Modeling NICs with Unicorn

Pravin Shinde * Antoine Kaufmann Kornilios Kourtis * Timothy Roscoe
Systems Group, ETH Zurich

Abstract

NICs are increasingly complex and diverse, offering a wide range of hardware functionality to aid network protocol processing. Harnessing the power of NIC hardware requires the ability to control and reason about a variety of different feature sets in the network stack. Towards this goal, we propose Unicorn, a language for describing modern NICs. Unicorn offers a simple set of abstractions for modeling both NIC functionality and the state of a protocol stack. To evaluate its expressivity and potential, we present a non-trivial model for the Intel i82599 10GbE NIC, and an algorithm that uses graph embedding to optimize the use of NIC hardware in the network stack.

1. Introduction

We present Unicorn, a language for describing the capabilities of modern network controllers (NICs) which can be used, online, to maximize the extent to which the OS network protocol stack can exploit whatever hardware acceleration the NIC provides.

Unicorn is part of Dragonet [16], a new network stack motivated by two related hardware trends: (i) the growing complexity (and diversity) of modern networking hardware, with increasing functionality provided in a wide variety of ways, and (ii) the increasing parallelism of modern hardware: individual cores are not getting faster, and (with the possible exception of software radios) network protocol processing has limited parallelism. Moreover, with the end of Dennard scaling [2], CPU designers will increasingly turn to specialization and heterogeneity in the search for increased performance, including network processing.

Dragonet addresses the problem of how to best exploit a wide variety of NIC hardware functionality without writing a large quantity of hardware-specific and/or policy-specific C code. It is based on two kinds of dataflow graphs:

A *Logical Protocol Graph* (LPG) captures the state of an OS protocol stack at a point in time at the level of individual connections or flows. An LPG describes the processing that is required for each individual packet which might be sent or received by the system, and can be derived from the current OS protocol stack state.

* Pravin Shinde and Kornilios Kourtis are financially supported in part by Microsoft Research.

PLOS '13, November 03-06 2013, Farmington, PA, USA.

ACM 978-1-4503-2460-1/13/11...$15.00.
http://dx.doi.org/10.1145/2525528.2525532

In contrast, a *Physical Resource Graph* (PRG) explicitly captures the processing functionality available on a given NIC, and is provided in advance by the system developer or (ideally) the hardware vendor. Broadly speaking, Dragonet optimizes use of hardware facilities by applying appropriate embedding policies, such as embedding as much of the PRG as possible into the LPG.

The contribution of this paper is to show that a language based on a small set of simple abstractions (described in Section 3) is sufficient to model the capabilities of sophisticated modern NICs. As an illustrative example, we demonstrate that Unicorn is powerful enough to model the Intel 82599 10GbE adapter (i82599) and we discuss an embedding algorithm that maximizes use of NIC hardware resources in the network stack (Section 4).

2. Background

Unicorn addresses the problem of writing system software to take full advantage of NIC hardware. NICs have been highly complex for some time, however to date the challenge of exploiting this hardware has been addressed by creating a common, fixed set of abstractions covering all available NIC features.

As we argued recently [16], this can lead to rejecting hardware incompatible with the given abstractions. For example, the SYN filter on the Intel 82599 10GbE adapter [5] requires kernel changes to use in Linux, since it does not fit the n-tuple filter abstraction assumed by the rest of the kernel [13]. Other OSes, such as Windows, go to great lengths to provide API abstractions for all available hardware features (e.g. [12]). However, in both approaches, the *policy* of when and how to exploit features must usually be configured manually by application programmers or system administrators, rather than automatically based on workload.

The challenge here is the programming of, and late-binding of functionality to, heterogeneous hardware. However, the problem is different to using heterogeneous *cores*, whether GPUs [7, 14] or others [18]. NICs mostly provide fixed hardware functions rather than programmable cores, and different NIC models, even within a vendor, offer very different features and configuration options[1].

