §2.1.1, complex configuration spaces. Hence, in the context of a dataplane OS the network stack is required to program NIC filters based on application requirements and global policies (e.g., which applications should operate without direct hardware access due to limited NIC capabilities).

2.3 Discussion

Overall, we believe that the OS should be capable of dealing with network queues analogously to how it deals with cores and memory, since ignoring NIC queues can lead to problems. In an OS like Linux, for example, it is not possible to ensure performance isolation for applications that use the network without exclusively allocating one or more NIC queues to the application. In OS dataplanes, the problem becomes more extreme because protection (e.g., from applications spoofing headers) is achieved by exclusive queue assignment.

Furthermore, as services are consolidated, a single machine is expected to deal with complicated, diverse, and varying workloads, potentially served by multiple applications with different requirements. The OS, therefore, should be able to dynamically assign hardware resources such as cores, memory, and NIC hardware queues to applications to fulfil these requirements.

While well known techniques exist for allocating cores and memory to applications, allocating NIC queues poses a significant challenge. For sending packets this is not a difficult task: the OS can just ensure that the queue is used exclusively by a single application. For receiving packets, however, allocating a queue requires ensuring that particular network flows are steered into a specific queue by the NIC. Different NICs offer different facilities for steering packets into queues, making queue allocation a non-trivial task. One way to perform this task, and how it is done in many cases in practice, is to manually select a static NIC configuration for a specific workload (e.g., via `ethtool` [6] for Linux). This leads to reduced flexibility in deployment and overcommitment of hardware resources.

In this paper, we argue that NICs should be configured by the core OS network stack based on the network state and given NIC-agnostic policies about how different network flows share resources. Moreover, this functionality should neither be hidden behind the device driver interface, nor left up to manual configuration.

Dragonet, our prototype network stack that realizes our ideas, is driven by dataplane OSes as a primary use case, but we argue that NIC queue management is a problem common to both dataplane OSes and monolithic kernels, and the techniques we present are applicable to both.

3 Managing queues in Dragonet

Dragonet operates on three abstractions: the network stack state, a NIC representation, and a system policy.

We model the first two as dataflow graphs using a common model [25], and express policies via cost functions.

The Dragonet network stack model, called *Logical Protocol Graph* (LPG), includes both static parts of protocol processing (e.g., checksum computation), but also dynamic network state (e.g., network flows as they come and go). The latter is build by applications interacting with the network. Applications operate on Dragonet via an API similar to traditional sockets. A server application, for example, calls the Dragonet variant of `listen()` and waits for incoming requests to serve. Such a call modifies the LPG by adding graph nodes to forward the appropriate network packets to the application.

An important design goal of Dragonet is to decouple policy specification from the NIC details. To do this we: (i) build NIC models that fully describe hardware operation and configuration, and (ii) describe queue allocation policies in a NIC-agnostic manner. To our knowledge, no other network stack supports this functionality. Our NIC models are called *Physical Resource Graphs* (PRGs) and are expressed in the Unicorn domain specific language [25]. We discuss them in detail in §3.1.

Dragonet configures NIC queues based on a system policy. A policy might, for example, safely enable direct queue assignment to applications by enforcing that only packets destined for these applications are steered into the queue. Furthermore, policies might also be concerned with performance: achieving load balancing or enforcing performance isolation for specific network flows.

We express user policies via cost functions (§3.4) that assign a cost to a specific queue allocation, given the set of active system flows in the system. Users can select an existing cost function, or provide their own. Writing a cost function does not require any knowledge about the NIC. Furthermore, cost functions can be composed to form complex policies. A system-wide cost function, for example, can split the cost in two parts: one representing the global queue allocation policy, and one representing the application policy. The latter can be determined by calling an application-specific cost function.

Using a PRG and a cost-function, Dragonet searches the NIC's configuration space for a configuration that minimizes the cost function. The configuration space is quite large, rendering naive search strategies impractical. As a result, Dragonet applies several techniques to efficiently search the configuration space (§3.2).

From the perspective of the search algorithm, the cost function is applied to a set of flows and a configuration for a PRG (that models the NIC). From the perspective of the policy writer, the cost function is applied to information about how system flows are mapped into queues. We translate between these two cost functions by computing how flows are mapped into queues (§3.3).

3.1 Modeling NICs

The PRG models two aspects of a NIC: its configuration and its hardware capabilities. Broadly speaking, a PRG describes what are the possible configurations for the NIC via a set of configuration variables, and how different values for these variables affect the behavior of the NIC.

Configuration values range from boolean values representing NIC hardware registers that can turn particular NIC features on and off, to queue steering tables. Such a table might contain multiple filter entries, each with multiple fields, such as a 5-tuple identifying a flow, a queue id, and a priority for applying the filter.

Applying values to configuration variables refines the graph. A configured PRG (i.e., one with no unassigned configuration variables) fully describes how the NIC operates: what happens when a packet arrives to the NIC from the wire, and what happens when a packet is queued for transmission by software. The PRG models the NIC not necessarily in terms of how the hardware is built, but rather on how it operates on network packets.

