Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 125

Systems Group, Department of Computer Science, ETH Zurich

Efficiently executing the Dragonet network stack

by

Antoine Kaufmann

Supervised by

Prof. Timothy Roscoe
Dr. Kornilios Kourtis

March 2014 – September 2014

Informatik
Computer Science

Abstract

Today's network cards are getting more and more complex. One reason for this is that networks are still getting faster, while cores are not. Two common approaches for allowing network stacks to keep up, are distributing packets to multiple cores directly in the network card, and moving protocol processing fully or partially to the network card. However current network stacks were not designed to accommodate the varied sets of hardware features supported by different network cards, resulting in sub-optimal performance.

This thesis discusses our approach of building a network stack based on modeling both the required network processing and the network card as dataflow graphs and then combining these graphs to arrive at a configuration for the network card and a description of what processing needs to be implemented in software. It extends on our earlier publications introducing the modelling based network stack approach, and discusses Dragonet, our implementation of a full network stack and its performance characteristics. Performance of the resulting implementation is evaluated and compared to Linux using multiple throughput and latency benchmarks.

Our results showed that Dragonet is capable of providing applications with competitive and often superior throughput and latency, compared to the widely used Linux network stack.

Acknowledgements

I would like to thank Prof. Timothy Roscoe for his support, advice, and for getting me started working on research. This thesis would also not have been possible without the mentoring and collaboration of Dr. Kornilios Kourtis and Pravin Shinde, and I would like to thank both of them for our many interesting discussions.

Both Kornilios and Pravin were closely involved in developing Dragonet. In particular Kornilios developed the LLVM-based execution engine for protocol graphs discussed below. Pravin ported `memcached` to Dragonet, ran the throughput benchmarks, and worked on the implementation of various other parts of Dragonet.

Next I would like to thank Dr. Adrian Schüpbach, who advised me during my bachelor thesis, and helped getting me interested in research. I am also grateful to the Barrelfish team and the Systems Group at ETH in general for providing an interesting environment to work in.

My special thanks go to my family for supporting me, and also for providing welcome distractions when needed, making even the most stressful times bearable. In addition I would also like to thank my roommates and friends Reto Wyss and Dominic Schweizer for putting up with me even when I was consumed by work. Finally I would also like to thank my friend Tobias Feller for our many interesting discussions, and for always being up for a chat.

Contents

Chapter 1

Introduction

Given the still ongoing trend towards increasingly distributed applications, network stack performance is an important factor influencing overall application performance. The fact that NICs are still getting faster while CPU cores are not also implies that the time spent in the software part of the network stack will be more significant relative to the overall time spent on the network. One response to this has been the development of various offload features, that allow part of the network processing to be moved directly to the NIC, freeing up CPU cycles and possibly allowing for faster implementations in hardware. Examples of this are checksumming offloads at various layers and TCP protocol offloads[7].

In practice there are however some limitations, some inherent to the offload feature and others specific to particular implementations [29]. The wide range of different offload features and small differences between implementations of specific features also make implementation in a network stack harder. This has lead to limited support for these features in network stacks such as in Linux [38].

Where such features are supported, they are often implemented in an ad-hoc manner inside a NIC driver. One problem with this approach is that the missing integration with the rest of the network stack often means that not enough information will be available to allow an efficient implementation [30]. In addition these ad-hoc solutions also make configuration more complicated since there is no central instance governing the policy of when to use what features, meaning that some options need to be configured when loading the drivers, others are run-time configuration parameters for the network stack, and others still can be configured on a per-socket basis using `ioctl()`.

1.1 Dragonet – The Vision

For our Dragonet project we argue that the fact that a specific offload feature might not be appropriate for all scenarios, does not imply that the feature should simply be discarded, but the network stack should be able to use it in cases where the limitations are not an issue [34]. One of the main goals is to provide a network stack that makes it easy to fully integrate a wide variety of hardware offloads, while making related policy decisions, such as when to use a particular offload, in a centralized manner.

To this end, our approach is based on structuring the whole network stack as a dataflow graph. The concrete graph used for a particular instance of the stack is derived by combining a graph-based description of the NIC, known as the physical resource graph or PRG, with a graph-based description of the network state and protocol processing that needs to be performed, called the logical protocol graph or LPG. Both of these graphs are expressed using the same semantics and will be combined to arrive at a graph describing what protocol processing will be implemented in hardware and what has to be done in software [33].

1.2 Project Context

This master thesis was realized as part of the Dragonet project in close collaboration with Pravin Shinde, Kornilios Kourtis, and Timothy Roscoe. The main goal for the implementation part has been to show that an implementation of a network stack based on our graph model can provide performance that is comparable to or better than the performance delivered by existing network stacks. This involved tasks such as porting/implementing drivers, implementing high-level reasoning parts in Haskell, as well as developing and benchmarking an efficient implementation for the data path in Dragonet. While I was involved in most of these tasks, because of time constraints this thesis focusses on the design and implementation of the data path, i.e. the question of how to execute a given protocol graph efficiently.

1.2.1 Related Publications

The following two papers about the Dragonet project have been published at the time of writing, and form the foundation for this master thesis:

- Pravin Shinde, Antoine Kaufmann, Kornilios Kourtis, and Timothy Roscoe. Modeling NICs with Unicorn. In *7th Workshop on Programming Languages and Operating Systems*, 2013.
- Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *14th Workshop on Hot Topics in Operating Systems*, 2013.

1.3 Related Work

This section will discuss related work on graph-based network processing and network stack specialization. Other related work will be referenced throughout the thesis.

1.3.1 Graph-based Processing

x-Kernel [24] is based on the idea of using a more structured approach for implementing a network stack and generally providing a framework for protocol implementation. They also show that their structured approach provides performance comparable to contemporary less structured network stacks. An x-Kernel configuration is described as a graph of protocols where edges denote dependencies, and protocol processing is implemented as messages being processed by a series of protocols. Note that the configuration is fixed at compile time, which is one of the main differences to Dragonet. In addition, x-Kernel uses much more coarse grained nodes, i.e. protocols, while nodes in Dragonet are usually fine grained operations such as calculating a checksum or checking a protocol field for a specific value. The graph in x-Kernel is also used only for initialization purposes, in contrast to Dragonet where the graphs are used as a basis for reasoning at runtime and for guiding execution.

Click [26] applies the idea of using a graph to do processing on network packets to implement routers and other packet processing applications such as firewalls. A graph in Click is composed of elements that communicate with other elements by receiving packets on their input ports, and send packets to other nodes on their output ports. This graph will be instantiated at runtime as a data structure of element objects, which is used for execution. Hot swapping even allows Click configurations to be replaced online without losing state, but not fast enough to allow changes for appearing and disappearing flows. The main difference to Dragonet is that Click is used for packet processing engines, and not intended as a network stack interfacing with applications. In addition the graphs are not used for reasoning but for execution, and the graphs in Click are developed with execution in mind, while Dragonet graphs are designed to allow a wide range of offload features to be exploited. There are also some differences in the execution model, e.g. Dragonet nodes do not have input ports, and all output ports are push ports, and in general Dragonet semantics are more limited by design, to allow for easier reasoning.

1.3.2 Network Stack Specialization

Marinos, Watson, and Handley [27] recently explored the idea of using specialized network stacks in a server setting with the goal of improving networking performance. Their approach is based on using a user space network stack which is implemented as a set of libraries, and communicates with the network using the netmap[32]. In addition to moving they implement a number of other optimizations such as zero copy I/O, avoiding the POSIX API, batching of packets when sending to amortize overhead, and the use of a synchronous interface to the stack driven by received packets. In the two applications developed to interface with the specialized network stack, a web and a DNS server, performance improvements of 2-10x were shown, when compared to running corresponding servers on Linux and FreeBSD.

In the context of embedded systems, and especially wireless networking devices, work is being done towards specializing network stacks to particular applications or even specific conditions at runtime [12, 21]. For embedded systems performance is often secondary to other metrics such as memory footprint or power consumption, thus leading to different optimization goals and motivations than for conventional network stacks. In the context of wireless networking devices such as cellphones, there is often a large number of different protocols being used over time during runtime. Running unneeded protocols leads to wasted resources, which motivates the desire for network stacks that can be specialized at runtime.

1.4 Outline

Chapter 2 will start off with a description of the model used, the processes involved in getting to a graph to be executed, and the control plane in general. After that, chapter 3 will discuss the main focus of this thesis: the data plane. The performance of the resulting implementation is discussed in chapter 4, which is followed by the conclusion and some suggestions for future work in chapter 5.

Chapter 2

Dragonet

As discussed in the introduction, the main idea behind Dragonet is to represent the required packet processing, including network stack state, as well as the capabilities of the NIC as a dataflow graph, termed the logical protocol graph or LPG for the former and physical resource graph or PRG for the latter, both using the same graph semantics (see 2.2). Both of these graphs are then combined (see 2.3) to decide what parts of packet processing can be done in hardware, and what parts need to be done in software. Note that most hardware features provided by NICs, such as checksumming or demultiplexing, can be configured to be enabled or disabled, which means that there will generally be more than one way to combine the graphs. Depending on the policy goals to be implemented, some combined graphs will be more desirable than others, which we decide based on a cost function (see 2.4.1) that is provided as an input and will assign a cost to the resulting combined graph. Now we are basically left with an optimization problem (see 2.4) of finding the graph that results in the minimal cost, which we implement based on a hardware specific oracle (see 2.4.2), that provides a subset of hardware configurations to evaluate. This optimization process will result in the optimal (or close) combined graph and the hardware configuration it is based on. As a next step, this graph will be used to instantiate the software processing part of the network stack, that we termed the data plane (see 3). Dragonet also has an application interface, that is both, a part of the data plane for sending and receiving packets, but also part of the control plane for control operations such as opening and closing sockets. Control operations from the application (see 3.6), such as opening a new listening socket, will result in changes to the network stack state, and thereby the LPG, which in turn implies changes in the resulting graph when going through the optimization process, thus requiring the optimization process to be applied to the new graphs.

Figure 2.1 shows an overview of this whole process.

2.1 Architecture

Architecturally Dragonet can be split up into two parts: The *control plane* decides how the network stack should be instantiated by choosing a graph to be implemented, and modifies it in reaction to external events. Packet processing, i.e. the critical path for sending and receiving data at runtime, is implemented

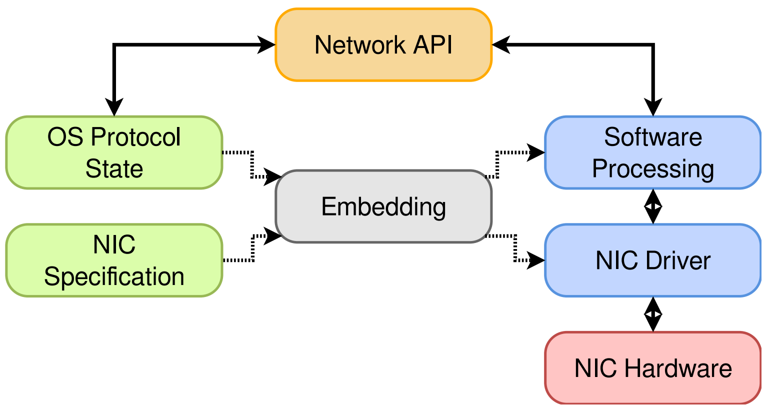

Figure 2.1: Dragonet: A conceptual overview

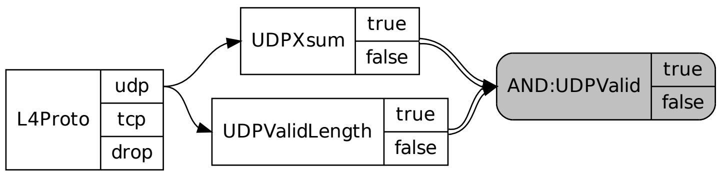

Figure 2.2: Simplified example of a protocol graph

in the *data plane*. The control plane is described in the following sections, while more details about the data plane are provided in chapter 3.

2.2 The Model

At the center of Dragonet there is our dataflow graph based model for describing protocol processing and hardware capabilities alike as so-called *protocol graphs*. The current model is an extension of our initial model presented in our PLOS'2013 paper by extending it with a task-based execution model, where the original model was purely dataflow-based. Both, the original model and our extensions for the current version will be described below, along with the rationale for the extensions. There is also a description of our domain specific language *Unicorn* used for writing protocol graphs.

2.2.1 Pure Dataflow

In our original model a protocol graph consists of nodes that are connected by directed dataflow edges, as shown in figure 2.2. No cycles are allowed in protocol graphs, which significantly simplifies reasoning. A node has a single input port and can have multiple output ports, of which at most one will be enabled during execution. An arbitrary number of edges can originate at an output port and thereby connect to an arbitrary number of successor nodes.

In the forward direction edges specify the order used during execution, where they can be enabled for a particular packet when the source node is done executing and enables the corresponding output port, thereby enabling the destination node of the edge. But edges can also be interpreted backwards, in

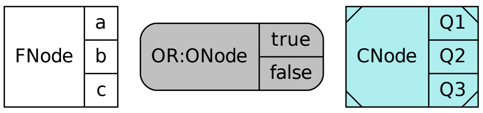

Figure 2.3: The graphical representation of the three different node types

which case they can be thought of as dependencies, i.e. if node `A` should be executed some/all of these nodes need to have been executed.

Node Types

There are three different types of nodes as shown in figure 2.3:

- *F-Nodes:* Implement basic protocol processing operations, such as calculating a checksum, or writing a header field. Can have *at most one incoming edge*. The actual functionality of the node is implemented separately in an implementation function (see 3.2.1). An F-node will enable exactly one output port if it is enabled.

- *O-Nodes:* Are logical operators used to implement control flow. Basic n-ary logical operators: `AND`, `OR`, `NAND`, and `NOR`. Note that `NOT` is missing on purpose, since it can be implemented using a `NAND` node, and because it cannot be generalized to multiple incoming edges and would therefore introduce an unnecessary special case. O-nodes have exactly the two output ports `false` and `true`. All edges ending at O-nodes must originate from a port labeled `true` or `false`, which will be used as the operand for implementing the logical operation. During execution short-circuiting can occur, e.g. once an `OR` node is enabled through a port labeled `true` for the first time, its `true` port will be enabled, regardless of whether all predecessor nodes have been enabled. Regardless of whether short-circuiting occurs the successor nodes will only be enabled once.