Our approach is to replace fixed abstractions with a language interface, based on two key ideas: (i) expressing both flow-level protocol state and NIC hardware as dataflow graphs (the LPG and PRG respectively), and (ii) reducing the problem of exploiting specialized hardware to graph embedding. Unicorn is a language for specifying the PRG for a given NIC, but the underlying data model it embodies represents both PRGs and LPG. For offline prototyping and testing purposes we use Unicorn syntax to manually write LPGs, though in a real implementation the LPG will be derived from the protocol stack state.

The LPG has two equivalent interpretations. The forward direction (from sources to sinks) represents the individual computations for processing a packet, as in Click [8]. The reverse direction cap-

[1] A few NICs do contain fully-programmable cores, such as [4]. Supporting such devices in a network stack is out of scope of this paper.

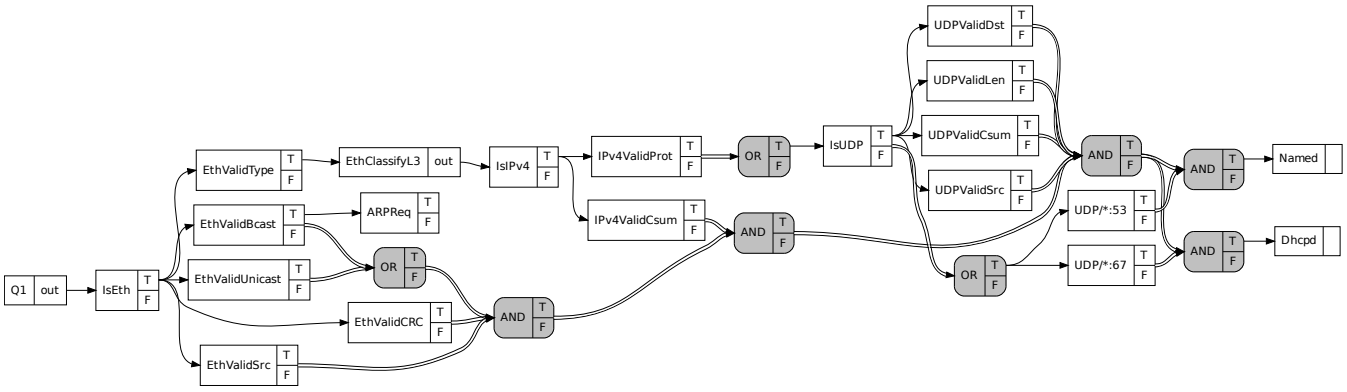

Figure 1: LPG example for receiving network packets.

tures dependencies between computations and is used for determining which NIC features should be used. Dragonet works by maintaining an embedding of the PRG in the LPG at runtime as network connections come and go, by assigning LPG nodes to PRG nodes (NIC fixed functions). The remainder of the LPG is then implemented in software (not necessarily as a dataflow graph).

There is a long tradition of hardware description DSLs for OS development. "Device trees" describing platform configuration are widely used in modern operating systems like Linux, particularly for system-on-chip hardware [9]. Research DSLs like Devil [11] and Termite [15] help automate hardware programming by generating correct-by-construction driver code. Unicorn is complementary: it does not provide a hardware access mechanism, but instead provides a semantic description of the available functionality.

The dataflow model of computation also has a long history, and has recently been applied in parallel programming [1, 6, 14]. Dataflow representations of network processing are used in Click [8] and the x-kernel [3]. Unicorn differs from these systems by having a set of primitive node types which facilitate both automated embedding of a PRG in the LPG, and semantics-preserving transformations on the LPG.

Finally, other novel language approaches to protocol implementation like Melange [10] could be used alongside Unicorn to facilitate implementing the software part of the LPG.

3. UNICORN

Unicorn is a domain specific language for expressing the LPG and PRG so that they can be both easily generated from concise descriptions, and effectively manipulated by appropriate algorithms that implement the embedding and possibly other operations. Our goal is for the Dragonet stack to use representations of the PRG (along with the associated device driver) and LPG internally when placing packet processing functionality at runtime.