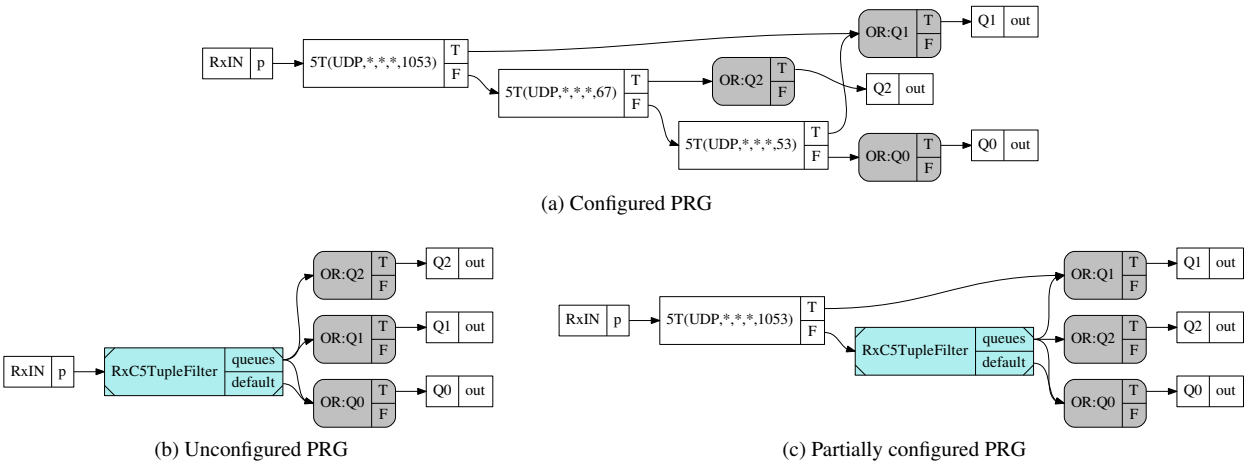
Essentially, PRGs provide a description of the NIC semantics to the network stack. Focusing on NIC queues, a configured PRG describes what are the available receive and send queues, how packets are steered into the receive DMA queues, and how packets are processed on the transmit DMA queues. An example PRG is shown in Fig. 1. We discuss it in detail next.

We build our graph-based models for NICs (but also for the network stack) using three node types: Function nodes (*F-nodes*), Operator nodes (*O-nodes*), and Configuration nodes (*C-nodes*).

F-nodes are processing nodes. Each F-node has a *single* input edge and multiple output edges organized in ports. An F-node processes input data and enables a *single* output port. Enabling an output port results in enabling the nodes pointed to by the edges of the port. Enabled nodes are executed until a sink node (i.e., a node without out edges) is reached, at which point the packet processing is completed.

O-nodes are used to combine outputs from multiple nodes, and each corresponds to a logical operator (`OR`, `AND`, etc.). Because O-nodes have multiple incoming edges, they cannot be implemented as F-nodes. The nodes that have ingress edges to an O-node are called its *operands*. Each operand connects two output ports to the O-node: one corresponding to a *true* and one to a *false* value. To simplify our figures, we sometimes omit operand edges. O-nodes activate one of their output ports (`T` for *true* and `F` for *false*) based on the usual semantics of logic operators.

Fig. 1a shows an example of the receive side of a configured PRG. The example is a simplified version of the Intel i82599 PRG. F-nodes have white background, while O-nodes have gray. The `Q1`, `Q2`, `Q3` nodes represent the



(a) Configured PRG

(b) Unconfigured PRG

(c) Partially configured PRG

Figure 1: Example PRG graph, based on the Intel i82599 NIC.

receive queues of the NIC. `5T` nodes represent 5-tuple filters of the i82599. A 5-tuple consists of the protocol, the source/destination IP address, and the source/destination port. In this example, the 5-tuple filters only specify the protocol and the destination port, leaving the other fields masked so that they match everything. The example PRG models a NIC where UDP packets with destination ports `53` and `1053` are steered to `Q1`, UDP packets with destination port `67` are steered to `Q2`, while all other packets end up in the default queue, `Q0`.

Dragonet uses boolean logic for reasoning. Each F-node port is annotated with a boolean predicate which describes the properties of the packets that will enable the port. Our expressions are build with atoms that are tuples of a label and a value. The label typically corresponds to a packet field. For example the predicate for the true (T) port of filter node '`5T(UDP,*,*,*,53)`' is: '`(EtherType,IPv4) AND (IPv4Prot,UDP) AND (UDPDstPort,53)`'. Note that it is not possible to have a different value for the same label. Hence, we can simplify expressions such as '`(UDPDstPort,53) AND (UDPDstPort,67)`' to false.

C-nodes represent the configuration space of the NIC. Each C-node corresponds to a configuration variable. It specifies the set of possible values, and how applying a value affects the graph. Applying a configuration value to a C-node results in a set of new nodes and edges that replace the C-node in the graph.

In the example of Fig. 1b, the `RxC5TupleFilter` C-node is configured with a list of values, each defined by a 5-tuple filter and a queue id. Applying: [`5T(UDP,*,*,*,1053)` → `Q1`, `5T(UDP,*,*,*,67)` → `Q2`, `5T(UDP,*,*,*,53)` → `Q1`] in `RxC5TupleFilter`, for example, results in the configured graph shown in Fig. 1a.

Each C-node defines how the graph is modified by adding new nodes or edges, based on a configuration value. If we allow each C-node to modify the PRG in arbitrary ways (i.e., add edges and nodes everywhere in the graph), reasoning about configuration becomes challenging, especially when multiple C-nodes exist in the graph. To avoid this, we constrain C-node modifications to be local in the area defined by the C-node.

Specifically, all new edges added by a C-node must have a source node which is either a new node, or a node x for which an edge exists from x to the C-node in the unconfigured graph. Analogously, all new edges must have a destination node which is either a new node, or a node x for which an edge exists from the C-node to x in the unconfigured graph. Under this restriction, the changes that a configuration node can apply to the graph are restrained to the nodes that it is connected with.

In our example (Figures 1a and 1b), configuring `RxC5TupleFilter` adds 3 `5T` nodes. Hence, for all new edges the source node can either be the `RxIN` (since there is a `RxIN` → `RxC5TupleFilter` edge in the unconfigured graph) node, or a `5T` node. Similarly, the destination node can either be one of the `OR:Qx` nodes, or a `5T` node. This restriction allows us to maintain predicate information when incrementally configuring the PRG (see §3.3).