- *C-Nodes:* These nodes are not involved in execution, but represent configuration parameters. They can have an arbitrary number of incoming edges and output ports.

  When applying a configuration for a C-node it will be replaced by a subgraph without C-nodes. The subgraph used to replace a C-node can only be connected to the rest of the graph through the incoming and outgoing edges of the original C-node. Meaning that for each edge connecting the subgraph to the rest of the graph, there has to be an edge between the C-node and the respective source or destination node of the original graph. This limitation is intended to keep changes by the C-node local to the node, and makes it possible to get at least a conservative estimate of how the configuration will modify the graph without knowing how the C-node is implemented. The function implementing a C-node, i.e. mapping a specific configuration value to the respective subgraph, is currently implemented separately in Haskell.

  A protocol graph containing C-nodes is called not fully configured and cannot be executed.

Execution

A protocol graph is executed on a per-packet basis. Execution is governed by a loop over all entry nodes in the graph, i.e. nodes without incoming edges, where each node will be enabled one after another with an empty buffer each. These source nodes usually poll some kind of queue and usually have a port without outgoing edges that will be enabled if polling was unsuccessful, and one port to continue processing. The graph is executed by keeping a set of nodes that have been enabled but have not been executed yet. Execution proceeds by removing a node from the set and executing it on the current buffer, which can potentially modify the buffer, and subsequently adding the successor nodes connected to the output port enabled by the execution into the set. Once the set is empty execution is done and the buffer will be zeroed out.

Note that while execution is based on executing nodes on a buffer, nodes can replace the actual buffer by modifying the buffer structure they are being passed. This happens e.g. if a buffer is passed to an application, since this will result in a temporary loss of the buffer's ownership. The only requirement is that the buffer be replaced so that the buffer structure is valid after executing a node.

2.2.2 Issues with Pure Dataflow Model

While the purely dataflow-based execution model was sufficient to implement a first prototype of a network stack based on this model and get some performance numbers, there were a number of issues that were difficult to address using the model.

Starting Multiple Executions from a Node

The first issue was caused by the limitation that a node can only enable a single port and pass the current buffer, although the buffer could be replaced this still only allowed for one buffer to be passed. A number of situations requires a variable number of of executions to be started from a certain node. This already became apparent when implementing ARP: When sending an IP packet to a destination for which the MAC address is not known, an ARP request has to be sent, and the IP packet can only be sent once the corresponding ARP response is received. Now for a single waiting packet this is not a problem, since we can simply replace the buffer of the ARP response with the one from the IP packet and continue sending this one. The problem occurs if there is more than one IP packet that is waiting for the same ARP response, here the execution for each of the waiting IP packets needs to be resumed. This issue occurs in multiple other situations as well, such as for example when segmenting TCP data to be sent, or when fragmenting IP packets.

One workaround for this that does not require any changes to the execution model would be to add a queue that will be polled by an entry node, where these IP packets would be enqueued when receiving the ARP response, and then continuing execution one at a time when polling the queue. Note that if this workaround is used, there will be no information in the graph about this coupling between the nodes, and knowledge about the node implementations is required to understand how these nodes interact.

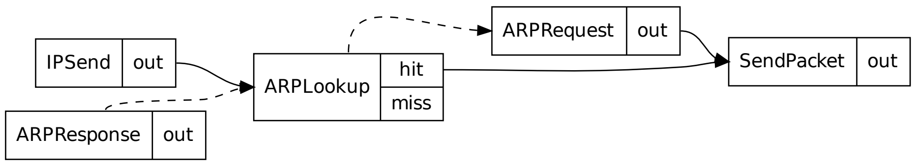

Figure 2.4: Basic example of a protocol graph using the task-based model.

Replacing Buffers

Another more general problem with replacing the buffer of an execution is that it makes reasoning more difficult, since replacing the current buffer will invalidate any knowledge that could be collected by just looking at the nodes in path leading through such a node. And especially when not substituting a fresh buffer, such as in the case when resuming an IP packet to be sent, knowledge about the buffer might enable significant optimizations.

Valid Buffer Requirement

We also ran into issues with the requirement of always having a valid current buffer during an execution. This turned out to be problematic when packets are being enqueued, be it on a send queue of a NIC or to be passed on to an application. In these cases the current buffer becomes unavailable for the execution, and since the execution model always requires a valid buffer to operate on, a new buffer needs to be allocated by the node. Now in most of these cases graph execution stops there, since all the processing will need to have been performed before adding it to the queue, if this would not be the case not all dependencies had been specified in the graph, so the allocation is basically only required since the next execution will reuse the current buffer. If the allocation of a replacement buffer fails, the current buffer can then not be enqueued since this would violate the valid buffer requirement, even though the buffer is not actually required. Also most entry nodes, especially the performance critical ones, tend to pick up packets from a queue, be it hardware or software, and will therefore already have a buffer which they then exchange with the current buffer and then free the unused one.

2.2.3 Task-Based

The problems discussed above lead us to revise and extend our model, resulting in a task-based execution model.

Spawn Edges

As a major change, an additional edge type is introduced: the so-called *spawn edge*. In contrast to a regular dataflow edge, a spawn edge does not originate at a port. A spawn edges basically indicates that a node can spawn a new task starting execution at the destination node of the edge. Multiple spawn edges can originate from a single node. During execution each spawn edge can be used to spawn an arbitrary number of tasks, including none at all.

Another major difference to dataflow edges is that spawn edges can be used to introduce cycles. This also includes the special case of a spawn edge with the same source and destination node, which is used for nodes that do polling allowing them to re-spawn themselves for the next polling iteration.

Spawning Tasks

In order to have a handle to refer to a specific outgoing spawn edge when spawning a task in the implementation, a label uniquely identifying the spawn edge on the source node is used. A new task can then be spawned in the node implementation by providing the label identifying the edge and thereby the start node for the task, optionally a buffer to start execution with, as well as a priority that basically controls what end of the task queue the new task is added to. Currently only two priorities, `LOW` and `HIGH`, are used. The former is used for tasks starting at nodes that do polling, and the latter for all other tasks.

Execution

Instead of looping over entry nodes, execution is based on a task queue. A task in the queue consists of a node to start execution at, and optionally a buffer to start execution with. The graph is then executed by taking a task off the task queue, and executing it to completion, as a result of which additional tasks might be enqueued on the task queue, and then continuing with the next task. Tasks in our model will always run to completion, and there is no preemption involved. If a spawn edge is used to spawn a new task, the new task will be added to the task queue and the current task continues executing.

In order to bootstrap execution of a graph, a number of nodes will be tagged as initialization nodes. For each initialization node, a task will be added to the task queue without a buffer as a parameter when preparing to execute the graph. Execution then progresses from there spawning additional tasks. Currently the task queue will never become empty during regular execution, since an empty task queue basically stops execution since tasks can only be spawned by other tasks.

Buffer Handling

The other major change from the pure dataflow model is about how to deal with buffers when starting and terminating tasks. Instead of always starting off execution with a valid buffer, the buffer provided in the task queue entry is used, which removes the requirement to the runtime to allocate a buffer to start execution whether a buffer is needed or not. When a task terminates and current buffer pointer still points to a valid buffer after the last node is executed, the buffer will be freed by the runtime.

In addition we are introducing the constraint that a single task should not access more than one buffer. For tasks started with a valid buffer this means that the buffer must not be replaced with another buffer, but it can be removed i.e. set to `NULL`. The latter is used if the buffer should not be freed e.g. if it has been enqueued and passed on to an application. In general this leads to cleaner semantics for graph execution, where a single execution of the graph can now be thought of as a *buffer path*, basically describing operations performed on a specific buffer.

2.2.4 Future Extensions

We discussed a number of extensions to the task-based model described above, that were not implemented due to time constraints. The following paragraphs provide a short description of these extensions.

Blocking Tasks

In a number of cases it would be desirable to have the ability to block tasks until a notification from another tasks arrives. One example is sending an IP packet where an ARP cache miss occurs, in which case sending the packet has to be delayed until the corresponding ARP reply arrives. Currently this is achieved by storing the buffer containing the IP packet in a global data structure of packets waiting for ARP responses, then spawning a new task to send an ARP request, and then just terminating execution of the IP send task. When the ARP response arrives, it will scan the data structure for packets awaiting ARP information for the IP address specified in the response, and then spawn a new task for each of the pending buffers.

While this solution works in practice, it is not very satisfying. The main problem is that there is no link in the model between the two tasks processing the same IP packet.

Our suggested extension to work around this problem is to allow tasks to explicitly block on certain conditions and then allow other tasks to notify blocked tasks. This would remove the requirement to split up the send task, and remove the need to work around the execution model. Notifications would also be captured in our model by adding a third edge type, the *notification edge*, and allowing notifications to be sent only along these edges.

Deferred Spawns

In addition to waiting for notifications, some use-cases require the ability to only start execution of a task after a certain delay, or in response to external events. The implementation of timeouts in a number of protocols, such as when waiting for ARP responses, is an example of this. Another example would be a task picking up packets from a queue that is not polled, in which case the external event could be data that is ready to be read on a file descriptor.

The interface for these deferred spawns consists of the same parameters required for a regular task spawn operation, plus a number of parameters specific to the type of deferred spawn, such as a delay or a file descriptor. Until now we identified applications for the following types of deferred spawn operations:

- *Basic timeout*: the task will start execution after a fixed time delay or at a fixed time.
- *File descriptor*: execution starts when data on the specified file descriptor is available to be read or written, depending on parameters.
- *Allocate buffer*: execution starts when a buffer is available to be allocated.

```
graph taskBased {
    node IPSend {
        port out [ARPLookup] }
    node ARPLookup {
        spawn miss ARPRequest
        port hit[SendPacket]
        port miss[] }

    node ARPRequest {
        port out[SendPacket] }

    node ARPResponse {
        spawn resume ARPLookup
        port out [] }

    node SendPacket {
        port out [] }
}
```
Listing 2.1: Unicorn representation for example graph shown in figure 2.4.

2.2.5 Unicorn

With our PLOS'2013 paper we presented a domain specific language, Unicorn, to describe protocol graphs. It provides a convenient way to manage and develop persistent protocol graphs, such as the unconfigured logical protocol graph and physical resource graphs for the various NICs.

Listing 2.1 contains the Unicorn representation for the graph shown in figure 2.4. More details about Unicorn are discussed in our PLOS'2013 paper [33]. Three extensions where made to the Unicorn language since then:

- Expressing spawn edges as already shown in listing 2.1.
- A type description for describing the configuration space for each C-node. This feature is introduced below.
- Specifying node semantics (described below in 2.3.2 in the context of graph simplification).

Configuration Space Types

These types describe a superset of acceptable configuration values for each C-node. More importantly they provide some general information about the structure of the configuration space for the node. Listing 2.2 an example of a configuration node with type information attached. The syntax for expressing types is described in table 2.1.

Not Captured by Unicorn

The following aspects are currently *not* captured by Unicorn:

- Implementation functions for F-nodes. Currently implemented in C.

```
config C5TupleFilter {
    type { (sip:   <UInt 32>,
            dip:   <UInt 32>,
            proto: <Enum (TCP,UDP,SCP,OTHER)>,
            sport: <UInt 16>,
            dport: <UInt 16>,
            prio:   Int 1 7,
            queue:  UInt 2) } <,128>
    function config5tuple
    port queues[Q0Valid Q1Valid Q2Valid Q3Valid]
    port default[CFDirFilter] }
```
Listing 2.2: Unicorn representation including type information of a node from our Intel 82599 PRG.

Syntax	Description
`Bool`	A basic boolean value, i.e. true or false.
`Int` *min-value max-value*	An integer in the specified interval.
`UInt` *bits*	Unsigned integer with the specified number of bits, i.e. a value between 0 and $2^{bits} - 1$.
`SInt` *bits*	Signed 2's complement integer with the specified number of bits.
`Enum (`*ENUM1,ENUM2,...*`)`	Enumeration type, constraining the allowed values to the specified set of labels.
`<`*T*`>`	A value of type `T` or no value, i.e. basically `null`.
`[`*T*`]<`*min-len*`,`*max-len*`>`	An ordered list of values of type `T`. Refer to table 2.2 for the syntax for the length constraints.
`{`*T*`}<`*min-len*`,`*max-len*`>`	Unordered list of values of type `T`, same length constraints as for ordered lists.
`(`*lbl1*: *T1*, *lbl2*: *T2*,...`)`	Product type, i.e. a tuple with the specified fields of the specified types. The labels are only used for informational purposes.
`|<`*lbl1*: *T1*, *lbl2*: *T2*,...`>`	Sum type, a value of either of the specified types. The labels are again purely informational.

Table 2.1: Syntax for expressing types of the configuration values for C-nodes.

Syntax	Description
`<`l`,`u`>`	$l \leq length \leq u$
`<`l`,>`	$l \leq length$
`<,`u`>`	$length \leq u$
`<`x`>`	$length = x$
	No length constraints.

Table 2.2: Syntax for expressing length constraints on list types.

- Configuration functions for C-nodes. Currently implemented in Haskell.

Note that the reason for these aspects to be missing in Unicorn is not that we established that they do not belong there, but rather due to time constraints and since we were looking to gain some experience with the model first to get more insight about what language semantics would be required and appropriate for expressing these in Unicorn. Adding the former could possibly even result in more efficient execution (see the discussion about future work 5.1), while adding the latter should make it easier to implement configuration functions.

2.3 Instantiating a Configuration

The process of generating the graph to be executed given the logical protocol graph, the unconfigured physical resource graph, and a configuration to be applied to the PRG and the hardware alike, consists of a number of steps that are discussed in this section. The first step will be taking the unconfigured PRG and applying the configuration to it, which will result in the configuration nodes being replaced by the subgraphs corresponding to the configuration specified and thereby a fully configured PRG, which describes the behavior of the NIC under the specified configuration. Afterwards the logical protocol graph will be embedded into the configured PRG, resulting in a graph containing information about what parts of the network processing specified in the LPG can be performed in hardware. The next step will be a simplification pass which will eliminate unreachable nodes and edges. A number of other small modification passes will be applied to the graph, after which the graph will be partitioned based on where each node will be executed, such as inside Dragonet on a particular core, or inside a particular application process. The resulting graph can then either be used to evaluate it when comparing different configurations, or to execute it as discussed in the next chapter.