The Unicorn language is actually two different concepts: (i) a *concrete syntax* for writing descriptions of NICs, and (ii) an *abstract model* for representing Dragonet dataflow graphs, both PRGs and LPGs. The concrete syntax is intended for only writing PRGs based on descriptions found in NIC vendor documentation, though we also use it to write test LPGs for development and debug purposes. Online, however, the Dragonet protocol stack itself will maintain the LPG as connections come and go. Since there is a close correspondence between the concrete syntax and abstract model, however, we will mostly conflate them in this paper.

The primary challenge motivating Unicorn is NIC diversity. We make only one assumption about NICs: that it is possible to represent their functionality as a dataflow graph that can be matched against the LPG. The names of graph nodes used in a PRG to describe a NIC must correspond to those used to represent an LPG. Contrarily to traditional NIC driver interfaces based on function hooks, LPG allows implementing arbitrary parts of the network stack in the NIC hardware or in the NIC driver.

We build our graph models with a small set of abstractions as basic building blocks. These abstractions are the minimum information required to transform and embed LPGs and PRGs. The core of Unicorn consists of three node types: function nodes (F-nodes), configuration nodes (C-nodes) and logical operator nodes. F-nodes combined with logical operators capture the execution flow and dependencies of protocol processing, while C-nodes capture the configuration options of a NIC.

These basic building blocks are not sufficient to fully model a NIC: for example, reasoning about performance requires more information than the PRG layout. To avoid making assumptions about NICs, we handle these cases by annotating nodes with attributes. Although we expect to build common abstractions for many of these attributes, they are external to the basic node model.

3.1 Logical Protocol Graph

Fig. 1 shows an example (partial) LPG for receiving packets that starts from the NIC hardware queue (`Q1`) on the hardware/software boundary and handles Ethernet, IP, and UDP processing. In this particular example, two applications, `named` and `dhcpd`, are waiting for packets on UDP ports 53 and 67 (depicted by nodes `UDP/*:53` and `UDP/*:67`) respectively. We next discuss how we build our LPG models.

Function nodes (F-nodes) F-nodes are the basic components of our graphs. They represent individual packet computations and are labeled based on the computation they implement. An F-node's outputs are grouped into *ports*. Protocol processing is modeled by each F-node applying a computation to the packet and subsequently enabling a single output port. Enabling an output port, effectively activates a set of F-nodes as defined by the port's edges. For example, the second LPG node is labeled `IsEth` and has two ports: the port labeled `T` (true) has multiple outputs leading to the next stages of packet processing, whereas the port labeled `F` (false) has a single output leading to the packet being dropped, which we omit for brevity.

Logical operators F-nodes have a single input. To handle cases where enabling an F-node might depend on the output of multiple other F-nodes we use *logical operator* nodes (`AND`/`OR`). For each operand, logical operators have two inputs (true and false) and only one of them can be enabled. Hence, each operand typically corresponds to a node and each of the operand's inputs to a different

port of that node. To simplify our model and the graph representations, we assume that only boolean F-nodes, i.e., nodes that have exactly two output ports: `T` and `F`, are connected to logical operators. Additionally, instead of drawing two edges from the boolean node to the logical operator, we draw a single double line. The logical operators have the usual semantics and may be short-circuited. In the LPG of Fig. 1, UDP header field checks (e.g., `UDPValidDst`, `UDPValidSrc`, etc.) are fed to an `AND` node because all fields need to be valid. Conversely, nodes `EthValidBcast` and `EthValidUnicast` are combined using an OR node to check whether this packet has a valid Ethernet destination address.

In our model, packet processing progresses by enabling different F-nodes until a terminal node is reached. At any given time, the state of the packet is defined by the paths of the enabled F-nodes. On the receive side, F-nodes typically inspect the packet to identify its proper destination. On the send side, F-nodes typically build the appropriate packet headers so that the packet can be sent over the network.

