


Technische Universität München
Department of Informatics

Bachelor's Thesis in Informatics

Accelerating Snabb Programs

Fabian Bonk

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Accelerating Snabb Programs

Snabb Programme beschleunigen

Author	Fabian Bonk
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Paul Emmerich, Dominik Scholz
Date	April 15, 2018

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, April 15, 2018

Signature

Abstract

Snabb is a network function virtualization toolkit written in Lua. It allows users to process packets using a customizable network of small Lua modules. libmoon is a Lua wrapper around the DPDK toolkit for fast packet processing. The goal of this thesis was to implement compatibility wrapper around libmoon allowing Snabb apps to run on top of DPDK and thereby increase their performance as well as allow NICs without Snabb-support to be used.

Contents

List of Figures

Chapter 1

Introduction

1.1 Outline

Network function virtualization (NFV) is the practice of abstracting network functions into smaller, more manageable modules. By combining simple modules, even complex network functions can be performed this way. Network function virtualization does not require dedicated hardware such as application-specific integrated circuits (ASICs). Instead it runs on cheap and readily available commodity (server) hardware.

Using modern CPU features such as Direct Memory Access (DMA) and Direct Cache Access (DCA) in combination with modern, high-performance (at least 10GbE) NICs can offer similar performance to ASICs without the high price and development effort. NFV allows operators to change a device's configuration at will and usually on-the-fly, depending on current demands.

Snabb [2] is a flexible network function virtualization toolkit. It allows users to implement network functions by combining Snabb-provided and self-written modules. Modules forward packets between each other and perform basic network functions. Combining multiple modules allows arbitrarily complex network functions. Most of Snabb is written in the Lua programming language and run on LuaJIT [3]. Some internal functionality is implemented in C and called from Lua using LuaJIT's foreign function interface [4].

libmoon [5] is a wrapper around DPDK [6] allowing fast packet processing using Lua scripts. It, too, uses LuaJIT and its foreign function interface.

1.2 Goal of the thesis

The goal of this thesis is to implement a Snabb-compatible runtime that achieves a performance improvement over Snabb by internally calling the Lua DPDK-wrapper libmoon. Ideally it runs Snabb programs with only minor modifications.

Chapter 2

Snabb

2.1 Overview

Snabb allows flexible packet processing using so-called *apps*. Users launch and connect apps together using *programs*. Programs are Lua scripts describing a network of apps. Programs are run on the Snabb runtime *Snabb Core*. Snabb also offers a large library of functions and modules allowing easy creation and modification of packets. See Figure 2.1.

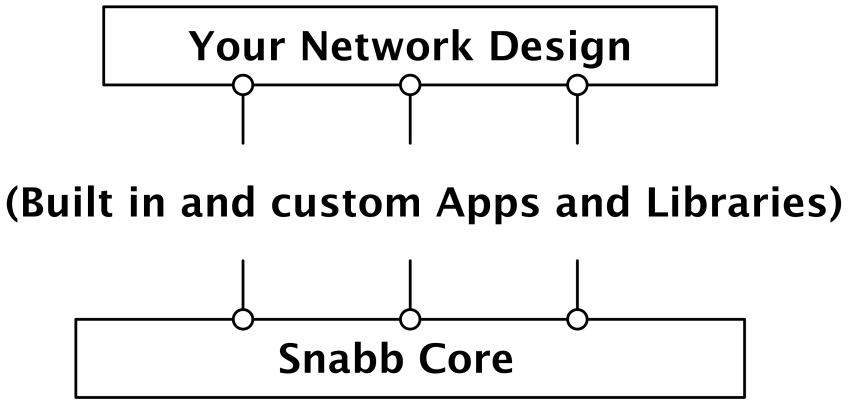

Figure 2.1: Snabb structure [1]

Snabb is largely written in Lua with some C functions to improve performance.

Snabb's reference manual defines Snabb's own interfaces as well as the methods that need to be exposed by custom apps. [1]

Figure 2.2 visualizes an example Snabb program containing four apps, two of which implement some filter functionality with the other two being NICs. The arrows are links along which packets flow from app to app.