3.2 Searching the PRG configuration space

Dragonet operates by searching the PRG configuration space for a configuration that minimizes the cost function. From the perspective of the search algorithm, a cost function (for a particular PRG) evaluates a configuration given a set of active flows. In contrast, the cost function that defines the policy accepts how flows are mapped into queues as input. Next, we discuss the search algorithm, and in subsequent sections (§3.3 and §3.4) we discuss computing the flow-to-queue mapping and expressing policies as cost functions.

3.2.1 Network flows

We define a flow as a predicate on a network packet. For example, a listening UDP socket defines the class of packets that will reach the socket. Similarly, a TCP connection defines the class of packets that are a part of this connection. It is worth noting that, even though UDP is connectionless, we can still define a UDP flow as the class of packets that have specific source and destination UDP endpoints (IP/port).

Determining active flows, however, is not trivial. Even for connection-oriented protocols like TCP the fact that the connection exists, does not mean that the connection is active (i.e., packet exchange might be minimal). Active network flows can be identified based on a traffic monitor mechanism, or registered directly by the application. In our current prototype, we use the latter approach (via a proper API), but we also plan to add support for the traffic monitoring in future versions.

If more fine-grained metrics than individual flows are needed (e.g., considering the traffic rate of each active network flow, rather than just whether it is active or not), our queue management algorithms can be easily adapted accordingly.

3.2.2 PRG oracles

A first observation is that a comprehensive search of the configuration space would require an unreasonable amount of time. For example, in many cases queue filters include IP addresses or ports in their configuration (e.g., 5-tuple filters). Considering *all* possible values for these fields is unrealistic. We can prune a large part of the search space, however, if we filter values for these fields based on the active flows. For example, if a field of a configuration value corresponds to a source IP address, we only consider source IP addresses that appear on the current network flows.

To further reduce the search space, we use NIC-specific configuration space iterators we call *oracles*. An oracle allows injecting NIC-specific knowledge into the search by, for example, eliminating symmetric configurations, or prioritizing specific configurations over others. An oracle accepts the current PRG configuration and a flow, and returns a set of incremental changes to the given configuration.

Our oracle implementations for the Intel i82599 and Solarflare SFC9020 NICs generate configurations that map the given flow to different queues by adding appropriate filters. We also apply some basic NIC-specific optimizations. For example, the i82599 oracle will only use flow director filters if all the 5-tuple filters are used (see §2.1.1).

Algorithm 1: Search algorithm sketch

Input : The set of active flows F_{all}
Input : A cost function `cost`
Output : A configuration c

```
c ← C_0    // start with an empty configuration
F ← ∅              // flows already considered
foreach f in F_all do
    // Get a set of configuration changes
    // for f that incrementally change c
    CC_f ← oracleGetConfChanges(c, f)
    F ← F + f                          // Add f to F
    find cc ∈ CC_f that minimizes cost(PRG, c + cc, F)
    c ← c + cc  // Apply change to configuration
```

3.2.3 Search algorithm

We use a greedy search algorithm, which starts with an empty configuration and accepts a set of flows and a cost function as input. We opted for a greedy strategy due to its simplicity and because it can be applied incrementally as new flows arrive (see §3.2.4).

A simplified version of our search is shown in Alg. 1, in which each step operates on a single flow (f) and refines the configuration from the previous step (c). At each step, we invoke the oracle to acquire a new set of configuration changes (CC_f) that incrementally modify the previous configuration. A configuration change can be applied to a configuration to form a new configuration ($cc + c$). We select the configuration change that minimizes the cost for the current set of flows (F), update the configuration and continue until there are no more flows to consider.

Depending on the problem, a greedy search strategy might not be able to reach a satisfactory solution. To deal with this issue, we allow cost functions to return whether the solution is acceptable or not in their cost value. An acceptable solution always has a lower cost than a unacceptable solution. If after the greedy search the algorithm is not able to reach an acceptable solution, the algorithm "*jumps*" to a different location of the search space and starts again by rearranging the order of the flows. To avoid jumps, we support a heuristic where cost functions are paired with a function to sort the flows before the search algorithm is executed.

3.2.4 Incremental search

The above algorithm operates on all the active flows. As flows come and go in the system, we need to consider that the optimal configuration might change. A naive solution for dealing with added/removed flows would be discard all state and redo the search. This, however, induces significant overhead and does not scale well as the number of flows increase. Next, we discuss how we deal with flow arrival and removal incrementally.

Adding flows is simple in the greedy search algorithm: we start from the current state and perform the necessary number of iterations to add the new flows. If an acceptable solution is not reached, we rearrange the flows (applying the sorting function if one is given) and redo the search.

Removing flows is more complicated to deal with. One approach would be to backtrack to a search state that does not include any removed flows, and incrementally add the remaining flows in the system. Because this can lead to large delays, we remove flows lazily instead.

As can be seen in Alg. 1, each flow is associated with a configuration change. When this change is applied to the PRG, a new set of nodes are added to the graph. When a flow exits the system, we maintain the configuration as is, and mark the configuration change that was paired with the removed flow as stale. This results in the nodes added by the configuration change to remain in the PRG, even though the corresponding flow was removed.

At some point, we need to remove the stale configuration changes. To do that we can backtrack the search as mentioned above. As an optimization, we allow oracles to repurpose the graph nodes that are associated with stale configuration changes when new configurations are needed. To that support this, we define special configuration changes called *replacements*. In our current prototype, replacements are implemented by changing the predicates of the generated nodes for the replaced configuration change, but not the graph structure.