Conceptually, each F-node applies a computation to the packet. Since ports might include multiple outputs, our LPG representation defines a partial order rather than a full order for these computations. Hence, there are multiple serialized execution schedules that can be generated from the same LPG, all of which have the same output. We do not reduce the partial order to a full order, so that we are able to match F-nodes to NIC fixed functions regardless of what order the NIC enforces, as long as it is compatible with our partial order. Another benefit is that it allows to model possible performance optimizations such as parallel execution, although in such a case, care must be taken to ensure that concurrent F-node execution does not cause inconsistencies.

Using the receive path as an example, packet data needs to be passed to user-space via a socket. In Fig. 1, two boolean nodes (`UDP/*:53`, `UDP/*:67`) match packets against network flows belonging to specific applications. In practice, these nodes most likely are implemented together as a single lookup table.

In our description so far, the only state we consider is the path followed in the graph and the packet itself. To implement a protocol such as TCP, however, additional state is required. This state might be per network connection (e.g., the TCP control block), or global for the whole interface (e.g., for implementing traffic shaping). We consider new packets generated for processing (e.g., ACKs) a part of this state. The computations that are modeled by F-nodes potentially need to access and modify this state. Our current model assumes that each packet is processed atomically, so the state can be safely accessed by F-nodes as needed. In practice, however, it is important that packets can be processed in parallel to exploit multiple cores. We plan to address this in our implementation by dividing and possibly replicating the state appropriately, allowing efficient and scalable access to it.

3.2 Physical Resource Graph

A PRG can be viewed as a high-level interface between the generic OS protocol stack and the driver code which accesses the NIC hardware. While the LPG captures an abstract view of the packet processing dataflow, the PRG models the fixed-function computations performed by the NIC. We write PRGs based on datasheets, using the same abstractions as for LPGs.

A PRG node P having the same label as an LPG node L implies that P implements L's function and thus L can be mapped to P. A PRG example is shown in Fig. 2.

Ideally, expressing the protocol in a fine-grained manner via the LPG should allow matching every possible hardware function offered by the NIC for this protocol. In practice, however, creating a fully future-proof LPG is very challenging, while defining what is part of the logical protocol state and what is not is ultimately a matter of taste. Thus, we do not assume that every PRG node can be matched against an LPG node. Unknown PRG nodes (those not in the LPG) represent NIC-specific functionality which is not a part of the logical protocol representation.

Configuration nodes (C-nodes) Modern NICs offer rich configuration options that can drastically vary the NIC's behavior. We represent NIC configuration and how it affects the resulting PRG using *C-nodes*. C-nodes are PRG nodes that represent unconfigured parts of the NIC. A C-node is configured by applying configuration values to it, which results in the C-node being replaced by a new set of nodes and edges (i.e., a subgraph). The new edges are restrained by the original C-node edges. For example, if there is no edge between a node n and the C-node, then the new edges cannot include n.

More formally, assuming a graph G (typically the PRG) with vertices $G.v$ and edges $G.e$, a *generic C-node* $x \in G.v$ consists of a configuration space C_x and a function f_x that maps each point in the configuration space $c \in C_x$ to a subgraph Γ. Γ consists of vertices $\Gamma.v$ and edges $\Gamma.e$, and is subject to a number of constrains: First, vertices in $\Gamma.v$ should not already exist in G ($\Gamma.v \cap G.v = \emptyset$). Second, if I_x are the nodes that point to x ($I_x = \{i \in G.v \mid (i, x) \in G.e\}$)[2] and O_x are the nodes pointed by x ($O_x = \{o \in G.v \mid (x, o) \in G.e\}$), then each edge in $\Gamma.e$ should start from a node in either I_x or $\Gamma.v$ and point to a node in either O_x or $\Gamma.v$ ($\forall (j, k) \in \Gamma.e : (j \in I_x \cup \Gamma.v) \wedge (k \in O_x \cup \Gamma.v)$). When applying a configuration c to a C-node x in a graph G, G changes in two ways: (i) x is removed and vertices $\Gamma.v$ are added to to $G.v$ (ii) x's edges are removed and edges $\Gamma.e$ are added to $G.e$.