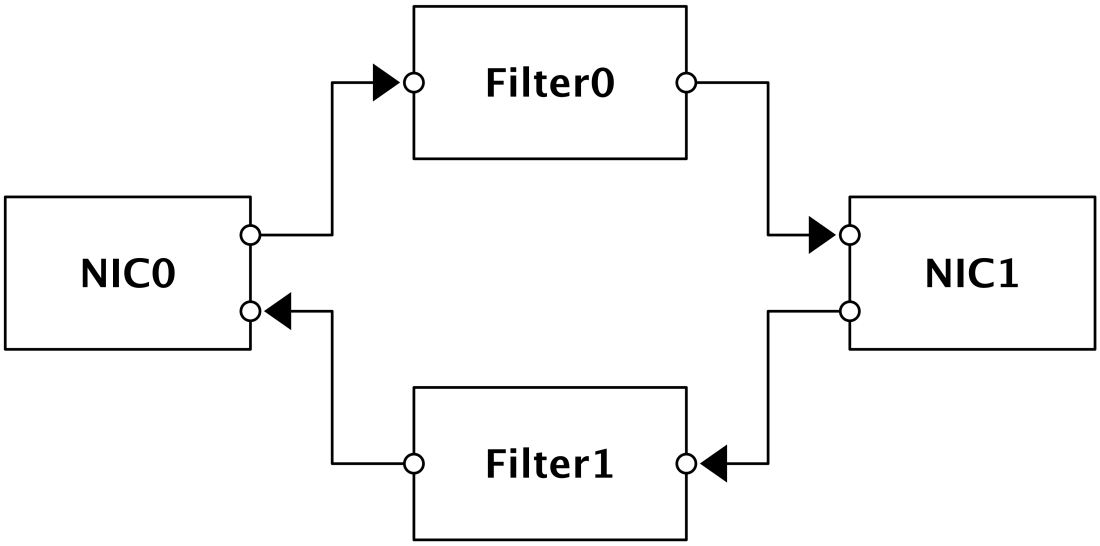

Figure 2.2: Example Snabb app network [1]

2.2 Snabb Apps

Apps are compact Lua modules that define a set of inputs and/or outputs for packets. Snabb itself offers a large variety of apps but also allows users to implement their own apps.

Apps need to expose a `:new()` method that returns a new app instance. To interact with other apps an app instance needs to have either a `:pull()` method, a `:push()` method or both.

Snabb abstracts many different network functions using apps: NICs are exposed as apps with their drivers being written in Lua and included in the app; Pcap files can be read or written using apps; packets can be filtered or modified using apps.

```
1  module(..., package.seeall)
2
3  local link = require("core.link")
4
5  Fwd = {}
6
7  function Fwd:new()
8    return setmetatable({transmitted = 0}, {__index = Fwd})
9  end
10
11 function Fwd:pull()
12   assert(self.output.output, "Fwd: output link not created")
13   assert(self.input.input, "Fwd: input link not created")
14   local n = link.nreadable(self.input.input)
15   for _ = 1, n do
16     link.transmit(self.output.output, link.receive(self.input.input))
17   end
18   self.transmitted = self.transmitted + n
19 end
20
21 function Fwd:report()
22   print(string.format("Fwd '%s' transmitted %d packets",
23     self.appname, self.transmitted))
24 end
```

Figure 2.3: Snabb forward app example

Figure 2.3 is a simple example app that just outputs all packets it receives on its input link.

Line 14 queries the input link and thereby determines the amount of packets currently stored in it. Lines 15-17 loop over the packets in the link and forward each packet onto the output link. Line 18 additionally updates an app-internal packet counter.

In addition to its `:pull()` method it also defines an optional `:report()` method that gives the user feedback on the app's traffic statistics. Snabb Core calls this method after terminating the main loop.

2.3 Snabb Core

Snabb Core is Snabb's runtime. It's responsible for initializing, starting and stopping apps, connecting them and finally running the app network. Snabb runs apps in a single loop, repeatedly calling each app's `:push()` and `:pull()` methods. Snabb calls this behavior *breathing*.

Snabb's runtime may be launched multiple times in sequence with different app network

configurations in a single Snabb program. The `core.app` (called *engine* in Snabb's reference manual, and henceforth in this thesis) module's `config()` and `main()` functions reconfigure and run the network respectively.