3.2.5 Executing the search

There are two main components in Dragonet, executed as different threads: the *control thread*, and the *protocol threads* (one thread per NIC queue). The protocol threads implement the network protocols (they effectively execute the LPG), passing packets from the NIC to applications and vice-versa. The control thread is responsible for executing the search. It accepts notifications about new or deleted active flows, and maintains state about the registered applications and their endpoints (sockets), the current active flows, and the NIC configuration. Once the search is completed, the controller passes the resulting configuration to the NIC driver which configures the NIC.

An important aspect is the granularity that the search is performed. Performing a new search for each added or removed flow is possible, but potentially excessive. There are several factors on which the granularity selection depends on: the overhead of the search, the rate of changes in flows, the importance of quickly reacting to changes, and requirements for precision in the search result. Overall, there is no single perfect way to solve this issue, since it is highly application dependant. In our current prototype, we leave it up to the application to trigger the search. Applications can be throttled on the frequency they request a search using a token-based or similar algorithm.

All new flows are assigned to the default queue. Hence, creating a new connection does not depend on executing the search. The new flows will be serviced by the default queue until the next search concludes, at which point they might be assigned to new queues.

3.3 Mapping flows into queues

Policy cost functions accept how active flows are mapped into NIC queues (*flow mapping*) as an argument and return a cost value. We can compute the flow mapping as follows. First, we apply the change to the configuration and use it to configure the PRG. Given a configured PRG, we can determine the queue on which a flow will appear using a flow predicate and a depth-first search starting from the source node (e.g., `RxIn` in Fig. 1a). For each of the flow's activated nodes, we compute the port that will be activated, and continue traversing the graph.

In F-nodes, we determine the activated port by checking the satisfiability of the conjunctions formed by the flow predicate and the port predicates. We assume that the flow predicate contains enough information to determine which port will be activated (i.e., for each flow predicate, only one port predicate will be satisfiable). We had no issues with this assumption. Although boolean satisfiability is an NP-complete problem, in practice the flow and port expressions contain a small number of terms for this to become a restriction.

For O-nodes, we check the incoming edges and determine the activated port using the usual operator semantics. For example, in an OR node the true (false) port is activated if the flow appears in the true (false) edge of *one* (*all*) operand(s). Note that for each operand, the flow can appear only in one edge (either true or false).

Computing the flow mapping dominates search execution time, and the method described above performs redundant computations. To improve search performance, we incrementally compute the flow mapping by maintaining a *partially configured PRG* across search steps. Applying a *configuration value* to a C-node results in the C-node being removed. Applying a *configuration change* to a C-node maintains the C-node and results in a partially configured PRG.

An example of a configuration change is "insert a `5T(UDP,*,*,*,1053)` $\rightarrow$ `Q1` filter". Applying this change to the graph of Fig. 1b results in the partially configured PRG of Fig. 1c.

To incrementally compute the flow mapping, we maintain information about how active flows are mapped in node ports in the partially configured graph. In Fig. 1c, for example, we can compute what flows match the T port of `5T(UDP,*,*,*,1053)` (and will consequently reach `Q1`)

and what flows match the F port. Note that C-nodes act as barriers, because we cannot compute flow mappings beyond them.

When an incremental change is applied, we propagate flow information to each newly inserted node. If the configuration change is a replacement, we recompute flow mappings for the affected nodes and propagate changes. As we show in our evaluation (§4.4), incrementally computing the flow mapping leads to a significant performance improvement for the search algorithm.

3.4 Specifying policies with cost functions

Next, we discuss expressing queue allocation policies via cost functions that operate on a mapping of flows onto queues. In a deployment of our system, we expect that there will be a number of available policies, as well as an interface that allows system administrators to provide their own. We examine two policies as examples.

First, **load balancing** aims to balance the flows to the available queues. This policy is expressed easily using a cost function: we compute the variance of the number of flows in each queue.

Algorithm 2: Cost function for performance isolation policy

Input : The available queues Qs and flows F
Input : K queues assigned to HP flows
Input : A function `isHP()` to determine if a flow is HP
Input : The flow mapping $fmap$
Output : A cost value
`// determine HP and BE flows`
$(F_{HP}, F_{BE}) \leftarrow$ partition F with `isHP()` function
`// determine queues for each flow class`
$Qs_{HP} \leftarrow$ the first K queues from Qs
$Qs_{BE} \leftarrow$ the remaining queues after K are dropped from Qs
`// are flows assigned to the proper queues?`
$OK_{HP} \leftarrow \forall f \in F_{HP} : fmap[f] \in Qs_{HP}$
$OK_{BE} \leftarrow \forall f \in F_{BE} : fmap[f] \in Qs_{BE}$
if *(not* OK_{HP}*) or (not* OK_{BE}*)* **then**
  return *CostReject 1*
$B_{HP} \leftarrow$ compute balancing cost of F_{HP} on Qs_{HP}
$B_{BE} \leftarrow$ compute balancing cost of F_{BE} on Qs_{BE}
if F_{HP} *is empty* **then**
  return *CostAccept* B_{BE}
else if F_{BE} *is empty* **then**
  return *CostAccept* B_{HP}
else return *CostAccept* $B_{HP} + B_{BE}$

Second, we consider a policy that offers **performance isolation** for certain flows. We distinguish between two classes of flows: *high-priority* (*HP*) and *best-effort* (*BE*) flows. Following the dataplane model, each class is served by an exclusive set of threads, each pinned on a system core, each operating on a single DMA NIC queue. To ensure the good behaviour of the HP flows we allocate a number of queues to be used only by these flows, and leave the rest of the queues for the BE flows. As a secondary goal, each class provides its own cost function for how flows are to be distributed among the queues assigned to the class. This illustrates the composability of cost functions, where each class may provide its own cost function (in this example we use load balancing), while a top-level cost function describes how queues are assigned to classes.