In practice, we model most of our configuration nodes using *simple C-nodes*, i.e., nodes that select one of their output ports based on the configuration value. For example, the SYN filter functionality of the i82599 is configured using two simple configuration nodes: `CSynFilter`, that enables or disables the filter, and `CSynOut` that selects an output queue for the filter.

C-nodes aim to address the diversity of modern and future NICs and are intended for algorithms that explore and evaluate different configuration options. C-nodes with a small configuration space (e.g., a single register that enables or disables a NIC feature) can be naively handled by these algorithms using exhaustive search. Exhaustive search, however, is highly inefficient for C-nodes with a very extensive configuration space (e.g., configuration for mapping network flows to hardware queues). These cases typically require additional a-priory knowledge to reduce the search space. When steering network flows to hardware queues, for example, the space can be reduced by only considering filters that match network flows corresponding to active connections.

An alternative approach to C-nodes would be to enforce a common abstraction on the PRG level that simplifies the search in the configuration space. However, the diversity of NICs makes it difficult, if not impossible, to devise a common and future-proof abstraction for the configuration space without substantially sacrificing flexibility. Instead, we argue for exposing the full NIC configuration space on the PRG. This allows implementing NIC-specific policies that can exploit all capabilities of a NIC, but does not exclude building common abstractions on top of the low-level PRG interface in a similar manner as common NIC drivers implement an OS-specific interface.

3.3 Attributes

The abstractions we have presented so far capture protocol processing in terms of individual computations. As we show in Section 4, these abstractions are sufficient for operations like the embedding that maximizes the use of hardware resources in the NIC. More ad-

[2] for simplicity, we ignore that in actuality our edges originate from ports rather than vertices.

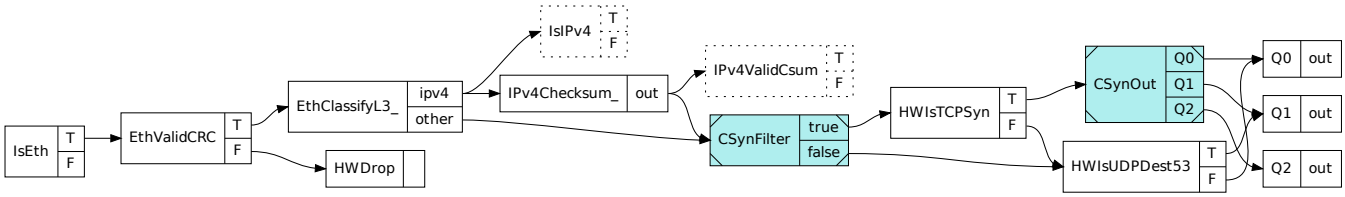

Figure 2: A partial PRG, modeling the receive side of the Intel i82599 10GbE adapter.

vanced reasoning (such as queuing models for performance), however, requires more information which we add by annotating nodes with attributes. As Unicorn matures, we expect to define common abstractions for many of these attributes. In the next paragraphs, we discuss some of the use-cases.

F-node implementation F-nodes in their pure form are not associated with a specific implementation. This leads to ambiguity when trying to model graph execution that includes both hardware and software F-nodes because it is impossible to distinguish between them. Hence, we annotate F-nodes with an implementation tag. For instance, an F-node for checksum calculation can be annotated to depict an implementation in NIC hardware, in the driver, or even in a hardware accelerator.

Modeling performance An important Dragonet goal is to enable reasoning about network stack performance, by annotating nodes with metrics such as CPU cycles for predicting CPU utilization or latency costs for predicting the latency of processing a packet. Additionally, attributes can be used for modeling F-nodes that maintain queues of packets, such as TCP segmentation offload implementations in hardware (e.g., TSO) or in software (e.g., GSO in the Linux network stack). These nodes offer trade-offs between latency, throughput, and CPU utilization. Subsequently, reasoning about performance requires the ability to describe how these nodes operate (e.g., under what conditions they forward the packet instead of storing it).