2.3.1 Embedding

We proposed an initial embedding algorithm for handling the receive side in our PLOS'2013 paper. Extending this to the send side turned out to be more complicated than expected. Because of this the implementation is currently based on a dummy embedding that is not able to push nodes into hardware, but will however be able to use multiple send and receive queues. Figure 2.5 shows a minimal example of the dummy embedding. The dummy embedding algorithm basically works by connecting each send and receive queue pair to a separate copy of the LPG. Note that the simplification pass will be able to figure out which

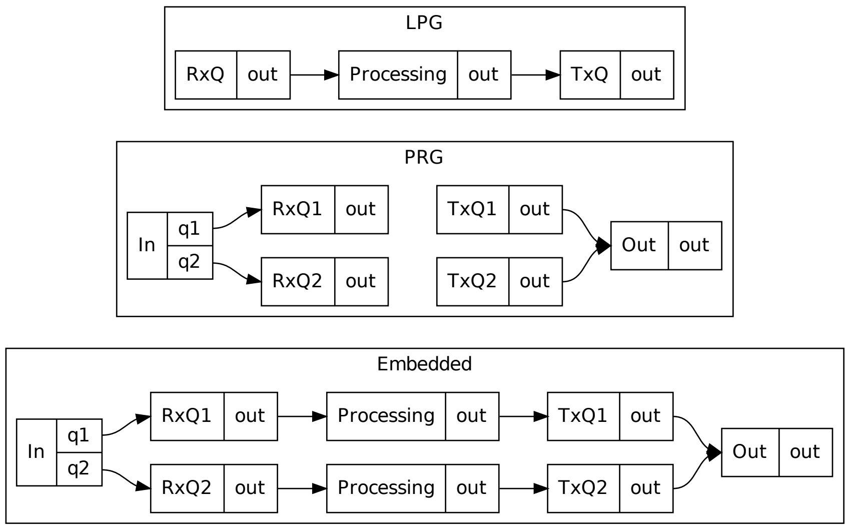

Figure 2.5: Simplified embedding example.

flows will be handled by which hardware queue based on semantics annotations in the graph, since this does not depend on the embedding algorithm.

2.3.2 Simplification

The graph simplification pass tries to remove parts of the graph that cannot be reached. Note that there usually are edges leading from reachable parts to unreachable parts, so the main goal is to find edges that will never be enabled, and then successively remove parts of the graph. Commonly this occurs in conjunction with hardware demultiplexing, where the hardware will only deliver certain types of packets to a specific hardware queue, and then demultiplexing tests for other types of packets can be removed if Dragonet can establish this fact. From there the simplification can continue by removing now unneeded parts of protocol processing, like e.g. removing TCP processing if only UDP packets will arrive on a specific hardware queue.

Procedure

The concrete implementation of this is based on tagging output ports on relevant F-nodes with logical expressions which provide some semantic information about the buffers for which the port will be enabled. These logical expressions are then aggregated by pushing them through the graph and combining them at O-nodes using the corresponding logical operator, and adding in additional semantic expressions along the paths by combining them with the incoming expression with a logical `and`. For entry nodes without incoming edge `true` is used as an input expression, which means *no information*. By aggregating the expressions along the graph edges, an expression for each output port will be generated. This aggregation pass is combined with checks for satisfiability for each port using an SMT solver, currently Z3 [18]. If an unsatisfiable expression is found for a port, `false` will be propagated along edges starting from the respective port, and the edges will be removed in a second step.

```
node ClassifyL3 {
    port ipv4[.L3IPv4ValidHeaderLength]
    port ipv6[]
    port arp[.L3ARPValidHeaderLength]
    port drop[]
    semantics ipv4 { (= L3P.IP4 (L3.Proto pkt)) }
    semantics ipv6 { (= L3P.IP6 (L3.Proto pkt)) }
    semantics arp { (= L3P.ARP (L3.Proto pkt)) } }
```

Listing 2.3: Unicorn definition of a node with some semantic annotations.

Note that a response `unknown` from the solver will never cause any correctness issues, so the solver need not be complete. In general the whole graph simplification pass is purely an optimization, and if skipped should not lead to different behavior other than possibly performance.

An unsatisfiable expression could arise if the PRG contains semantic information that would lead to an aggregated expression like

$$L4.protocol(pkt) = UDP \wedge UDP.port(pkt) = 7$$

at the receive queue node, and if this expression is then combined with an expression from another node in the LPG leading to

$$L4.protocol(pkt) = UDP \wedge UDP.port(pkt) = 7 \wedge UDP.port(pkt) = 8$$

which is obviously not satisfiable.

Semantics Syntax

As a syntax we decided to use the expressions from the SMT-LIB language [8] directly, which is a syntax understood by most SMT solvers. SMT-LIB in a parsed form seemed to be a good choice as an internal representation, as it avoids the need to target one specific SMT solver. We decided to use it directly to specify the constraints in the Unicorn files since it is definitely expressive enough to capture the required semantics, and avoided the need for developing a specialized language for this. The downside of the SMT-LIB syntax, which is based on S-expressions, is that it is neither particularly easy to read nor write, and using a less expressive alternative might make it easier to use more efficient specialized solvers.

In listing 2.3 an example of a node with semantic annotations is shown. Some semantic expressions in the LPG and PRG are very close to the node implementations, where functions are applied to the abstract `pkt` to basically check the values of protocol fields. Currently mostly uninterpreted functions are used, since these are usually sufficient for reasoning about basic packet properties as shown in the example above. For a lot of these functions more information could be provided in the definition, by basically specifying them as interpreted functions that only use functions to access particular offsets in the abstract `pkt`. This would possibly even allow some reasoning such as the distribution of packets where a known hashing scheme is used by the NIC to assign packets to queues.

2.4 Optimization

At the center of our network stack is the idea of using our model for reasoning targeted at finding a (close to) optimal configuration for the network stack in a specific scenario. We decided to implement this based on optimization guided by a cost function rating the stack generated for a particular configuration. The optimization is guided by an oracle that provides a subset of the configuration space that should be feasible to explore. Ideally the cost function would be the only entity that needs to be changed to implement a different policy.

2.4.1 Cost Function

As already mentioned above, there are often trade-offs and resource management decisions involved when using hardware features, meaning that there is in general no universally correct configuration for each feature, but rather the decision depends on what goals should be achieved, e.g. optimizing for latency instead of throughput. An example of a resource management decision often arises with hardware demultiplexing, since usually only a fixed number of filters for demultiplexing is supported by the NIC, and if there are more flows than filters some subset needs to be picked for demultiplexing in hardware. This implies that Dragonet needs some knowledge about the policy to be implemented, which should be configurable since the policy to be implemented is influenced by the concrete scenario, including requirements dictated by the application such as QoS parameters for prioritizing its flows, but others dictated by the administrator such as QoS parameters for managing multiple competing applications or power management concerns.

We are encoding this policy as a cost function which maps the resulting graph, which directly represents the processing to be implemented in software for the configuration to be evaluated, to a cost which then allows graphs to be graded. The cost function is actually evaluated on the partitioned graph, introduced in section 3.3.2, as this graph is as close as we can get to what will actually be executed.

A basic example of a cost function could evaluate the average path length to get packets from the NIC queue to a number of sockets and vice versa. Given additional information such as packet rates for different flows, and certain annotations on graph nodes about demultiplexing decisions, which are used anyway for the simplification pass discussed below, a cost function could be implemented to evaluate how well flows are balanced between queues. Using a number of additional node annotations, such as e.g. an estimated number of cycles per node, more accurate execution cost predictions for single flows or a distribution of flows should be possible.

Currently we are only using a dummy cost function, since the oracle discussed below will only emit a single configuration to evaluate (more on that below).

2.4.2 Oracle

The oracle can be thought of a function calculating a set of configurations to be evaluated based on the logical protocol graph. At the moment the oracle is implemented for each NIC or physical resource graph, since it requires detailed knowledge about the semantics of the configuration options. The optimization process will calculate and evaluate the resulting stack for each of the configurations suggested by the oracle, and then pick the best one. For future versions a more interactive role for the oracle could be beneficial, where the oracle calculates a configuration or a small set of configurations, and then waits for an evaluation of these, and chooses further configurations based on the response.

Note that the full configuration space for NICs such as the Intel 82599 [16] has a very high dimensionality, on one hand just since there is a lot of different options to be enabled or disabled, but more importantly the configurations for different filters tend to have very high dimensionality by themselves. The 5-tuple filter on the Intel 82599 is an example of this: up to 128 filters can be configured to filter based on a 5-tuple of protocol, source and destination IP address and port, where a wildcard can be specified for each field. Even without considering the wildcards, and additional per-filter parameters this leaves roughly 100 bits of information per filter, which results in 2^{100} possibilities to consider. Clearly just enumerating all configurations is not feasible.

Adding additional hints about filter fields, that would allow the optimization to reduce the set of configurations to evaluate, such as providing hints about how the fields of the 5-tuple can be taken from a flow specification in the LPG will reduce the search space considerably. But even with this optimization the search space will still become intractable quickly as the number of flows grows, since the subset of flows to be configured as filters, together with the wildcards, will still need to be chosen, and other filter types such as the Intel 82599's flow director filters will also need to be configured.

Our ideal goal is to externalize all policy decisions in the cost function, so that a change in policy can be accomplished by only changing the cost function. But currently a lot of policy is also contained in the oracle, as further research is required about how to implement a policy independent oracle. The current oracle functions in Dragonet each only provide a single configuration, thereby rendering the cost function useless for now. Again this is due to time constraints and not due to a design decision.

Chapter 3

Data Plane

3.1 Haskell Prototype

We implemented our first prototype capable of processing packets by augmenting the nodes in our existing graph data structure in Haskell with an implementation function and in addition implementing a function that traverses the graph for a packet. The prototype was capable of performing basic network stack functionality such as handling ARP, ICMP echo packets (ping), and implementing a basic UDP echo server. In order to interact with the prototype, we implemented connectors for the Linux TAP device [28], Intel DPDK [17], and Openonload [35]. While this implementation gave us some confidence that our model was suitable to implement a basic network stack, it was not developed with performance as a goal, and thus was abandoned in favor of a more efficient implementation where the data path is not implemented in Haskell.

3.2 Execution Engine

An execution engine in Dragonet is responsible for executing a protocol graph or, to be precise, execute tasks on the graph and thereby processing packets. It operates on a protocol graph that is given as input, and operates on it by maintaining a task queue and executing a single task at a time to completion.

We have implemented two different execution engines with different performance characteristics: First, an execution engine based on a *dynamic* in-memory representation of the graph, that is traversed when executing a task, which allows for cheap incremental modification of the graph at runtime but also incurs some execution overhead for traversing the graph. The second execution engine is based on generating *LLVM* code to execute the graph for a task, which reduces the runtime overhead for the execution since optimized code tailored to the graph is generated, but also makes modifications to graph more expensive since the LLVM code needs to be generated, optimized and compiled again. Both implementations are discussed in sections 3.2.2 and 3.2.3.

3.2.1 C-Implementation of Nodes

F-nodes are currently implemented as C functions, which allow for efficient execution of protocol processing. O-nodes are implemented directly by the execution engine.

These implementation functions are independent of the execution engine used. They take three parameters and return the corresponding identifier for the port that is being enabled. The following three parameters are passed:

- *Node context*: A node specific structure that currently only contains the generic node context. The generic node context is used to perform runtime operations that require information about the current node, such as spawning a new task or allocating a buffer. This structure is node-specific because it is intended to be used to pass parameters from Dragonet to some nodes when instantiating them (e.g. a queue identifier for nodes polling a hardware queue), once Dragonet supports this.
- *Global state pointer*: Pointer to the global state structure (state shared by all nodes in the system).
- *Buffer handle*: Pointer to the buffer handle that is used for execution. The buffer handle can be modified in cases where a node is enabled without a buffer, or to free a buffer. But note that our convention dictates that one execution of the graph, i.e. a buffer path, is only allowed to operate on one buffer.

Referring to Ports and Spawn Edges

Implementation functions need some way to refer to the port that is being enabled when the function returns, so we chose to assign numbers to ports based on the order they are listed in the Unicorn file (starting at 0 for each node). One exception to this rule are nodes that have `false` and `true` as ports, in which case the ports will be reordered in such a way that they are always assigned the values 0 and 1 respectively while preserving the ordering of the other ports. This exception was introduced because these ports are very common and to simplify the use of generic helpers, such as a helper converting a boolean value to a port identifier, even in nodes with additional ports.

The same approach is used to identify outgoing spawn edges when spawning new tasks (without the exception for `true` and `false`). To make the code easier to understand and maintain, symbolic constants are generated for these values by the helper utility discussed below.

Generating Skeleton

To simplify the implementation of F-nodes we implemented a tool that takes Unicorn files as parameters and generates a number C definitions to be used in the implementation. In addition to function prototypes for the C functions implementing the node behaviour a number of symbolic constants for referring to ports and spawn edges by name, as well as C structures for node contexts are generated.

Listing 3.1 shows the Unicorn definition of an example node, and listing 3.2 shows the generated C definitions.

```
node Lookup_ {
    spawn request SendRequest
    port false true[.L2EtherPrepare]
    port miss[]
}
```
Listing 3.1: Unicorn definition of example node

```
enum out_ports {
    P_TxL3ARPLookup___false = 0,
    P_TxL3ARPLookup___true = 1,
    P_TxL3ARPLookup___miss = 2,
};
enum out_spawns {
    S_TxL3ARPLookup___request = 0,
};
struct ctx_TxL3ARPLookup_ {
    struct ctx_generic generic;
};
node_out_t do_pg___TxL3ARPLookup_(
    struct ctx_TxL3ARPLookup_ *context,
    struct state              *state,
    struct input             **in);
```
Listing 3.2: C definitions for example node

Interface to the Runtime

A number of functions for interaction with the runtime are available for node implementation functions. An overview of the most important functions is shown in listing 3.3.

Spawning a new task can be accomplished by a call to `spawn`, providing the current node context, the identifier for the spawn edge to use, a priority, and optionally a buffer to use. The priority can currently only be high or low, and controls to what end of the task queue the new task will be added. A high priority task will be added to the front of the queue, where it will be found when looking for the next task to execute, while a low-priority task is added to the back. Low-priority tasks are especially useful for nodes that do polling and therefore re-spawn themselves.