The cost function for this policy is also simple. Its implementation is about 20 lines of Haskell code, and its pseudocode is shown in Alg. 2. It rejects all solutions that assign flows to queues of different classes, and returns an accepted solution with a score equal to the sum of the balancing cost for each class.

In our experience, cost functions, although in many cases small and conceptually simple, can be very tricky to get right in practice. Operating on the Dragonet models, however, considerably eased the development process because we could experiment and build tests for our cost functions without the need to execute the stack.

3.5 Implementation

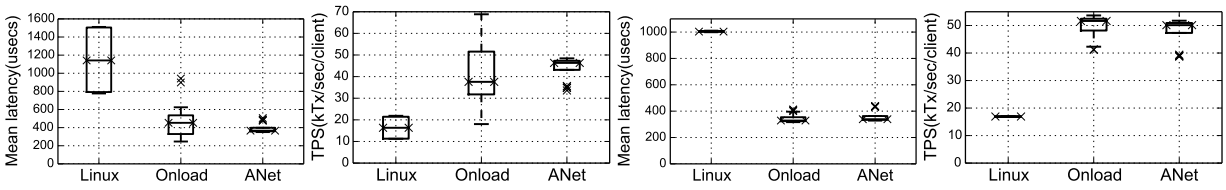
Dragonet is written in Haskell and C.[1] The Haskell code is responsible for implementing the logic, while the C code implements low-level facilities such as communication with the NIC drivers and stack execution.

PRGs and LPGs are written in the Unicorn domain specific language [25], which is embedded in Haskell. Configuration functions for applying configuration values to C-nodes are written in Haskell.

Dragonet runs in user-space, and spawns a control thread and a number of protocol threads, each operating on a different receive/send queue pair. In each of these queue pairs, Dragonet connects a separate instance of the software stack implementation (i.e., the LPG). This ensures that all processing of a single packet happens on the same core. This allows to specialize the LPG implementation based on the properties of the NIC queue that it is attached to. For example, if a queues is configured so that no packets for a particular application endpoint are received, we remove the relevant nodes for steering packets in this endpoint from the LPG instance connected to that queue.

The LPG is transformed from Haskell to a C data structured and sent from the controller thread to the protocol threads. It is executed in the protocol threads by traversing the graph and calling C functions that correspond to F-node and O-node functionality. Our current prototype supports UDP, IP, and ARP. All communication

[1] The Dragonet prototype source code is available in http://git.barrelfish.org/?p=dragonet



(a) Latency, 1024 bytes (b) Throughput, 1024 bytes (c) Latency, 64 bytes (d) Throughput, 64 bytes

Figure 2: Comparision of echo server performance on the Solarflare SFC9020 NIC for different network stacks

between Dragonet threads and application threads is done via shared-memory queues. More details can be found in [15].

The Dragonet driver for each NIC needs to implement: a PRG, its oracle, the shared memory queue for communication with Dragonet threads, and a function that accepts a PRG configuration and configures the NIC. We have implemented drivers for the Intel i82599 and the Solarflare SFC9020 NICs. The first uses Intel DPDK [10], while the second uses OpenOnload [27].

We are currently investigating several options for implementing the necessary boolean predicate logic. The implementation we describe here is based on a custom solver that we developed in Haskell, and we have also experimented with the Z3 SMT solver [4]. There is significant room for performance improvement in our current boolean solver.

4 Evaluation

In this section we evaluate our system. We first investigate if Dragonet has comparable performance to traditional network stacks under similar NIC configurations (§4.2). Next, we investigate the performance benefits of using Dragonet smart queue allocation capabilities. We specifically examine the performance effect of enforcing performance isolation for specific client flows in a memcached server (§4.3). Finally, we quantify the search overhead for Dragonet to find an appropriate NIC configuration (§4.4).

4.1 Setup

As a server, we use an Intel Ivy Bridge machine with 20 cores, running Linux (kernel version 3.13). The server is equipped with an Intel i82599 [11] and a Solarflare SFC9020 [28] NIC.

For load generators (clients), we use different multicore x86 machines (with the same software as the server) using an Intel i82599 [11] NIC to connect to the server over a 10GbE network. We always use the same allocation for client threads in the load generators to reduce the variance of the applied workload for each run.

Dragonet runs in its own process context (separate from applications) in user-space using one thread per NIC queue (for polling NIC receive queues and application send queues), and one controller thread that runs the solver. We allocate 10 cores to the 11 Dragonet stack threads (and subsequently 10 NIC queues), and 10 cores to the server application. Although the protocol threads are not required to have an exclusive core, we do this because our queue implementation used for communication between the application and the protocol threads support only polling and cannot block.