Protection Traditional NICs perform unprotected DMA transfers, so the OS must mediate all application send/receive operations and copy data between user-space buffers and those used for DMA. Advanced NICs that target high-performance [17] offer protected DMA operations using IOMMUs that allow bypassing the OS and zero-copy communication. Annotating software/NIC boundaries accordingly can allow Dragonet to adapt to such functionality where it is supported by the NIC.

3.4 A first Unicorn implementation

We have prototyped Unicorn using Haskell's QuasiQuotes as an embedded DSL in Haskell. This is highly convenient for trying out LPG models for protocol processing, PRG models for NICs, and embedding algorithms. We also use the same setup to implement network processing using a simulator that processes packets by executing the graph. Dragonet will reimplement the finalized abstract model to achieve acceptable performance (e.g., using C).

An example is shown in Fig. 3. Most model objects are defined using a keyword followed by an identifier and a body in braces. Keywords have the expected semantics (e.g., `node` defines F-nodes, `config` defines C-nodes, and `graph` defines graphs). Outputs are defined using ports that connect to a list of nodes. For example, port `ipv4` of node `EthClassifyL3_` connects to two nodes: `IsIPv4` and `IPv4Checksum_`. Attributes are defined using the `attr` keyword. An explanation of `IsIPv4`'s software attribute is given in §4.

Simple C-nodes are defined using a set of ports that forms their configuration space. Configuring a simple C-node statically defines a port for the node's output. For example, `CSynOutput` can

```
1  graph prg {
2      node HWDrop { }
3
4      node EthClassifyL3_ {
5          port ipv4[IsIPv4 IPv4Checksum_]
6          port other[.CSynFilter] }
7
8      boolean IsIPv4 {
9          attr "software"
10         port true[]
11         port false[] }
12
13     config CSynOutput {
14         port q0[Q0]
15         port q1[Q1]
16         port q2[Q2] }
17 }
```

Figure 3: Unicorn snippet for PRG shown in Fig. 2.

be configured with `q0` to select the first queue. Generic C-nodes are implemented using an additional Haskell function that implements the functionality of f_x as described in §3.2.

The constructs discussed above constitute the basic core of the language. We are currently adding syntactic sugar for avoiding boilerplate code, support for more intuitive error messages, and generally simplifying programming. For example, our prototype supports boolean nodes, defined using `boolean` instead of `node`. Boolean nodes are constrained to a specific structure: they are expected to have exactly two output ports: `true` and `false`.

4. Modeling the i82599 with UNICORN

We are currently evaluating Unicorn by modeling different NICs and experimenting with embedding algorithms. Here we discuss modeling the i82599 NIC [5]. A part of i82599's receive path PRG is shown in Fig. 2. We focus our discussion on two interesting features of i82599 from a modeling perspective: hardware checksum calculation and hardware queues.

Hardware checksum calculation The i82599 supports hardware checksum calculation for a number of different protocols (e.g., Ethernet, IPv4 and TCP). However, modeling complications arise. On the receive side, for example, the NIC supports classifying IPv4 packets and verifying the IPv4 checksum. The results of these computations are stored in the descriptor passed to the network stack, making a minimal amount of software processing necessary. To handle these cases, we add software nodes in the PRG. This allows expressing dependencies from PRG nodes to software nodes, capturing any additional device-specific software functionality required from the driver. Hence, the `IsIPv4` and `IPv4ValidCsum` in i82599's PRG are software nodes that model the checking of specific flags in the descriptor, rather than the full check. We denote software PRG nodes with dashed boxes in our graphs.

Hardware queues Hardware queues are a standard feature of modern NICs. They provide multiple receive and transmit descrip-