Allocating and freeing a buffer requires the node context, to provide a reference to the buffer pool to use.

3.2.2 Dynamic

Given our graph model and the implementation functions for the F-nodes, an obvious approach to execute the graph is to use the same approach as for the Haskell prototype discussed above. But in order to achieve better performance, we decided to build a similar execution engine in C. This also implies that dynamic execution engine will have to operate on a separate data structure that can now be tailored to allow faster execution. Using a pointer-based data

```
// Spawn a new task
bool spawn(
    struct ctx_generic *ctx,
    struct input       *in,
    enum out_spawns     s,
    enum spawn_priority p);

// Allocate buffer
struct input *input_alloc(
    struct ctx_generic *ctx);

// Free buffer
void input_free(
    struct ctx_generic *ctx,
    struct input       *in);
```

Listing 3.3: Runtime functions available to implementation functions

structure also means that applying changes incrementally can be done efficiently with good locality. In addition to the C code for manipulating and executing the graph, there is also a Haskell module that is responsible for building and modifying the dynamic graph.

Data Structure

A slightly simplified version of the dynamic graph data structure currently used for execution is shown in listing 3.4. Basically the graph is built from a number of `dynamic_node` structures that are connected by `dynamic_edge` structures, and for each entry point to the graph, where a task can be spawned, a `dynamic_spawn` structure is allocated and points to the respective start node.

Edges are stored in a linked list on both sides, in a per-port list on the source node, `ports`, and in the list of predecessors, `preds`, on the destination node. This means that both adding an edge when modifying the graph, and finding the successors nodes to enable during execution, can be done efficiently. To simplify the execution of O-nodes edges are also tagged with the originating port identifier, which makes it easier to find out which nodes can contribute true inputs and which nodes can contribute false inputs.

Instead of just storing a pointer to the node in the task handles, we added another layer of indirection in form of the `dynamic_spawn` structure. This avoids the need to scan through the task queue when removing or recreating an entry node, since only the spawn structure has to be updated to point to the new node or `NULL`. Outgoing spawn edges are therefore stored as an array of pointers to `dynamic_spawn` structures.

Depending on the type of a node, different parameters are stored in the `tdata` union. For regular F-nodes a function pointer along with the node context are stored, and the generic node context will contain a pointer to the `dynamic_node` structure used by some of the runtime functions in the node implementation. O-nodes only need to be tagged with the logical operator they implement. In addition to these node types from the model, there are some special cases for nodes that are generated by Dragonet and generally need some runtime parameters, such as a pointer to a queue or socket handle. These nodes are executed directly by the execution engine, and do not have a pointer to an implementation function.

Execution

An execution iteration of the graph begins by taking a task off the task queue, which will result in a pointer to a `dynamic_spawn` structure and possibly a buffer. If the node pointer in the spawn structure is `NULL`, the task will be dropped, the circumstances leading to this are discussed in more detail in 3.5.2. Otherwise execution will start as a depth-first search from this node based on the port enabled in each node. Edges that are not enabled are ignored for the search. The concrete implementation is based on a stack where the enabled successor nodes are pushed to and nodes to be executed are popped from.

O-Nodes Note that this simple DFS-based approach can result in O-nodes being enabled repeatedly since they can have multiple incoming edges. The successor nodes of an O-node must only be enabled once, which can only happen once the result from the logical operation based on the inputs is defined. O-nodes can also be short-circuited, in which case the node might already have been executed when it has already been enabled previously in the same task. This is where the `out_value` and `out_version` members of the node structures are used to implement the correct behavior. The `value` member is used to store the output port that was enabled once a node executed successfully, and the `version` member is used to recognize values from earlier task executions, based on a version value that is incremented with each iteration and is saved together with the `value` member. Tagging the values with the version of the current iteration avoids the need for resetting the values in all node structures after the execution. The `value` member is also used when executing O-nodes to determine which input edges were enabled by the predecessor nodes.

Special F-Nodes In addition to regular F-nodes and O-nodes there are some nodes that are treated as special cases by the execution engine. These special nodes are nodes that are generated by Dragonet and usually require certain parameters, such as queue handles in the case of the `ToQueueX` nodes introduced in 3.3.2. In the future it would be desirable to implement a general mechanism to pass node parameters through the context structures to nodes, and implement them like regular F-nodes in implementation functions.

Termination A task is executed completely once the stack of nodes is empty, or a node tagged as a terminal node is executed which will terminate execution even if there are other nodes left to execute. Currently the only nodes tagged as terminal nodes, are some the special case node types mentioned above.

Control Interface

The graph inside a dynamic execution engine can be initialized and modified using a number of control functions, that implement changes to the internal data structures. Currently the following control functions are offered:

```
struct dynamic_node {
    struct dynamic_graph   *graph;
    const char             *name;
    enum dynamic_node_type type;
    /** Set to enabled port during execution */
    node_out_t              out_value;
    int                     out_version;
    /** Incoming edges */
    struct dynamic_edge    *preds;
    /** Outgoing edges arranged into ports */
    size_t                  num_ports;
    struct dynamic_edge   **ports;
    /** Outgoing spawn edges */
    size_t                  num_spawns;
    struct dynamic_spawn **spawns;
    /** Node type specific data */
    union { struct { nodefun_t           nodefun;
                     struct ctx_generic *ctx;
                 } fnode;
            struct { enum dynamic_node_op op;
                 } onode;
            struct { int32_t muxid;
                 } mux;
            struct { queue_handle_t queue;
                 } queue;
            struct { void    *sdata;
                     uint64_t sid;
                 } socket;
    } tdata;
};
struct dynamic_edge {
    /** Source and destination node of edge */
    struct dynamic_node *source;
    struct dynamic_node *sink;
    /** Links for list of edges at both ends */
    struct dynamic_edge *so_next;
    struct dynamic_edge *si_next;
    /** Originating port */
    node_out_t          port;
};
struct dynamic_spawn {
    struct dynamic_node *node;
    uint32_t refcount;
}
```
Listing 3.4: Graph data structure used in dynamic execution engine (simplified).

- *Creating a node.* These are actually a number of functions to create different types of nodes, but all of them will return a pointer to the node.
- *Adding ports to a node.*
- *Adding an edge* originating from a port pointing to another node, returning a pointer to the edge structure.
- *Creating a spawn structure* for a node, returning a pointer to it.
- *Modifying the node pointed to by a spawn structure.*
- *Removing a spawn structure.* Must only be called once no more nodes point to this spawn structure. The spawn structure will only actually be freed once no more tasks in the queue point to it. Reference counting is used to ensure this.
- *Clear the whole graph.* Removes all nodes and edges, but the tasks in the queue will remain and spawn structures will not be removed but only modified by setting the node pointer to `NULL`.
- *Spawning an initial task.* Takes a pointer to a spawn structure and adds a task without a buffer to the task queue. This is used to bootstrap graph execution.

Note that this interface should be extended in the future to allow more modifications to the graph, such as removing individual nodes or edges.

Remote To allow Dragonet to control instances of the dynamic execution engine that are residing in different address spaces, an adapter is provided to forward control commands through general communication channels such as sockets. Since none of the operations is expected to fail, except possibly in the case of memory allocation problems, and to provide better performance, the remote control interface does not provide feedback messages. Since most control operations either return a pointer to some structure and/or take such pointers as parameters, a mechanism is implemented that requires the user to pass an id for operations that return pointers. This id can then be used later on to refer to the result of the corresponding operations, when providing it as a parameter for other operations. The Haskell module to control dynamic execution engines is based on this remote adapter.

3.2.3 LLVM-based

The other execution engine we implemented for the pure dataflow-based model is based on the idea of generating LLVM code for executing the graph, which should allow more efficient execution of protocol graphs. Due to time constraints this execution engine has not yet been adapted for the task-based model. But there is no inherent problem in adapting the implementation.

Generated Code

The implementation used for executing the pure dataflow graphs is also based on executing the nodes in depth-first search order. But the main difference is that the graph structure is not represented by a data structure used at runtime, but directly in the source code. For each node a wrapper function is generated that will be called when a node is enabled, and if/when a port to enable is determined, the wrapper functions for the successor nodes will be called. These wrapper functions take a pointer to the buffer as well as the port that was enabled on the predecessor node as parameters.

The part of the wrapper function that decides if and what port should be enabled, is again dependent on the node type. For O-nodes a counter is maintained to count the number of non-short-circuit edges (either true or false, depending on the operator) that have been enabled. If either all non-short-circuit edges are enabled, or one of short-circuit edges is enabled, the counter will be reset and the corresponding port is enabled. Note that in the cases of short-circuiting or if not all edges are enabled, the counter is not necessarily reset to 0 at the end of an execution iteration. Because of that, all counters are reset before each graph execution. F-nodes are again executed using the same implementation functions discussed before. The same special cases for F-nodes discussed for the dynamic execution engine apply here as well, for this execution engine code will be generated for the different special cases.

Optimizations

Using LLVM as a framework for generating machine code also allows us to use the generic LLVM optimization passes. Depending on the settings for the optimizer pass, the result will be one big function for executing the graph after inlining and other optimization passes have been applied. There are presumably other opportunities for execution that could be found by analyzing the resulting code.

Adapting to Task-Based Model

The following changes could be used to adapt this execution engine to the current execution model. For the wrapper functions, the type signatures need to be changed to pass a pointer to a pointer to the buffer, since the implementation functions need to be able to change the buffer. Also some of the special cases for F-nodes would need to be adapted to the new model, e.g. by changing nodes that poll queues to respawn themselves. The major change is replacing the current main execution loop that is generated, which basically executes the graph by starting execution at each of the entry nodes in turn, by a function that implements a single iteration of taking a task off the queue and calling the respective wrapper function for the entry node. The last part that is missing is an implementation of the task spawn function, which could be implemented by reading a node identifier from the node context that is passed, and then basically building a case statement that will return an entry node id based on the current node id and the spawn edge id passed as a parameter.

3.3 Multiple Cores

Given the concept of an execution engine we can now process packets using a protocol graph. But having a single thread somewhere operating on a big protocol graph and doing protocol processing will not be sufficient to achieve good performance.

3.3.1 Naive Approach

A naive approach to improve this would be to add multiple threads working from the same task queue, but there are a number of drawbacks to this approach.

First off, there is an obvious *scalability* issue caused by the shared task queue, although this could be addressed by using multiple work-stealing queues [11]. In addition there will also be the need for synchronization measures for any state (that is not per task) accessed by a node, since this state might be accessed concurrently on any of the other threads. This also includes accesses to the NIC, even in cases where the NIC provides multiple hardware queues, since the queues are not dedicated to specific threads.

The other major issue is with *locality* which has also been shown to have a significant impact on network stack performance [30, 13]. For example protocol processing for multiple packets of a single specific TCP connection should be performed on the same core [13]. Another heuristic is that (as far as possible) packets should only be touched by one core [19]. This implies that if a packet for a specific socket arrives, it should arrive on the NIC receive queue assigned to the core servicing that socket in the application, and protocol processing should also be performed on this core.

Given that there are a significant performance impact as well as a number of policy decisions involved with spatial scheduling of protocol graph execution, this factor should also be captured by our model. If these factors are captured in our model, we can also make these decisions based on our model and thereby also externalize the policy decisions.

3.3.2 Partitioning

In order to be able to reason about the resulting performance we need to be able to figure out where a particular node will be executed for a particular packet. The obvious solution given our graph abstractions is to assign each node to a particular core, resulting in a partition of the graph into subsets of nodes executed on the same core. While this restricts each node to only one core, nodes can be replicated before partitioning the graph, meaning that this restriction does not reduce flexibility.

Reasoning

Using this information it is straight forward to start reasoning about communication costs and interconnect loads. The graph provides all the necessary information, if an edge crosses between protocol graph components there will be communication involved, and therefore a path in the graph provides the necessary information to arrive at a cost estimate.

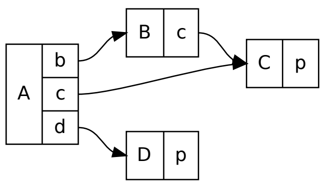

Figure 3.1: Example protocol graph for partitioning

Implementation

From an implementation point of view each of the resulting graph components is assigned to an instance of an execution engine. An edge that crosses between components means that the buffer (including its attributes) needs to be handed off to a different execution engine instance, which implies some communication mechanism.

One possibility for implementing this hand-off would be to extend the execution engines so they know about the other instances and can hand off buffers when required. We chose an alternative implementation that is based on modifying the graph and adding nodes for handing off buffers to, and receiving buffers from other components. This approach avoids the need to modify and couple the execution engines, and also adds explicit information to the graph about communication.

We implement a partitioning pass that takes the protocol graph as an input, and outputs both, independent protocol graphs for each execution engine instance, as well as information about what communication channels to establish. The partition pass will add a number of additional nodes to the resulting graphs. The following kinds of nodes will be added when partitioning a graph:

- `ToNodeX`: Add multiplexing identifier to buffer, to enable demultiplexing at the receiving end.
- `ToQueueX`: Enqueue buffer on queue.
- `FromQueueX`: Poll queue.
- `Demux`: Demultiplex based on the multiplexing identifier of buffer.

Both regular dataflow edges and spawn edges crossing between components will be redirected to the local `ToNodeX` node, from where they will be passed on to the responsible `ToQueueX` node.

Figure 3.1 shows an example protocol graph, and Figure 3.2 shows the resulting output after partitioning.

Restrictions

To avoid excessive synchronization and generally simplify the implementation, we currently allow a buffer to be used by only one execution engine at a time. This means that partitions where nodes in more than one component are enabled for the same buffer are considered illegal. If the need should arise in the future, it might be possible to allow this in some cases by adding additional nodes during the partition pass and passing the buffer through the components sequentially.