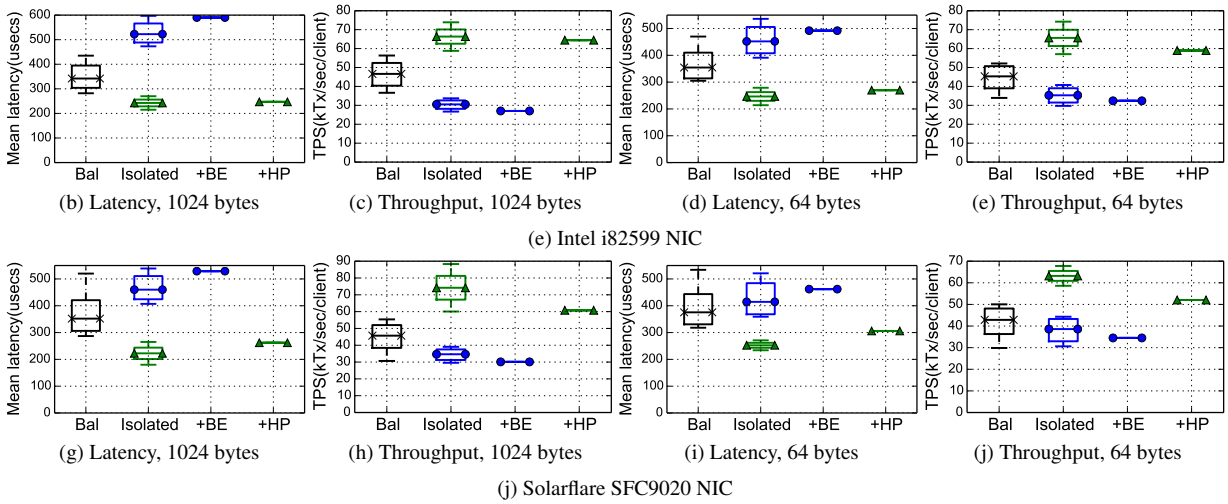
4.2 Basic performance comparison

To put Dragonet's performance in perspective, we start with a comparison to other network stacks. We do not claim that Dragonet has the best performance. Our goal is to show that Dragonet has reasonable performance under the same conditions, and exclude the possibility that the benefits of smart queue allocation are artifacts of Dragonet's poor performance. To that effect, we use a load-balancing NIC queue allocation in which flows are evenly distributed across queues policy for Dragonet and RSS for the other stacks.

We use a UDP echo server with 10 threads, and generate load from 20 clients running on different cores on 4 machines. Each client runs a netperf [19] echo client, configured to keep 16 packets in flight.

The results are shown in Fig. 2, in which we show boxplots for mean latency and throughput as reported for each of the 20 clients, using 64 and 1024 byte packets. We compare Dragonet (*Anet*) against the Linux kernel stack (version 3.13) (*Linux*) and the high-performance Solarflare OpenOnload [22] network stack (*Onload*) using the Solarflare SFC9020 NIC, which we configure for low-latency. OpenOnload is a user-level network stack that completely bypasses the OS in the data path and can be transparently used by applications using the BSD sockets system calls. We got similar results for the Intel NIC, which we omit for brevity (no equivalent to OpenOnload exists for the Intel NIC).

Linux network stack has the worst performance. For example, for 1024 bytes we measured a median latency of



(b) Latency, 1024 bytes (c) Throughput, 1024 bytes (d) Latency, 64 bytes (e) Throughput, 64 bytes

(e) Intel i82599 NIC

(g) Latency, 1024 bytes (h) Throughput, 1024 bytes (i) Latency, 64 bytes (j) Throughput, 64 bytes

(j) Solarflare SFC9020 NIC

Figure 3: Evaluation of memcached using a priority cost function on Solarflare SFC9020 and Intel i82599 using 10 queues.

1.14 ms and a median throughput of 16.3K transactions/s across clients. For Onload (Dragonet), we measured a median latency of 453 µs (366 µs), and a median throughput of 36.6K (46.3K) transactions/s across all clients. The aggregate throughput for Onload (Dragonet) is 803K (878K) transactions/s, and the aggregate transfer rate is 6.6 (7.2) Gbit/s.

Onload and Dragonet perform significantly better than Linux mainly due to bypassing the OS in the data path. Dragonet and Onload have similar performance. For 1024 byte requests, Dragonet outperforms Onload, while the reverse is true for 64 byte requests.

4.3 Performance isolation for memcached

In this section we evaluate the benefits of smart NIC queue allocation using a (ported to Dragonet) UDP memcached server as an example of a real application. We consider a scenario in which a multi-threaded memcached serves multiple clients (e.g., web servers) and we want to prioritize requests from a subset of the clients that we consider high-priority (HP clients). We use the performance isolation cost function described in §3.4 to allocate 4 out of 10 NIC queues exclusively to HP clients. The thread on the default queue (queue 0) maintains a hash table to detect new flows and inform Dragonet of their presence.

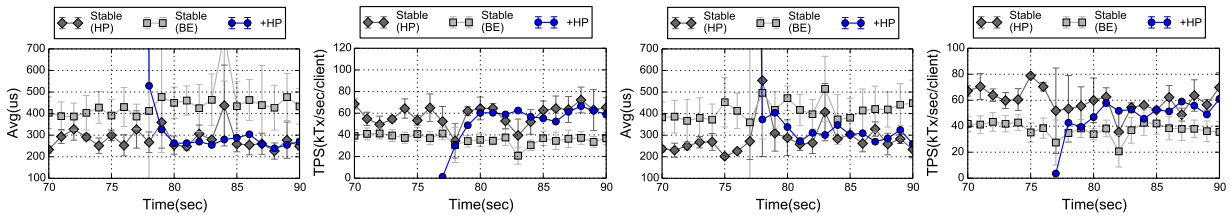
Our experiment is as follows: we start a multi-threaded memcached server with 10 threads exclusively using 10 of the server's cores. We apply a stable load from 2 HP clients, and 18 best-effort (BE) clients, each with 16 flows, resulting in a total of 320 flows. We generate the load using memaslap, a load generation and benchmark tool for memcached servers.

After 10 s we start a new BE client, which runs for 52 s. After the BE client is finished we wait for 10 s and start a new HP client, which also runs for 52 s. Each of the new clients are added as new flows and a search is triggered by the server. We collect aggregate statistics from each client (mean latency and throughput), and show results for 64 and 1024 byte server responses for both NICs in Fig. 3. We use a 10/90% Set/Get operation mix.