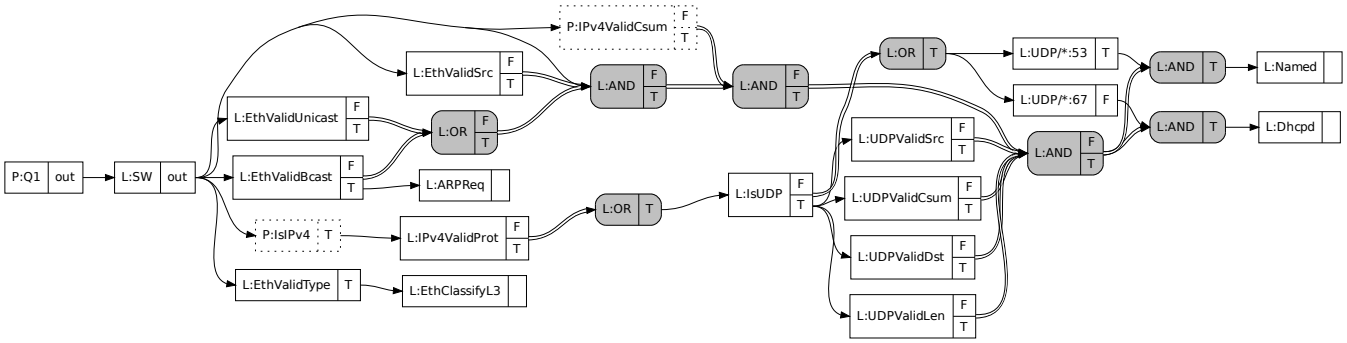

Figure 4: Software nodes of an embedded graph that results from embedding queue `Q1` of the PRG shown in Fig. 2 to the LPG shown in Fig. 1. The SYN filter is assumed disabled.

tor queues to the OS, as well as facilities for steering packets into them (filters). Queues expose multiple instances of the NIC to the OS and are typically used to improve the scalability of the network stack with techniques such as RSS, or to improve quality of service for specific network flows. There are multiple different filter mechanisms and each NIC offers its own variant.

The i82599 provides a wide range of different filters, each with possibly different configuration options. Two examples are 5-tuple and SYN filters. The former are specified by a 5-tuple that includes the protocol, the source and destination IP address, and the source and destination port. Any field in the tuple can be masked. SYN filters, when configured, match TCP packets whose SYN flag is set. In Fig. 2, a 5-tuple filter is configured to match UDP packets with a destination port equal to 53 (`HWIsUDPDest53`) and forward them to `Q1`. Currently, we model 5-tuple filters using generic C-nodes. The ordering in which the filters are applied needs to be encoded in the PRG so that the resulting model is valid. In our example, the SYN filter will be matched before matching the 5-tuple filters.

In the presence of multiple queues, the network stack is responsible for multiplexing and demultiplexing packets between the queues and the application network flows. In the general case, all flows can interact with all queues. The explicit descriptions provided by the Unicorn models allow Dragonet to reason about which flows interact with which queues and specialize the network stack accordingly. A simple, yet common example is exclusively mapping a network flow to a queue for providing quality of service (QoS).

Reasoning about how flows are mapped to queues requires interactions between LPG nodes that represent flows and PRG nodes that represent filters. Our approach is based on calculating constraints for LPG flow nodes, by examining the set of node/port combinations that dominate each flow node. Constraints act on LPG flow nodes by constraining the set of output ports that can be enabled. As the graphs become complicated (e.g., multiple paths reaching a hardware queue via an `OR` node), reasoning about flows and queues becomes challenging and computationally unaffordable. Nevertheless, we believe that there are cases that do not fall into this category and can benefit from such an analysis (e.g., when an a queue is configured to serve a single network flow).

An additional difficulty arises because a number of hardware features can be configured on a per-queue basis. This means that the parts of protocol processing done by the hardware and with it the configuration of software protocol processing varies across queues. Also knowledge about the possible flows on a queue can allow simplification of protocol processing (e.g., assuming a queue only receives UDP packets, TCP protocol processing can be dropped). We deal with this by treating each queue in isolation for the embedding.

4.1 Embedding

Ultimately, Unicorn models are to be used by the Dragonet network stack to configure and operate NICs. This requires determining what NIC hardware functionality is used and how hardware and software cooperate. We reduce this problem to embedding the PRG into the LPG. The embedded graph includes PRG and LPG nodes, but for any label only one node is included. All LPG labels exist in the embedded graph, and all LPG/PRG dependencies are respected. To enable for adaptation to the flows of each queue, we perform a separate embedding for each queue.