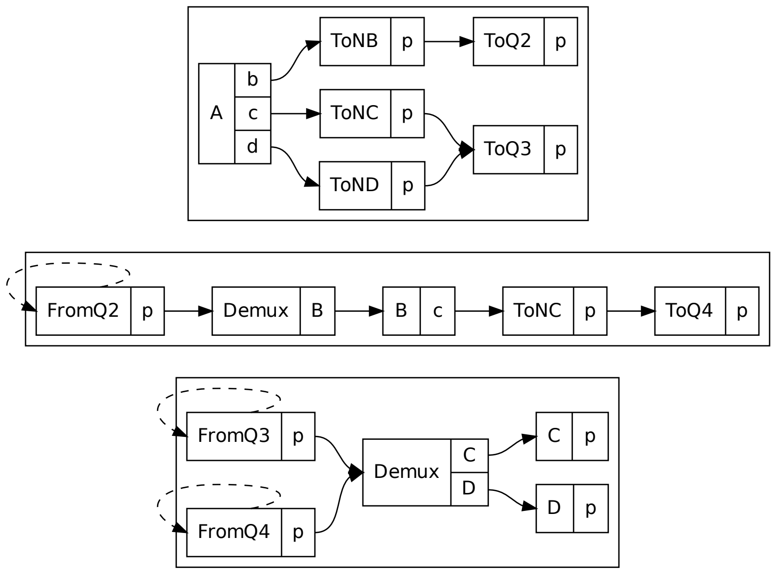

Figure 3.2: Simplified example of partitioned example graph

For simplicity's sake, we currently also assume that only one node in the destination component is enabled for each buffer. Allowing multiple nodes to be enabled could presumably be accomplished by using a bit map for the multiplexing identifier, and rearranging the `ToNodeX` nodes.

It is currently also not possible to spawn tasks without buffers using spawn edges crossing between components. This is due to the way the channels between components are currently implemented (basically by passing buffers with some annotations).

In practice, none of these restrictions have caused problems for our use cases.

3.3.3 Communication

For the implementation of the channels used for communication between graph components we decided to use a Linux implementation of the bulk data transfer infrastructure developed in an earlier project for Barrelfish [6, 14]. The channel implementation used provides single-producer/consumer channels, and is based on a modified version of URPC [10] and Barrelfish UMP [9], which offer inexpensive communication between different cores without kernel involvement in the critical path. We decided to use the bulk transfer infrastructure since it provides full control over buffer management, good performance, and a generic interface making it easy to use multiple channel implementations.

Bulk Transfer Overview

The bulk transfer infrastructure developed earlier provided both a generic interface for transferring bulk data over a variety of underlying communication mechanisms, as well as implementations for machine local communication and cross machine communication based on Ethernet. The actual interface developed is callback-based, i.e. an endpoint specifies a set of functions to be called in reaction to different events. Data is transferred in buffers over unidirectional single producer and consumer channels. Buffers are managed in pools, to avoid some per-buffer management overhead, while still providing the flexibility by using different pools.

Communication starts with the establishment of a channel between two endpoints. Before a buffer can be transferred through a channel, the buffer's pool needs to be assigned to the channel, which will prepare the channel to transfer buffers in the pool, i.e. by mapping the associated memory on both endpoints to enable fast transfers. After this, two different data transfer operations can be used to send a buffer through the channel: `move` transfers the buffer through the channel, and at the same time passes ownership of the buffer to the receiver, meaning that the sender can no longer access this buffer, while the receiver now has full control over it. With `copy` on the other hand, buffer ownership remains with the sender, and the receiver gets read-only access to the buffer, while ownership remains with the receiver allowing for transfers of the same buffer over other channels, although the buffer will be read-only for the sender until the buffer is released on the receiver. When a buffer is no longer needed by the receiver it can be passed back to the sender by either a `pass` operation thereby passing ownership back to the sender but without a guarantee for the contents of the buffer at the sender side, or by a `release` operation if a read-only copy was received.

Design Choice

We decided to use single producer and single consumer queues for handing off packets from one specific component to another. The other two alternative extremes for this design choice would be, on one side using one multi-producer/single-consumer channel for each component, or on the other side to use one channel for each edge. We decided against the first extreme since multi-producer/single-consumer channels require more synchronization and are therefore generally more expensive, and we would have lost the flexibility of choosing different channel implementations for communicating between different component pairs. The other extreme of using one channel per edge results in higher overheads, since each channel will require memory and possibly other resources. Thus the chosen solution seems like a reasonable compromise.

Modifications to Bulk Transfer

In the course of adapting the bulk transfer infrastructure to implement communication between components we made some changes to the original infrastructure. One one hand we implemented a channel back-end for Linux, which is based on POSIX shared memory, and can be applied for both communication inside and across process boundaries. This also involved implementing generic infrastructure for sharing buffer pools between different processes, and enabling clients to allocate physically contiguous buffers.

The other significant change was to the interface. After starting to use the existing interface, we realized that the original callback-based interface was not ideal for our purposes, since it did not offer sufficient control over when what kinds of callbacks and from what channel events will be received. Therefore we added a more low-level interface exposing more detail, while at the same time allowing for an implementation of the existing callback-based interface based on the new low-level interface.

The main change in the low-level interface is to use a function to poll for events and another function to indicate that the application is done processing an event. This allows the application to control when events should be accepted, and also makes it possible to process events asynchronously and possibly out of order.

3.4 Buffer Management

In Dragonet a buffer consists of three things:

- `struct input`: This structure is the main handle for referring to buffers and passing them around, and contains pointers to the other two parts and some bookkeeping information. It is local to an execution engine instance.
- Data buffer: Contains the actual packet data in a bulk transfer buffer. The `input` struct contains information about what part of the buffer actually contains valid data.
- Attribute buffer: Used to pass information about the buffer between nodes. Also stored in a bulk transfer buffer to enable zero-copy transfer when passing the buffer.

3.4.1 Requirements

Buffer management operations, i.e. allocate and free, are heavily used on the critical path, and therefore slow buffer management will result in a significant slowdown of overall networking performance. Another factor that makes buffer management more difficult is that it is required on multiple cores and, in the case of applications, even in multiple processes. Combining those two facts also implies that a scalable solution is required, which in turn suggests using a distributed approach instead of a centralized one. In the case of applications the issue of trust also enters the picture, since the buffer management scheme must not leak data between applications, and there is also a potential issue of malevolent applications trying to sabotage the system.

3.4.2 Implementation

The bulk transfer infrastructure provides the some of the necessary mechanisms to implement distributed buffer management. For our case we currently allocate one local pool for each component, and buffers are always allocated from the local pool.

Since pool management is also local to the pool owner, buffers must be returned through the bulk transfer channels to the pool owner in order to free them. And given that a buffer can be moved through an arbitrary number of channels, and since not all components are pairwise connected, in the general case there is no direct channel to the pool owner when freeing a buffer, and it needs to be passed through a series of channels. To avoid the need for a general routing scheme in order to figure out which channel to use for transferring a buffer back to the pool owner, a buffer will be tagged with the channel it was received from, and when it is being freed it will be returned through this channel.

3.4.3 Limitations

Note that this simplistic buffer management scheme does not necessarily use the shortest possible path when freeing buffers, which would require a general routing scheme. The current implementation also does not provide any facility to enforce buffer management, since it just keeps all the buffers from all assigned pools mapped for performance reasons. This can be problematic when dealing with uncooperative applications.

3.5 Incremental Changes

Until this point we assumed that the graph remains static once execution starts. But since the protocol graph also contains the network state, changes to the network state at runtime will also result in changes to the protocol graph. The network state will change for example in when an application opens a new connection or closes an existing connection. Depending on the workload such changes can occur frequently, therefore the cost of implementing the graph modifications at runtime, will contribute directly to the cost for these actions.

3.5.1 Implementation

Note that it is also not feasible to stop the execution of all graph components by waiting until all communication channels and task queues are empty. First, since the number of graph components in real systems can be large and since scheduling can cause unknown reaction times for components, synchronizing would just take too long. The other problem is that tasks queues will never be empty at runtime, since there will always be some tasks that re-spawn themselves, e.g. for polling queues.

Implementing changes at runtime is also complicated by the fact that components are potentially executed in different processes. This implies some communication and/or synchronisation mechanism is required in order to implement changes. Currently commands describing changes are sent to components using either a Unix socket if the component is running outside of the Dragonet main process, or a basic locked queue for local communication.

Partitioning

In general changes to the protocol graph, will result in both, changes to the component graph, as well as changes to the protocol graphs for the different components. Changes to the component graph can involve the following:

- Starting new components; includes preparing a new instance of an execution engine.
- Adding a communication channel between two components.
- Modifying the protocol graphs executed by components.

- Removing a communication channel.
- Stopping components

The above changes are shown in the order in which they need to be applied if multiple changes are required, since there are dependencies between some of the changes. For example, a node polling a communication channel cannot be added before the channel itself is initialized, and vice versa when tearing down a channel.

Note that some of these changes are (mostly) independent of the execution engine being used. Adding and removing communication channels needs to be done the same way for all execution engines.

Dynamic Execution Engine

For the dynamic execution engine applying changes is by construction incremental. The challenge here is calculating a diff between the existing graph and the new graph, to know what exactly needs to be changed. Applying the incremental changes can be done using the same interface that is used to set up the initial graph, and the only difference is that the commands are preceded by a command to suspend graph execution and followed by another command to resume execution.

LLVM Execution Engine

In the LLVM execution engine changing the protocol graph basically boils down to generating new LLVM code for the changed graph, and applying the LLVM optimization and code generation passes from scratch. There might be some opportunities to apply caching to avoid regenerating parts of the LLVM code that have not changed, but most of the time will be spent in the optimization and code generation passes anyway.

Hybrid Execution Engine

Given the trade-offs between the dynamic execution engine allowing for cheap updates, and the LLVM execution engine featuring more efficient execution a combined approach seems to be a reasonable compromise. The main idea here, would be to keep both an instance of the dynamic and LLVM execution engines. Changes will be applied first to the dynamic instance, and then execution will start based on this instance. Afterwards, possibly after some delay since changes might occur in bursts, an optimized LLVM implementation of the graph can be generated and optimized. Once the LLVM instance is ready, the dynamic instance will be stopped and the LLVM instance can start executing. Due to time constraints this approach has not been implemented or evaluated at the time of writing.

3.5.2 Limitations

Currently not all types of changes are supported at runtime. Moving tasks between components is not supported at the moment, i.e. a task in the task queue whose start node disappears or moves to a different component, it will simply be dropped. This can be problematic in cases of tasks that spawn themselves where the destination node is moved to another component, since the task will not be started in the destination component. For our current graphs we can work around this issue because all of these self re-spawning nodes are tagged as with the `init` attribute, and will therefore be spawned manually after creating the node in the new component.

A similar issue also occurs with the communication channels between components if the destination node for a buffer on the channel moves. Note that simply waiting for the channels to be drained is not sufficient due to the asynchronous updates of components. Currently these buffers will be dropped and freed.

One possible approach to solve this issue would be to add proxy nodes, that will forward these tasks to the right component. The difficulty here is to find out when it will be safe to remove the proxy nodes. In the general case, especially with fast successive updates, just dropping the nodes after a fixed number of updates will not work. Another complicating factor is the fact that the communication channels between components are generally opaque to the execution engine and to the Dragonet control path.

3.6 Application Interface

The following section describes the interface used by applications for both data and control operations in the network stack in the first half, while the second half will discuss its implementation.

3.6.1 Interface

At the time of writing Dragonet only provides a low-level interface that provides the application with full control. In the future there will likely be a compatibility interface providing (limited) socket compatibility, and possibly also another interface hiding some of the details while still allowing for an efficient zero-copy data path.

There are two main concepts in the low level application interface:

- *Application Queue*: The application queue represents a connection of the application to the network stack. An application can open more than one application, e.g. one for each thread. Application queues are used both for to exchange control information with the stack, such as opening a new connection, and also to send and receive data.

- *Socket Handle*: Used to send and receive packets for a particular network endpoint or connection. Note that there can be multiple socket handles, usually on different application queues, that serve the same network endpoint/connection. In case of multiple socket handles packets can be sent from each, and received packets can be received on any of the socket handles, although the stack provides no guarantees about the distribution of packets to socket handles.

It is worth noting that operations on a specific application queue, even if accessed through distinct socket handles, are not thread-safe. So if multiple application threads will be performing operations on a particular application queue, the application is responsible for ensuring mutual exclusion. This choice is in line with leaving full control to the applications, since it allows applications that do not require synchronization to avoid the unnecessary overhead.

The resulting full application interface is reproduced in appendix A.

Application Queue Operations

The following operations in the interface are related to dealing with application queues:

- *Create* a new application queue. A label identifying the stack instance to connect to, and a label for the application queue, that is used for informational and debugging purposes, need to be passed as parameters.
- *Destroy* an application queue.
- *Allocate* and *free* buffers. Currently a fixed-size buffer pool is allocated when creating an application queue. In the future the interface should be modified to allow the application to allocate buffer pools of specified size.
- *Process events* on the application queue. This is the operation that needs to be called to receive packets, as well as perform internal processing that might be required. There are three outcomes for this operation:
  - An event occurred, and information about the event is returned.
  - Internal processing was performed.
  - No visible event occurred and no internal processing was performed.

  The idea behind the separation of no event and internal processing is to let the application know about opportunities to perform other tasks than network processing when no more work is to be done temporarily. Note that it is also quite likely that multiple calls in a row will perform internal processing, since one goal for the implementation is to keep the run time of this call bounded, as far as possible.

  Currently the only event is the receipt of some data on a socket handle, in which case the event information will provide the buffer and socket handle.

Socket Handle Operations

Socket handles can be manipulated with the following operations:

- *Create* new socket handle on the specified application queue. The socket handle cannot be used for receiving or sending data until it is bound.
- *Destroy* a socket handle.
- *Span* a socket handle, i.e. create another socket handle bound to the same network endpoint. This operation is usually used to span a socket to another application queue, allowing packets for the same network endpoint to be received on the specified application queue as well.

  Note again that the interface does not provide any guarantees about load balancing between socket handles for the same endpoint. The client also needs to make sure events from all application queues with socket handles for a particular network endpoint are being processed, otherwise data could be missed.

- *Bind* an unbound socket handle to a network endpoint. The current implementation only supports IPv4/UDP endpoints, and the endpoint specification consists of source and destination IP and port, where not all entries need to be provided. After a successful bind operation the socket handle will be ready to send and receive data. Attempting to bind to a network endpoint that is already used by another socket handle in the system will result in an error.
- *Send* out the specified buffer on the socket handle.

3.6.2 Implementation

From an implementation point of view an application queue consists basically of two parts: There is the data path used to send and receive data (packets), which is basically implemented by instantiating a component for each application queue. The control path to Dragonet that is used to send commands such as to create a new socket, and to receive control messages from Dragonet, such as commands for modifying the data path.