Each plot includes: (i) the performance of the workload under a load-balancing policy (*Bal*) for reference, (ii) the performance of the workload under the performance isolation policy (*Isolated*), (iii) the performance of the added BE client (*+BE*), and (iv) the performance of the added HP client (*+HP*). For the performance isolation policy, we use two different boxplots in our graphs: one that aggregates the HP clients (green color, median marked with triangles), and one that aggregates the BE clients (blue color, median marked with circles). For the load-balancing policy, we use one boxplot (black color, median marked with 'x') for all clients.

As an example, we consider the case of the Intel i82599 NIC for 1024 byte requests. Under a load-balancing policy, the median average latency across clients is 342 µs, the median throughput is 46.6K transactions/s, and the aggregate throughput is 927.5K transactions/s. Under the performance isolation policy, HP clients achieve a median latency of 246.5 µs (27% reduction compared to balancing) and a median throughput of 65.6K transactions/s (41% improvement compared to balancing). Furthermore, Newly added HP flows and BE flows maintain the same level of performance as their corresponding classes in the stable workload.

For all cases, Dragonet queue allocation allows HP clients to maintain a significantly higher level of performance via a NIC configuration that is the result of a NIC-



(b) Latency (c) Throughput (e) Latency (f) Throughput

(c) Intel i82599 NIC (f) Solarflare SFC9020 NIC

Figure 4: Impact of adding an HP client when using 64 byte requests

agnostic policy. To the best of our knowledge, no other network stack enables this.

We also instruct memaslap clients to provide results about latency and throughput every one second (the minimum possible value). Fig. 4 shows our results for 64 byte requests, focusing on adding an HP client. It shows median throughput and latency for all clients in the initial workload, and the individual throughput and latency measurements for the new HP client. The initial latency of the HP client is high (12.6 ms for i82599 and 4.5 ms for SFC9020) and is omitted from the graphs for clarity. During the addition of the new client, all clients performance drops for one second (sampling period), but it quickly stabilizes again. We attribute these delays to the Dragonet solver executing and contending with the rest Dragonet threads, and the time it takes to pass the new LPG graph to the protocol threads after a new configuration is found. We believe that with careful engineering the majority of these overheads can be eliminated. For example, in many cases the LPG graph does not actually change across different configurations so it is not necessary to actually reconstruct it in the protocol threads.

4.4 Search overhead

Here, we examine the search overhead, i.e., the time it takes Dragonet to complete the search. For each possible new configuration given by the oracle, Dragonet computes how flows are mapped to queues (see §3.3), which dominates the search cost. Table 1 shows the search cost for a varying number of flows (ranging from 10 to 500) when using 10 queues on the Intel i82599 PRG for the balancing cost function. The *Basic* column shows the cost of finding a solution without incrementally computing the flowmap. All results in subsequent columns use incremental flowmap computation. They show the cost for computing the solution from scratch (*full*), but also the cost of incrementally adding (*+1/+10 flows*) and removing flows (*-1/-10 flows*). For example, it takes 484 ms to find a solution for 10 new flows added when the system has 500 flows. Because we apply a lazy approach, removing flows has small overhead.

	Basic	Incremental flowmap computation				
flows	full	full	+1 fl.	+10 fl.	-1 fl.	-10 fl.
10	11 ms	17 ms	2 ms	22 ms	9 µs	23.7 µs
100	1.2 s	0.6 s	9 ms	94 ms	74 µs	117 µs
250	13 s	4 s	21 ms	219 ms	190 µs	277 µs
500	76 s	17 s	43 ms	484 ms	382 µs	548 µs

Table 1: Search overhead for Intel i82599 PRG using 10 queues

The biggest challenge of our approach is reducing the search cost, which is not an easy problem. Our results show that incrementally computing flow mappings, not only allows to efficiently add and remove flows with small overhead, but also significantly improves the full computation because information is kept across search steps.

4.5 Discussion

Overall, our evaluation shows that Dragonet offers significant benefits by automatically configuring NIC queues. But, there is clearly a tradeoff: the search overhead. In general, the number of flows and the rate of changes in the workload determine the applicability of our approach. Considering two extremes, our system is well-suited for coarse-grained machine allocations in datacenters for applications whose execution spans minutes, but cannot deal with load spikes in the order of a few miliseconds.

There are two aspects of the search overhead: constants and scalability in the number of flows. In this paper, we focused on the latter and showed that incrementally computing the necessary information can significantly alleviate the overhead. We believe that there is significant room for improvement in both of these aspects. One one hand, we use a basic search algorithm that can be significantly improved. On the other hand, our profiling showed that more than 10% of the search execution time goes to basic operations (e.g., finding successors and predecessors) in the functional graph library [5] we use. Moreover,

more than 10% of the time goes to predicate computation done with our suboptimal library, even though we use Haskel's mutable hash tables [3, 17] to cache predicates.

5 Related work

Our techniques build on many areas of related work; we structure this discussion around (i) high-performance network stacks, (ii) network stacks organized as graphs, and (iii) declarative techniques for dealing with hardware complexity.

Scalable network stacks Recent work [7, 14, 20] has focused on improving the poor TCP performance for small messages and short-lived connections. Unsurprisingly, all of these works aim for good locality, i.e., ensuring that all processing for a particular network flow happens on the same core, which requires the use of NIC steering filters to distribute packets among cores. Affinity-accept [20], redesigns the Linux `accept` system call so it preferably returns connections of flows processed in the same core as the application, and provides short- and long-term load balancing mechanisms. MegaPipe [7] is a based on a redesigned API, and in addition to splitting up the acceptance of new connections among different cores, batches multiple requests and their completion notifications in a single system call to improve performance. mTCP [14] applies similar techniques to a user-space network stack that interacts with applications via a traditional socket API. Taking a step further, "data plane OSes" [1, 21] propose fully removing the OS from the data path of packets.