Next, we discuss a basic embedding algorithm that tries to maximize use of PRG nodes, assuming a fully configured NIC. We start by adding to the embedded graph the desired NIC queue and all its dependencies, which guarantees that we use as much of the PRG as possible. Then we iteratively pick nodes from the set of unembedded LPG nodes (U) so that all their dependencies are already embedded. If LPG is acyclic (which is currently the case in our models), then either U is empty or such a node exists. When the embedded graph is produced, we need to determine a single point for the software/hardware boundary. We use a node (`SW`) to define that point, which we place right after the PRG queue node. Subsequently, all PRG software nodes are moved beyond this boundary.

Figure 4 shows the resulting graph's software nodes when embedding queue `Q1` of the PRG shown in Fig. 2 to the LPG shown in Fig. 1, assuming the SYN filter is disabled. Nodes starting with `L` are LPG nodes and nodes starting with `P` are PRG nodes. If the SYN filter is disabled, `Q1` in the PRG is dominated by a 5-tuple filter for UDP packets to port 53. Applying the 5-tuple filter constrains to the two UDP flows in the LPG, reveals that only one flow is assigned to `Q1`. Although this is a simple example, we argue that it illustrates the potential benefits of our proposed PRG/LPG models.

5. Conclusions and Future work

In this paper we presented Unicorn, a language for modeling NICs. Based on its primitives, we built a model for the i82599 NIC and discussed how we can manipulate this model for controlling and reasoning about NIC hardware resources.

We believe that Unicorn is a first step towards implementing a network stack that can fully exploit modern NICs. There are, however, many unresolved challenges. Our future work will focus on implementing: (i) an efficient network stack that can be controlled by an (incremental and online) embedding of the PRG(s) into the LPG, and (ii) the corresponding NIC drivers.

References

[1] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, Feb. 2013. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/2408776.2408797.

[3] N. C. Hutchinson and L. L. Peterson. The X-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991. ISSN 0098-5589. . URL http://dx.doi.org/10.1109/32.67579.

[4] Intel Corporation. *Intel IXP2400/IXP2800 Network Processor Programmer's Reference Manual*, November 2003.

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

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. . URL http://doi.acm.org/10.1145/1272996.1273005.

[7] *The OpenCL Specification, Version 1.2*. Khronos Group, 2012.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3), August 2000. ISSN 0734-2071. . URL http://doi.acm.org/10.1145/354871.354874.

[9] G. Likely and J. Boyer. A symphony of flavours: Using the device tree to describe embedded hardware. In *Proceedings of the Linux Symposium*, volume 2, pages 27–37, 2008.

[10] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" internet. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, Lisbon, Portugal, 2007. ISBN 978-1-59593-636-3. . URL http://doi.acm.org/10.1145/1272996.1273009.

[11] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: an IDL for hardware programming. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, 2000.

[12] Microsoft Corporation. Information about the TCP chimney offload, receive side scaling, and network direct memory access features in windows server 2008. Microsoft Knowledge Base article 951037, http://support.microsoft.com/kb/951037, revision 7.0, February 2011.

[13] mle1000-syn. Discussion in the e1000-devel mailing list entitled 'Configure Rx ntuple filters'. http://article.gmane.org/gmane.linux.drivers.e1000.devel/10140, June 2012.

[14] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011. ISBN 978-1-4503-0977-6. . URL http://doi.acm.org/10.1145/2043556.2043579.

[15] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with termite. In *22nd ACM Symposium on Operating Systems Principles*, pages 73–86, Oct. 2009. ISBN 978-1-60558-752-3. .

[16] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We need to talk about NICs. In *14th Workshop on Hot Topics in Operating Systems*, May 2013.

[17] Solarflare Communications, Inc. *Solarflare SFN5122F Dual-Port 10GbE Enterprise Server Adapter*, 2010. URL http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_SFN5122F_10GbE_Adapter_Brief.pdf.

[18] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–188, 2008. ISBN 978-1-59593-958-6. .