Data Path

The component used for implementing the data path of an application queue is treated by Dragonet like any other component. There are however some special nodes, similar to the nodes generated by the partitioning pass (see 3.3.2), that can only appear in these application queue components:

- `ToSocketX` node: One node for each socket handle that is bound to a network endpoint will be created. This node is used for receiving data, if this node is enabled a data receive event for the respective socket handle will be issued.
- `FromSocketX` node: Again, one of these nodes will be created for each socket handle, and is used to send out data originating from the respective socket. Data will be sent out by externally spawning a task starting at this node.

Note that the execution engine instance will not be running in a separate thread, but execution works by processing one task at a time for each call to the "process events" function. No task will be executed if internal processing on the control channel is performed on a particular call.

If a `ToSocketX` node is enabled during task execution, an event containing the socket handle and the buffer will be returned by the call. For sending out a buffer over a socket handle, a task starting at the `FromSocketX` node will be spawned. Note that the socket send call will only spawn the task but not execute it, meaning that the buffer will only be actually sent out on one the next "process events" calls.

Currently, send tasks are spawned with high priority, which means that they will be enqueued at the top of the task queue. However, in the case of multiple sending tasks being spawned on the same application queue without processing events in between, this will lead to a LIFO order when executing the sending tasks. While this might be useful in terms of cache locality, this can lead to reordering and even starvation. Spawning these tasks with low priority is also undesirable, since they should be executed before other operations such as polling queues. Therefore fixing this will probably require the introduction of a third priority higher than low, where tasks are processed in a FIFO manner.

Control Path

The control path is used to communicate information that is not data to and from Dragonet. A Unix socket is currently used for communicating control information related to each application queue. The socket is opened when creating an application queue. Currently there are two different types of control information that is exchanged.

On one side, there will be control messages that are sent to the execution engine instance, to implement graph changes and so on. These messages can arrive either in response to a request sent by the application, e.g. to close a socket, or they can arrive unsolicited in the case where Dragonet implements changes in the graph in response to some other event not originating from this application queue.

For operations such as e.g. binding, closing or spanning a socket handle, a request is sent to Dragonet which will either succeed or return an error. In the case of success this usually leads to changes in the protocol graphs issued by Dragonet. Operations such as binding a socket will depend on the changes in the protocol graph being implemented (otherwise they might run into missing `FromSocketX` nodes). These operations will currently block and wait until the necessary changes have been implemented. This is implemented by processing messages on the control channel until the success message arrives, which will only be issued by Dragonet once all the graph modification are issued, and since the control commands to the execution engine instance are sent through the same channel and will not be buffered, this is sufficient.

Chapter 4

Evaluation

This chapter provides evaluation results to show that Dragonet can provide throughput and latency that is competitive with existing network stacks, using a comparison to the Linux network stack. Performance is evaluated using a micro benchmark for each latency and throughput, plus a benchmark evaluating throughput of `memcached`[3], a widely used key-value store. In addition there is also an experiment showing Dragonet's current Achilles' heel: the cost of opening new sockets.

4.1 Setup

4.1.1 Hardware

Two different sets of machines were used for the following experiments. For the throughput experiments, both the micro benchmark and the `memcached` benchmark, the machine described in table 4.1 was used. The latency benchmarks as well as the socket open benchmarks were run on machines of the type described in table 4.2.

Both machines support Intel Data Direct I/O also known as DDIO [15], which allows the NIC to directly write packets to the L3 cache and also read them from the L3 cache without the need for going through RAM. DDIO replaces direct cache access or DCA [23] on previous generation Xeon CPUs which allowed for data to be prefetched into L3 cache when receiving data, but data was still going through RAM, which can be avoided in DDIO for both directions. Also according to Intel, DDIO is fully transparent in the sense that it does not require support from PCIe devices, and should work with existing devices. There is one caveat, which is that data will be written to the L3 cache of the CPU where the PCIe bus is local to, which implies that careful placement of tasks to cores is needed, to get the maximum benefits.

4.1.2 Dragonet Configuration

Intel 82599 [17] based NICs were used for the experiments below. Support for the NIC is implemented in Dragonet based on the `igb_uio` kernel module from the Intel data plane development kit [17], or DPDK, which is used to expose the hardware registers to user space. On top of that a modified version of the

CPU:	2 x Intel Xeon E5-2670 v2
	Ivy Bridge, 10 cores each, hyper-threading disabled
NIC:	Intel 82599ES
OS:	Ubuntu 14.04 x86_64, Linux Kernel 3.13.0

Table 4.1: Machine used as a server in for the throughput benchmarks

CPU:	Intel Xeon E5-2430
	Sandy Bridge, 6 cores, hyper-threading enabled
NIC:	Intel X520 (82599-based)
OS:	Ubuntu 13.04 x86_64, Linux Kernel 3.8.0

Table 4.2: Machine for the latency and socket open benchmarks

Barrelfish `e10k` driver was used for interacting with the NIC. No protocol offload features besides Ethernet CRC and padding were used.

For these experiments Dragonet was configured to implement a basic policy distributing flows evenly to 4 hardware queue pairs using the 5-tuple filters offered by the Intel 82599. The policy will start assigning flows in a round robin manner starting with queue 1, since queue 0 handles a number of application-independent functions such as ICMP and ARP. The reason for not using more than 4 queues is that Dragonet will currently poll queues, which means that each additional queue will add a thread that will just be polling the hardware queue, and processing packets when they arrive. Combining these hardware queue threads with application threads that are also busy polling results in poor performance, because both threads will just keep polling until preempted by the kernel.

4.2 Latency Micro Benchmark

The first experiment is a simple UDP round-trip time benchmark. On the server side a basic UDP echo server, `fancyecho`, is used, that just sends back any data it receives. `fancyecho` is configured to listen on a single port and use a single thread. Note that the server itself does not touch the packet payload, but just forwards it. To determine the RTT the benchmark client will send packets of various sizes, with one packet in flight at a time, and measure the time delay until a response arrives. For both cases, Dragonet and Linux, the server is run on the respective platform, while the client runs on Linux in both cases.

4.2.1 Results

The results for this experiment are shown in figure 4.1, which compares the round-trip time when running with the server on Linux to the scenario when the server is running on Dragonet. The x-axis shows the UDP payload size, to which at least 46 bytes of headers will be added [1]. On the y-axis the round-trip

[1] 8 bytes UDP header + 20 bytes IPv4 minimal header length + 14 bytes Ethernet header + 4 bytes Ethernet CRC = 46 bytes. Note that the Ethernet CRC will be both added and stripped by the NIC on both ends, so these four bytes will never be visible to software or transfered over the memory bus

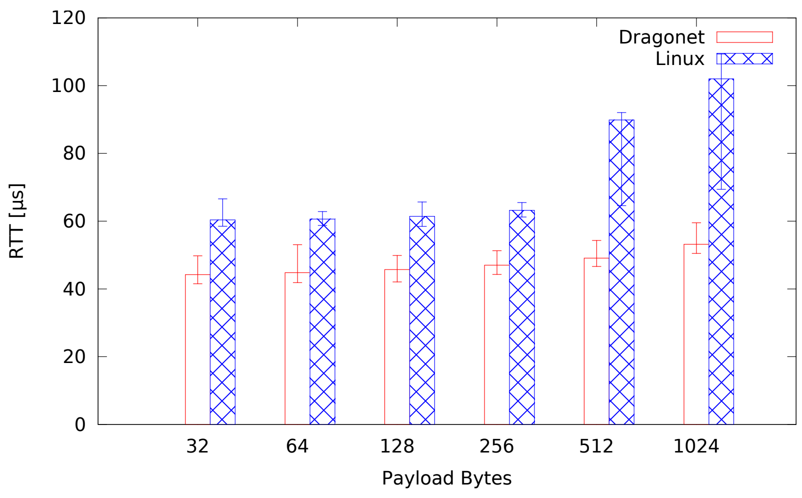

Figure 4.1: Round-trip time

time is shown, where each bar shows the average over 10000 runs, and the error bars represent the minimum and 99th percentile. For small packets with 32 bytes payload the RTT with Dragonet is roughly 25% lower, and this difference increases further to over 45% for 1024 bytes payload. This leads to two questions:

Why is the RTT lower with Dragonet than with Linux?

One of the differences between the Linux stack and Dragonet is that Dragonet currently uses *polling* for getting packets from the NIC and also in the application for receiving packets from the stack. For this experiment there will be 3 threads running and polling: One for the application polling the application queue, and two for polling hardware queues. There will be two hardware queues being polled since Dragonet will assign the echo server flow two queue 1 to avoid interference from other packets on the default queue. Each thread is assigned to its own core, in which case a single ping-pong cycle will involve two cores. The packet will be received on the NIC receive queue and processed by the corresponding thread and then passed to the application, which will in turn pass it back to the originating thread for protocol processing and to send it out through the hardware queue. Note that passing the packet between the two threads is cheap since there are no context switches involved, and the queue is polled. And since the echo server does not touch the packet payload, there should only be cache misses for the queue, the attributes used to identify the socket handle, the header, and a number of attributes used to pass the destination information when sending the packet.

Another major difference is that Dragonet implements a *zero copy* data path from the NIC hardware queue to the application and vice versa. Note that the impact of this even more pronounced as the echo server does not actually touch the packet payload, otherwise the copy at least serves to warm up the cache and thereby partially amortizing the cost. The POSIX API used by Linux makes it hard, if not impossible, to implement efficient zero copy, since the application can just specify any range of memory as a buffer for storing received data, or to send out data. For receiving, the issue is that advanced hardware support would be required to make sure that the right part of the right packet ends up in the specified buffer. In addition for both directions, there is also the complication of integrating this with the rest of the memory management, to ensure that the physical memory backing the buffers will remain available while a buffer is used by the NIC, be it for receiving or sending.

For the forwarding scenario used in the echo server, Dragonet also allows for the *buffer* in which the packet is received to be *reused* for sending out the response. On one hand this saves buffer management overhead, since no buffer needs to be freed or allocated. In addition there is also a locality advantage for protocol processing on the send path, since the headers as well as the payload, will still be warm in the cache when doing the processing such as adding headers and calculating checksums.

In general Dragonet uses quite *simplified protocol processing* as opposed to Linux which implements full protocol processing including all kind of generic corner cases. While part of this is currently definitely due to the fact that Dragonet is missing many of these features, there is also an argument to be made that the performance should not suffer from adding support for more advanced corner case to Dragonet if they are not actually being used. The rationale here is that Dragonet can use the model to figure out which parts are actually used in a specific instance of the stack, and therefore eliminate unused parts using the graph simplification pass. Marinos et al. [27] also provided some evidence that there is significant cost to the generality in Linux' network stack, and that specialized network stacks can significantly improve performance.

Why does Dragonet scale better with the packet size?

The graph points to some higher per-byte overhead in Linux than in Dragonet. Looking at the discussion the obvious candidate is the copying of the payload in Linux. Linux actually copies the data twice, once when receiving and once when sending. Note that the performance penalty is not just about the CPU cycles wasted for the actual copy, but might also incur cache misses for the destination buffers, depending on how exactly buffers are managed.

4.2.2 Improving Latency

The next question is: what can we do to further improve latency? For achieving lower latency one of the main goals is to minimize the number of cycles required for getting a packet from the NIC, processing it, and handing it off to the application.

One way to reduce the latency for receiving and sending packets is to *directly map a receive and send queue pair* to the application address space [25, 31]. The NIC needs to provide the required hardware support, both for allowing this do be done in a secure manner, and to get sufficient flexibility to set up packet demultiplexing in hardware. For implementing this in Dragonet, only a small number of changes is required: The graph execution environment in the application that is already used in connection with the application interface would need to be extended to support executing generic F-nodes. In order to implement support for moving Intel 82599 driver nodes to the application, the Intel 82599 driver infrastructure will need to be modified so nodes have a way to access the descriptor rings in hardware, where currently the assumption is used that all Intel 82599 nodes are all in the same process. There is also often a trade-off to doing user space networking of losing control of outgoing packets, which can only be done if the NIC provides sufficient hardware mechanisms, including egress filtering, rate limiting, etc. With Dragonet the decision if user space networking should be used if e.g. proper egress filtering is not possible should eventually be captured by the cost function, so it can be used in scenarios where the missing egress support is not a problem.

Implementing support for a number of *offload features* such as checksum offload could also improve latency. For example if the NIC can calculate checksums faster than the software implementation, offloading the checksum will reduce the latency.

Updating the *LLVM-based execution engine* to work with the current execution model and possibly implementing additional optimizations for generating more efficient code, could also potentially reduce latency by reducing the number cycles spent for executing the graph for the packet.

There is also a number of *inefficiencies in the node implementations*, such as bounds checks for each packet access for debugging purposes. Addressing these inefficiencies could also result in some improvement.

4.3 Throughput

The following two benchmarks are discussing the UDP throughput that can be achieved with Dragonet compared to Linux, in a micro benchmark and using a real application, and how performance scales with an increasing number of application threads.

4.3.1 Micro Benchmark

On the server side this experiment again uses `fancyecho`, the UDP echo server. In contrast to the latency experiment, the payload size is now fixed to 1024 bytes, but the number of threads used will vary. Four other servers, each with a 10Gbps NIC, were used to generate load and measure throughput, using `netperf` [5] on the Linux network stack for both cases. Static load balancing based on the source IP address of the packets is used to distribute packets to the echo server threads, this is implemented by opening multiple listening sockets that specify both the destination port and the source IP. Note that locality regarding which server thread processes which client packet does not matter for this particular experiment, because the server only echoes back the data. The load balancing is only used to balance the packets across the hardware receive queues, so as to achieve parallelism for packet processing.

Results

In figure 4.2 the resulting throughput with different numbers of application threads is shown, both for Linux and Dragonet. For each data point 3 iterations with a measurement time of 10 seconds each were performed, and the error bars show the minimal and maximal values. The x-axis shows the number of threads used for `fancyecho`. On the y-axis the throughput, measured as *payload/time*, is shown.