Network stacks organized as graphs Structuring the network stack as a graph is not a new idea. The *x*-Kernel [9] aimed to provide a common set of abstractions for building network protocols without, however, sacrificing performance compared to an ad-hoc implementation. Each protocol (e.g., IP, UDP, TCP) in the *x*-Kernel is represented as an object. The organization of the protocols is determined at kernel configuration time, and each protocol is given the capability to invoke the low-level protocols upon which it depends. Click's [16] defines a software architecture for building modular software routers. A program in Click is a directed graph, build out of nodes called elements. Each element implements a specific computation in the software graph and has a number of input and output ports.

Contrarily to these works, we target managing NIC hardware rather than building good software abstractions.

Declarative techniques for dealing with hardware complexity Dragonet is also inspired by systems applying declarative techniques for addressing hardware complexity, particularly in the OS design space. For example, the Barrelfish OS uses constraint logic programming to deal with problems such as PCI configuration, multicast messaging, and global resource management [23, 24]. In another context, Merlin [29] provides a declarative language for expressing high-level policies for managing resources in software-defined networks.

6 Conclusion and Future Work

In this paper, motivated by the increasing importance of exploiting NIC queues, we presented Dragonet, a network stack that can effectively reason about and utilize NIC hardware queues. Dragonet treats queue allocation as a primary concern instead of hard-coding queue allocation policy in the protocol implementation or in the NIC driver.

This leads to a radical network stack design. Dragonet operates on a NIC hardware abstraction that represents NICs as directed graphs. Using this abstraction, Dragonet searches the NIC's configuration space for optimal solutions based on cost functions expressing policy requirements. We show the performance benefits of our approach by implementing and evaluating a load-balancing and a performance isolation policy.

As future work, we aim to further extend Dragonet to deal with a wider range of problems. We aim to explore alternative search strategies and experiment with additional policies. We are specifically interested in investigating the tradeoff between result precision and search overhead, as well as applying advanced search techniques to explore the configuration space (e.g., metaheuristics [2]).

Acknowledgements

We thank the Barrelfish team at ETH Zurich, the anonymous reviewers, and our shepherd, Gilles Muller, for their feedback.

References

[1] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014).

[2] BLUM, C., AND ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv. 35*, 3 (Sept. 2003), 268–308.

[3] COLLINS, G. Hackage: The hashtables package. https://hackage.haskell.org/package/hashtables.

[4] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), TACAS'08/ETAPS'08, pp. 337–340.

[5] Erwig, M. Inductive graphs and functional graph algorithms. *J. Funct. Program. 11*, 5 (Sept. 2001), 467–492.

[6] ethtool - utility for controlling network drivers and hardware. https://www.kernel.org/pub/software/network/ethtool/.

[7] Han, S., Marshall, S., Chun, B.-G., and Ratnasamy, S. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 135–148.

[8] Herbert, T., and de Bruijn, W. Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, May 2013.

[9] Hutchinson, N. C., and Peterson, L. L. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (January 1991).

[10] Intel. Intel DPDK: Data Plane Development Kit. http://www.dpdk.org/.

[11] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, December 2010. Revision 2.6.

[12] Intel corporation. Intel 10 Gigabit Linux driver. https://www.kernel.org/doc/Documentation/networking/ixgbe.txt, Aug. 2013.

[13] Jang, H.-C., and Jin, H.-W. Miami: Multi-core aware processor affinity for TCP/IP over multiple network interfaces. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects* (Washington, DC, USA, 2009), HOTI '09, IEEE Computer Society, pp. 73–82.

[14] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation* (Seattle, WA, Apr. 2014), NSDI '14, pp. 489–502.

[15] Kaufmann, A. Efficiently executing the Dragonet network stack. Master's thesis, ETH Zurich, October 2014.

[16] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (August 2000).

[17] Launchbury, J., and Peyton Jones, S. L. Lazy functional state threads. *SIGPLAN Not. 29*, 6 (June 1994), 24–35.

[18] Microsoft corporation. Scalable networking. http://msdn.microsoft.com/en-us/library/windows/hardware/ff570736%28v=vs.85%29.aspx.

[19] netperf 2.6.0. http://www.netperf.org/netperf/.

[20] Pesterev, A., Strauss, J., Zeldovich, N., and Morris, R. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys)* (2012).

[21] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014).

[22] Pope, S., and Riddoch, D. Openonload: A user-level network stack. http://www.openonload.org/openonload-google-talk.pdf, 2008.

[23] Schüpbach, A. *Tackling OS Complexity with Declarative Techniques*. PhD thesis, ETH Zurich, 2012.

[24] Schüpbach, A., Baumann, A., Roscoe, T., and Peter, S. A declarative language approach to device configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 119–132.

[25] Shinde, P., Kaufmann, A., Kourtis, K., and Roscoe, T. Modeling NICs with Unicorn. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems* (2013), PLOS '13, pp. 3:1–3:6.

[26] SHINDE, P., KAUFMANN, A., ROSCOE, T., AND KAESTLE, S. We need to talk about NICs. In *14th Workshop on Hot Topics in Operating Systems* (May 2013).

[27] SOLARFLARE COMMUNICATIONS, INC. *Onload User Guide*. 9501 Jeronimo Road, Irvine, California 92618, 2010. Version 20101221.

[28] SOLARFLARE COMMUNICATIONS, INC. *Solarflare SFN5122F Dual-Port 10GbE Enterprise Server Adapter*, 2010.

[29] SOULÉ, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), ACM, pp. 213–226.