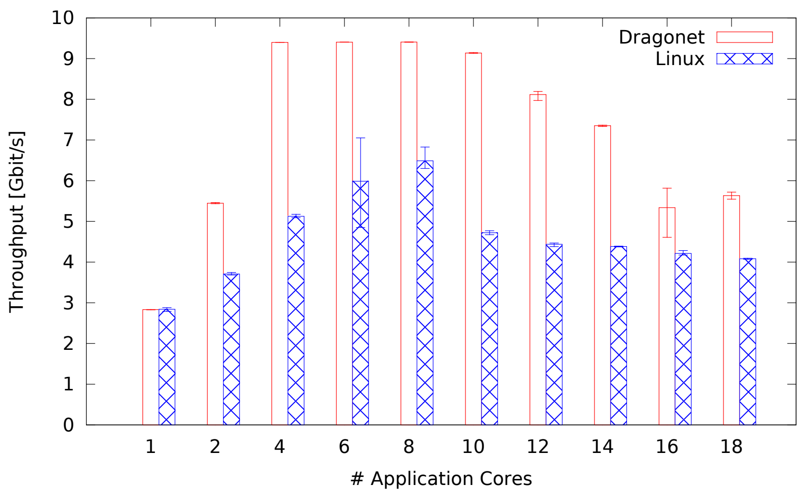

Figure 4.2: Throughput micro benchmark

After factoring in the overhead for protocol headers, i.e. 46 bytes of headers in addition to the 1024 bytes of payload, the actual throughput including headers is $\sim 4.4\%$ above the numbers shown in the graph. Which makes for a peak throughput of roughly 9.9Gbps. This graph again suggests a number of questions:

Why is Dragonet's throughput higher than Linux'? Note that many of the causes already discussed in the context of the latency benchmark also apply here. In particular both using zero copy and the simplified protocol processing results in *fewer cycles that are required for processing* a single packets, thereby freeing up CPU cycles for processing additional packets, resulting in a higher throughput.

In addition, for this throughput benchmark *not doing protocol processing on the same core* where the application thread resides could actually improve performance. The reason here is that the penalty for the reduced locality will be a small number of cache misses for the queue and accessing a small number of the attributes or headers to identify the socket handle when the buffer arrives in the application process, but no cache misses for the actual payload. Processing on a separate core also almost eliminates the instruction cache footprint of the code required for sending a packet in the application thread. And as long as enough cores are available to avoid sharing cores between threads, this will actually lead to some parallelism between protocol processing and the application code.

Linux also has a number of points in the network stack where *coordination among cores* is required. Some profiling of the Linux stack actually showed that an increasing fraction of the total time spent in the network stack will be spent for the lookup in the routing table when sending out an IP packet as the number of cores used grows. In Dragonet currently no routing table is used, since no complex routing decisions are required benchmarking scenario. But even if some routing were required, this could presumably be expressed in the protocol graph directly without requiring synchronization. The only data structure currently requiring synchronization in Dragonet is the ARP cache, where a reader-writer lock is used.

What happens with Linux when going from 8 to 10 cores? In the graph there is a significant performance drop for Linux when going from 8 to 10 cores. Note that the application threads are manually assigned to cores while skipping the first core, which means that there will be one thread pinned to a core on the *second CPU die*. This causes synchronization between the cores to become more expensive, as it involves traversing the interconnect. In addition this also means that DDIO will send the packets for the additional thread to the wrong L3 cache, and sending out packets will also get more expensive since the buffers cannot be accessed directly from the L3 cache.

Why does Dragonet's throughput drop after 10 cores? For Dragonet the drop from 8 to 10 cores is less pronounced, since much less coordination among cores is required. But there is still some throughput drop-off, which is partly due to the fact that communication between the application thread and the hardware queue threads will be more expensive, but after 12 cores there will also be some penalty because there will be cores that are shared between threads that are all polling. This experiment was using 8 threads for serving the send and receive hardware queues, 4 each [2], meaning that there won't be enough cores once there are more than 12 application threads.

4.3.2 memcached Benchmark

`memcached` throughput is evaluated using the widely used `memaslap` benchmark [2]. The setup for this benchmark is identical to the throughput micro benchmark, except that `fancyecho` is replaced by `memcached` as a server. Again the performance of Dragonet is compared to Linux with different numbers of worker threads for `memcached`. For `memaslap` a configuration of 10% writes and 90% reads, with 64 byte keys and 1024 byte object size was used.

Results

The results are shown in figure 4.3. The x-axis again shows the number of worker threads used in `memcached`, while the y-axis shows the average over 3 runs of 10 seconds each, with the error bars showing the minimal and maximal value. In general the discussion about the throughput results from above still applies. But there are some additional questions to be discussed for this experiment:

Why is the throughput for Dragonet lower than in the micro benchmark? The major difference to the micro benchmark is that the application now performs some work before a response will be sent, which will also include using the packet payload. Therefore it is not surprising that more threads will be required to get close to line rate. This also explains why the performance hit taken when going beyond 12 cores is significantly more pronounced, since the time spent polling when there are already packets enqueued for the corresponding application thread will be more costly in terms of throughput, and since the application thread might not even be able to process all requests during its time-slice. Also note that executing a thread doing processing on a core together

[2]Due to time constraint the experiments were not re-run once Dragonet supported serving a send and receive queue pair with one thread.

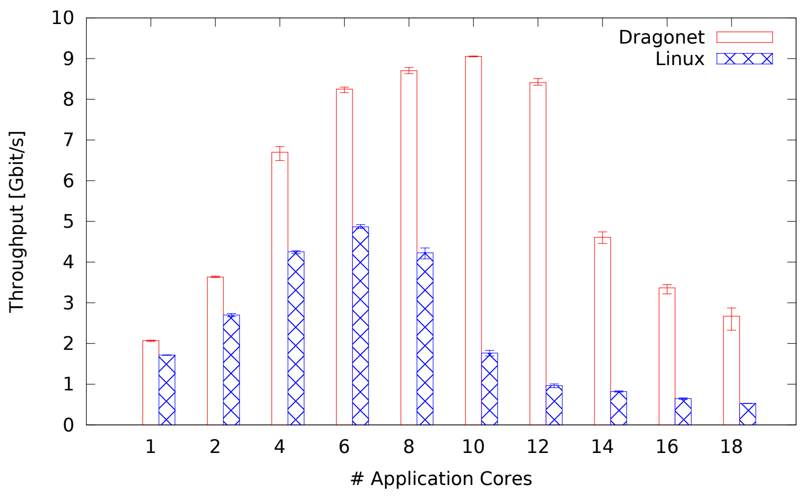

Figure 4.3: `memcached` throughput benchmark

with an application thread will not only impact the specific application thread, but might also cause additional idle time for other application threads waiting for packets from the same queue.

Why is Linux doing so much worse than in the micro benchmark? One part is that with Linux no control over which packets go to which core will be available, resulting in no locality for requests. In the case of the throughput micro benchmark, the impact of this will be fairly minimal, since the echo server will just send the packet back out, while `memcached` will have to perform some operations on internal data structures, where request locality will have an impact.

Note that the hit for the instruction cache pollution caused by the operations that are performed for sending and receiving packets in Linux will cause a higher performance penalty since there will be significantly more instructions to execute with `memcached` than with the micro benchmark.

The drop from 8 to 10 cores where threads will start to be executed on the second CPU socket, and with that in another NUMA domain, is also more pronounced since the `memcached` threads will have to access common data structures. But this effect will apply to Dragonet and Linux equally.

4.3.3 Improving Throughput

Most of the suggestions for improving latency mentioned above will also lead to improved throughput. Protocol offloads will generally lead to increased throughput as long as the NIC can perform them at line rate, and as long as there is no dominating configuration overhead. In general any optimization leading to fewer cycles spent on packet processing should improve throughput, since more packets can be processed per time interval.

For *directly mapping NIC hardware queues* to applications, the case is a bit less clear cut. If CPU heavy processing is required for requests and if separate cores for processing can be spared, it could in some cases be beneficial to hand off protocol processing to a different core, so as to reduce the number of required cores for application threads, and thereby improving cache locality for the application. This would have to be evaluated empirically for the specific application scenario, but again this should be something that should very easy eventually by just modifying the cost function.

Introducing *notifications* instead of polling queues, both for software queues and also for hardware in form of interrupts, will also significantly reduce the overhead for mixing multiple threads on the same core, since threads can be suspended until a notification arrives instead of just polling for the full time slice. This should significantly reduce the drop-off for higher numbers of application threads in the scenarios above. Here there will again be a number of different design choices that will have to be evaluated, and some form of adaptive polling might still be desirable depending on the scenario [37]. Notifications could also be useful in conjunction with directly mapped hardware queues, to allow the application to execute some low-priority operations while there are no packets to be processed.

There is currently still a more general issue not directly related to performance which is the fact that there are still some implementation details preventing Dragonet to *gracefully drop packets* when queues fill up and so on. Currently an overflowing software queue will lead to an abort, which is obviously not acceptable for real applications, and also complicates benchmarking throughput somewhat.

4.4 Graph Modification Performance

One consequence of the implementing changes on the model first, and then going through the whole process of re-running the optimization to find a configuration and then preparing the graph to be implemented, is that changing things will be more expensive than just adding an entry to a data structure. The following experiment shows the current state of this in Dragonet, by measuring the time it takes to open different numbers of listening UDP sockets. Note that this part of Dragonet has not been optimized for performance in any way, so this experiment is only intended to show that this needs to be addressed, especially for scenarios where sockets are being opened and closed dynamically after the initialization phase.

4.4.1 Results

Figure 4.4 shows the results of this very basic benchmark. The x-axis shows the number of sockets that were opened, while the y-axis shows the total time it took to open x sockets. Looking at this graph it is clear that this is an issue to be addressed, since times in the order of magnitude of seconds will clearly be inappropriate for many applications. The graph also shows that the time to open a number of sockets is growing faster than linear, which might also need to be addressed. Due to time constraints no additional analysis was performed to evaluate where the time is spent.

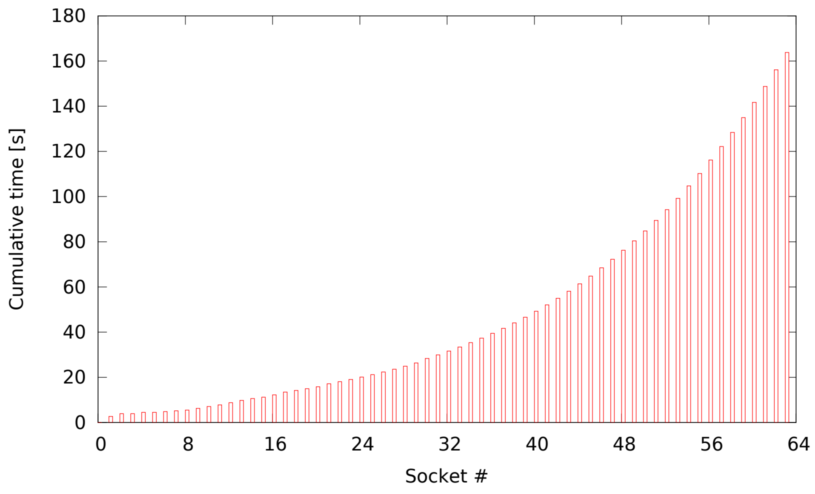

Figure 4.4: Cumulative time for opening multiple sockets

4.4.2 Improving

As no in-depth analysis was performed of where the time is spent, the following suggestions are just educated guesses to be taken with a grain of salt. The process for instantiating the stack for a particular LPG is also likely to change while performing subsequent research on the optimization process for finding the configuration to be used.

One thing that should definitely help to reduce this cost, would be to use *incremental algorithms* for the whole process of generating an updated LPG, then going through the optimization loop, and finally setting up the graph to be executed. Currently every each change to the network stack state implies starting from scratch with an unconfigured LPG and go through the whole process of building a stack from there. At the moment no incremental algorithms are used since they are generally a lot more difficult to develop than an algorithm which just starts from scratch.

As discussed in section 2.3.2 currently there is also an *SMT solver* used in each iteration when building up the graphs. It is possible that this process could be sped up by using more efficient special purpose solvers. There might also be optimizations for the interaction with the SMT solver, such as using models from previous versions of the graph as a starting point for the solver.

The *data structure for representing protocol graph* is another design point that might be subject to change. Currently a fully functional representation from Martin Erwig's functional graph library[20] is used. Given how heavily we rely on graphs, it might could turn out that a more efficient not purely functional data structure will need to be used.

Chapter 5

Conclusion

Processing network data at the still increasing rates offered by current and future network technologies requires hardware support to effectively distribute packets to multiple cores and generally reduce CPU load, resulting in more complicated NIC designs, which are difficult to fully exploit with current network stack designs. Our approach to address these issues is to leverage a model-based approach to build a flexible network stack capable of adapting to the myriad of hardware features offered by current and future NICs.

This thesis discussed our design of Dragonet, a full network stack based on this approach, and compared its performance characteristics to Linux. The performance evaluation showed that Dragonet offers both throughput and latency that are at least comparable to Linux, and in many cases superior. Dragonet's current Achilles' heel was also evaluated, namely the cost of control operations such as opening new sockets.

The contributions of this thesis are as follows:

- Discussion of the current *minimal control plane* in chapter 2, which produces a graph to be implemented by the data plane. The graphs are built and modified in reaction to changes in the network state, such as sockets being opened and closed. Although the control plane is currently very basic, it provides a foundation for future extensions.

- The *data plane* implementation in chapter 3, is responsible for executing the software part of the graph generated by the control plane. This includes executing the graph efficiently across multiple cores, and interfacing with applications.

- Chapter 4 presented a performance evaluation comparing Dragonet to Linux, demonstrating that a network stack built using our model-based approach can deliver competitive performance. For a UDP latency micro benchmark improvements of 25-45% were seen, depending on the payload size, when running the server on Dragonet instead of Linux. Throughput results also show significant improvements in most cases, both for a micro benchmark, and also for an application-based benchmark using `memcached`. Dragonet has not been optimized yet for fast control operations such as opening sockets, and a micro benchmark also confirms that more work will be needed on that front.

5.1 Future Work

In addition to addressing the limitations documented in the previous chapters, there are a number of possible extensions to Dragonet that could be implemented as future work. The following sections discuss some potentially interesting starting points.

5.1.1 Extending the Control Plane

In contrast to the data plane, the control plane in Dragonet is currently fairly minimal, providing just enough to allow basic performance evaluation. Now the data plane and the current Dragonet implementation in general provide a reasonable foundation for implementing and evaluating options for the control plane implementation. Some interesting questions to answer there are:

- Is there an efficient way to implement the configuration space exploration where all policy is externalized in the cost function?
- How much information does need to be provided on a per-device basis to allow for an efficient search?
- What representation should be used for the cost function to enable both a high degree of generality and make it feasible for sys-admins to write cost functions?
- Can Dragonet provide good performance for scenarios with a lot of short-lived connections being opened and closed at a high frequency, and what optimizations and trade-offs does this require?

5.1.2 Modelling Additional Protocols and NICs

In order to substantiate our claim for the generality of our approach, it needs to be evaluated both with additional protocols such as TCP, and also by applying it to more different NICs. One example, that would make a strong case, would be a sufficiently complete TCP implementation, together with physical resource graphs for NICs supporting different types of TCP offloads, as these are notoriously difficult to handle by traditional network stacks.

5.1.3 Instrumentation of Execution Engine

Given our execution model, which is based on combining implementation functions for multiple nodes into a full protocol stack, it should be possible to add various instrumentations to graph execution without the need to modify the implementation functions. The instrumentations could be added with a number of different goals:

Profiling

One motivation would be to obtain detailed accounts of the execution time, using profile annotations, that are added to all nodes or some subset thereof. An infrastructure similar to the trace-based approach used in Barrelfish for both collection and analysis of event traces [36] could be used.

Debugging

Given the relatively complex graphs used during execution, some bugs and therefore a need for a debugging mechanism are inevitable. While general purpose debuggers can certainly be applied to Dragonet, building special-purpose debugging tools could definitely make debugging substantially easier. One example would be a graphical representation of a graph execution for specific packets, which could then be combined with features such as breakpoints on certain nodes/ports/edges, and single step execution on a per-node granularity, or more fine-grained if desired.

5.1.4 Application Interface Changes

Currently Dragonet exposes a fairly low-level interface which, while providing the application with a high degree of control, might not be appropriate for all applications.

Notifications

One problem that has to be addressed is the current need for polling, which is not suitable for all applications. Ideally Dragonet should provide a light-weight notification mechanism, that can be used when appropriate, but is not mandatory if polling is preferred. In addition there should be a possibility to combine the notifications from the network interface, with other notifications, such as data being ready on a file descriptor in a Unix system.

Backwards Compatibility

While there is a lot of evidence that the POSIX socket interface imposes substantial performance overhead [22, 31, 32], it is never the less desirable to provide applications where performance is not critical with a backwards compatible socket interface. Ideally Dragonet should still be able to provide some performance improvements even when using the socket interface, depending on the scenario.

Integration of the Application into the Graph

Another approach for applications to interface with the network stack requiring more substantial changes to an application, but possibly yielding additional benefits, would be to integrate them directly into the graph. This integration could be partial, by basically adding a number of nodes to the graph, which the interface with other code running in some application threads in an application specific manner. But depending on the application, it could be possible to implement it fully using our graph execution semantics, allowing for tight integration with the rest of the stack, and therefore possibly resulting in performance benefit, as well as a simpler implementation. There are a number of options in the design space such as how the application-contributed parts of the graph will be executed, or what kind of isolation guarantees between applications need to be provided.

5.1.5 DSL for Node Implementations

Node implementation functions are currently implemented in C. But there might be a number of reasons to switch to alternative languages there. It would be especially interesting to see what benefits can be obtained by using a domain-specific language especially designed for implementing nodes in Dragonet. One benefit could be significantly simpler implementations of nodes, but there are others as discussed below.

Optimization for Generated Code

Using a DSL for specifying node implementations would also presumably enable more efficient code generation, since Dragonet would then actually understand node implementations and the assumptions used therein. This applies especially to the LLVM-based execution engine, where Dragonet could be able to implement additional optimizations when stitching together node implementations to generate code for executing the graph.

Offloading Code to NIC

Having a representation of node implementations that can be understood by the control plane, could also enable it to take advantage of additional hardware. One example would be generating code for NICs that can be programmed[1], or even NICs with FPGAs on board [4], at runtime.

5.1.6 Extending to more Complex Systems

Once Dragonet is sufficiently complete as a network stack using a single NIC, it is conceivable to extend Dragonet to more complicated systems.

Multiple NICs

The obvious next step would be support for managing more than one NIC. This should be mainly an engineering issue, but there is some question about what information needs to be specified about the NICs to allow Dragonet to decide which NIC to use for what, e.g. if they are connected to different networks.

Across Machine Boundaries

In a data-center setting it could even be desirable to extend Dragonet to control the network stack across multiple machines communicating with each other, e.g. to control what requests should be sent to what machine, or to move different parts of processing to different machines, in settings where packets pass through multiple machines. This approach could possibly even be used to model and configure additional network devices such as switches.

Bibliography

[1] Flownics | netronome. `http://www.netronome.com/product/flownics/`. Retrieved September 2014.

[2] memaslap - load testing and benchmarking a server – libmemcached 1.1.0 documentation. `http://docs.libmemcached.org/bin/memaslap.html`. Retrieved September 2014.

[3] memcached - a distributed memory object caching system. `http://memcached.org/`. Retrieved September 2014.

[4] Netfpga. `http://netfpga.org`. Retrieved September 2014.

[5] The netperf homepage. `http://www.netperf.org/netperf/`. Retrieved September 2014.

[6] Reto Achermann and Antoine Kaufmann. Bulk transfer over network. Distributed systems lab, ETH Zurich, February 2014.

[7] Boon S Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. *Computer Systems and Technology Laboratory HP Laboratories*, 2001.

[8] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[9] Andrew Baumann. *Inter-dispatcher communication in Barrelfish, Barrelfish Technical Note 011.* Barrelfish Project, ETH Zurich, December 2011.

[10] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[11] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[12] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. CiAO/IP: A highly configurable aspect-oriented IP stack. In *10th International Conference on Mobile Systems, Applications, and Services*, pages 435–448, 2012.

[13] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–8, 2010.

[14] Jeremia Bär and Claudio Föllmi. Bulk transfer over shared memory. Distributed systems lab, ETH Zurich, February 2014.

[15] Intel Corporation. Intel data direct I/O technology (Intel DDIO): A primer. `http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf`, February 2012. Revision 1.0. Retrieved September 2014.

[16] Intel Corporation. Intel 82599 10 GbE controller datasheet. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`, January 2014. Revision 2.9. Retrieved September 2014.

[17] Intel Corporation. Intel data plane development kit (Intel DPDK). `http://dpdk.org/doc/intel/dpdk-prog-guide-1.7.0.pdf`, June 2014. Reference Number: 326003-008. Retrieved September 2014.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[19] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *22nd ACM Symposium on Operating Systems Principles*, pages 15–28, 2009.

[20] Martin Erwig and Ivan Lazar Miljenovic. fgl: Martin erwig's functional graph library. `https://hackage.haskell.org/package/fgl`. Retrieved September 2014.

[21] V. Gazis, E. Patouni, N. Alonistioti, and L. Merakos. A survey of dynamically adaptable protocol stacks. *IEEE Communications Surveys and Tutorials*, 12(1):3–23, January 2010.

[22] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–148, 2012.

[23] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network I/O. In *32nd Annual International Symposium on Computer Architecture*, pages 50–59, 2005.

[24] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[25] Antoine Kaufmann. Low-latency OS protocol stack analysis. Bachelor thesis, ETH Zurich, January 2012.

[26] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[27] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *2014 Conference on SIGCOMM*, pages 175–186, 2014.

[28] Florian Thiel Maxim Krasnyansky, Maksim Yevmenkin. Universal tun/tap device driver. `https://www.kernel.org/doc/Documentation/networking/tuntap.txt`. Retrieved September 2014.

[29] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems*, pages 5–5, 2003.

[30] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *7th ACM European Conference on Computer Systems*, pages 337–350, 2012.

[31] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. Technical Report UW-CSE-13-10-01, University of Washington, June 2014. Version 2.1.

[32] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference*, 2012.

[33] Pravin Shinde, Antoine Kaufmann, Kornilios Kourtis, and Timothy Roscoe. Modeling NICs with Unicorn. In *7th Workshop on Programming Languages and Operating Systems*, 2013.

[34] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *14th Workshop on Hot Topics in Operating Systems*, 2013.

[35] Inc. Solarflare Communications. OpenOnload. `http://www.openonload.org/`. Retrieved September 2014.

[36] David Stolz and Alexander Grest. Trace collection, analysis, and visualization for Barrelfish. Distributed systems lab, ETH Zurich, February 2013.

[37] The Linux Foundation. napi. `http://www.linuxfoundation.org/collaborate/workgroups/networking/napi`, November 2009. Retrieved September 2014.

[38] The Linux Foundation. toe. `http://www.linuxfoundation.org/collaborate/workgroups/networking/toe`, November 2009. Retrieved August 2014.

Appendix A

Application Interface

```
// These structures are opaque to applications
struct dnal_app_queue;
struct dnal_socket_handle;

/**
 * Connection of the application to the network stack.
     Concepitionally this is a queue pair packets can be received
     on or sent out through. An application can have multiple
     application connections (e.g. for different threads).
 */
typedef struct dnal_app_queue *dnal_appq_t;

/**
 * A socket handle represents a particular socket on a particular
     application queue. It will be used to send out packets from
     the respective socket through the respective application
     queue, as well as to specify the destination socket for
     packets received on the respective application queue.
 */
typedef struct dnal_socket_handle *dnal_sockh_t;


enum dnal_aq_event_type {
    DNAL_AQET_INPACKET,
};

/** Event received on app queue */
struct dnal_aq_event {
    enum dnal_aq_event_type type;
    union {
        struct {
            /** Socket handle this packet is destined for */
            dnal_sockh_t   socket;
            /** Buffer for received packet */
            struct input *buffer;
        } inpacket;
    } data;
};


enum dnal_net_destination_type {
    DNAL_NETDSTT_IP4UDP,
};

```

```
/** Specifies a network destination. */
struct dnal_net_destination {
    enum dnal_net_destination_type type;
    union {
        struct {
            // 0 can be used as a wildcard
            uint32_t ip_local;
            uint32_t ip_remote;
            uint16_t port_local;
            uint16_t port_remote;
        } ip4udp;
    } data;
};


/******************************************************/
/* Application queues */

/**
 * Create application queue
 *
 * @param stackname Label for the stack to connect to (currently always 'Dragonet')
 * @param slotname  Label for slot this queue connects to on Dragonet side (can only connect one application queue to each slot)
 * @param appqueue  Location to store handle
 */
errval_t dnal_aq_create(const char  *stackname,
                        const char  *slotname,
                        dnal_appq_t *appqueue);

/**
 * Destroy application queue (not implemented). All sockets created on or spanned to this app queue need to be destroyed first.
 *
 * @param appqueue Handle for app queue to destroy
 */
errval_t dnal_aq_destroy(dnal_appq_t appqueue);

/**
 * Poll application queue for an event
 *
 * @param appqueue Application queue to poll
 * @param event    Location to store the received event
 *
 * @return Four cases to handle:
 *   - If an event is found, SYS_ERR_OK is returned
 *   - If internall processing is done, but no event is generated
 *       DNERR_EVENT_ABORT will be returned. Polling again immediately could
 *       return an event.
 *   - If no event was found, DNERR_NOEVENT
 *   - Other error codes might be returned in case of failure
 */
errval_t dnal_aq_poll(dnal_appq_t            appqueue,
                      struct dnal_aq_event *event);

/**
 * Allocate new buffer for use on this queue.
 *
 * NOTE: Eventually we should decopule buffer allocation from
     application queues.
 *
 * @param appqueue  Application queue to allocate buffer from
 * @param buffer    Location to store pointer to the buffer
 */
errval_t dnal_aq_buffer_alloc(dnal_appq_t      appqueue,
                              struct input   **buffer);

/**
 * Free buffer
 *
 * @param appqueue  Application queue to free buffer to
 * @param buffer    Buffer to free
 */
errval_t dnal_aq_buffer_free(dnal_appq_t     appqueue,
                             struct input  *buffer);

/**
 * Get pointer to shared global dragonet state.
 */
struct state *dnal_aq_state(dnal_appq_t appqueue);


/*****************************************************/
/* Sockets */

/**
 * Create new socket. Note this socket needs to be bound to a
     network endpoint before it can be used.
 *
 * @param appqueue      App queue to create socket on
 * @param sockethandle  Location to store socket handle
 */
errval_t dnal_socket_create(dnal_appq_t     appqueue,
                            dnal_sockh_t  *sockethandle);

/**
 * Bind socket to network endpoint.
 *
 * @param sockethandle  Socket handle
 * @param destination   Network endpoint to bind to
 */
errval_t dnal_socket_bind(dnal_sockh_t
     sockethandle,
                          struct dnal_net_destination
     *destination);

/**
 * Span socket to other queue. Creates a new socket handle that
     can be used to receive packets that belong to the specified
     socket on the specified application queue. The new socket
     handle can also be used to send out packets from this socket
     using the new application queue. Note: This provides no kind
     of guarantees about which queue which packets will be received
     on.
 *
 * @param orig          Handle for socket to be spanned (must be
     bound)
 * @param newqueue      App queue to create socket on
 * @param sockethandle  Undbound socket handle (from socket_create)
     on newqueue
 */
errval_t dnal_socket_span(dnal_sockh_t orig,
                          dnal_appq_t  newqueue,
                          dnal_sockh_t sockethandle);

/**
 * Close particular socket handle. Other handles to the same
    socket will remain untouched.
 *
 * @param sockethandle Handle to be closed
 */
errval_t dnal_socket_close(dnal_sockh_t sockethandle);

/**
 * Send out data on a socket handle.
 *
 * @param sockethandle Handle to send on
 * @param buffer       Buffer to send out
 * @param dest         Network destination to send to. Can be NULL
    for
 *                       flow-based sockets (UDP flows, or TCP
    connections in
 *                       the future).
 */
errval_t dnal_socket_send(dnal_sockh_t
    sockethandle,
                          struct input                 *buffer,
                          struct dnal_net_destination *dest);

/**
 * Reads out the per-socket opaque value saved previously, or NULL
    if not initialized.
 */
void *dnal_socket_opaque_get(dnal_sockh_t sockethandle);

/**
 * Set the per-socket opaque value.
 */
void dnal_socket_opaque_set(dnal_sockh_t sockethandle,
                            void         *opaque);
```