Snabb Core initializes each app's `input` and `output` tables according to the configuration given to `engine.config()` by the user program. Apps can read from and write to the links in these tables using the functions provided by the `core.link` module.

Apps are connected using ring buffers (called *links*) from which exactly one app reads packets and onto which exactly one app writes packets. Links are fixed in size, dropping new packets when full.

Packets are represented as simple structures containing a `length` field and a `data` pointer. LuaJIT's foreign function interface allows easy access to these structures. Snabb maintains an array of empty packet buffers to speed up allocation.

2.4 Snabb Programs

Snabb Programs are the user-defined Lua scripts that instruct Snabb Core to configure and run the app network.

Usually Snabb programs contain a `run()` function which acts as the entrypoint. Snabb passes the command line arguments into this function. Snabb offers basic command line parsing functionality via Snabb Lib (see 2.5). The user program may be any arbitrary Lua program. Snabb's example programs contain both very simple pcap replay programs as well as large and complicated systems such as an implementation[1] of the lwAFTR data-plane part of the "Lightweight 4over6" IPv6-transition mechanism[2].

Figure 2.4 is an example Snabb program that connects a forward app and an Intel82599 app. This creates a fast echo server that replays all packets it receives.

Lines 10-22 parse the command line options given by the user. Lines 26-30 create an empty Snabb configuration and then proceed to add an app for the NIC as well as a number of forward apps (see Figure 2.3). In lines 32-37 all apps are connected with links. Finally, in lines 41-44, the configuration is loaded into the engine and run.

The programmer has to bind the program to the appropriate CPU manually using Snabb's `lib.numa` library.

[1] `https://github.com/snabbco/snabb/tree/master/src/program/lwaftr`
[2] `https://tools.ietf.org/html/rfc7596#section-6.2`

```
module(..., package.seeall)

local fwd = require("apps.fwd.fwd")
local intel = require("apps.intel.intel_app")
local numa = require("lib.numa")

function run(parameters)
  print("running echo")

  if #parameters > 4 or #parameters < 1 then
    print("Usage: echo <pci-addr> [chain-length] [duration] [cpu]\nexiting...")
    main.exit(1)
  end

  local pciaddr = parameters[1]
  local chainlen = tonumber(parameters[2]) or 1
  local cpu = tonumber(parameters[4])

  if chainlen < 1 then
    print("chain-length < 1, defaulting to 1")
    chainlen = 1
  end

  local c = config.new()

  config.app(c, "intel", intel.Intel82599, {pciaddr = pciaddr})

  for i = 1, chainlen do
    config.app(c, "fwd" .. i, fwd.Fwd)
  end

  for i = 1, chainlen - 1 do
    config.link(c, string.format("fwd%d.output -> fwd%d.input", i, i + 1))
  end

  config.link(c, "intel.tx -> fwd1.input")
  config.link(c, string.format("fwd%d.output -> intel.rx", chainlen))

  -- need to manually bind to the appropriate CPU core
  -- this needs to be determined by the user
  if cpu then numa.bind_to_cpu(cpu) end
  engine.configure(c)
  engine.busywait = true
  engine.main{duration = tonumber(parameters[3])}
end
```

Figure 2.4: Snabb echo server program

2.5 Snabb Lib

Snabb Lib is a set of libraries bundled with Snabb. It includes functionality for building packets, timers, checksumming etc. It uses some C code as well as LuaJIT's DynASM [7] dynamic assembler. Many of Snabb's bundled apps use Snabb Lib.

Chapter 3

libmoon

3.1 Data Plane Development Kit (DPDK)

DPDK is a networking toolkit targeted at high-performance packet processing without using dedicated hardware. It provides a large set of drivers for a variety of NICs.[1] It runs on Linux computers (x86-64, ARM and POWER), although a subset of its functionality is also available on FreeBSD. In this thesis we only used x86-64 Linux computers and Intel ixgbe/i40e-compatible NICs.

DPDK achieves its high performance by using *batching*, *polling* and through custom drivers supporting a variety of hardware acceleration mechanisms.

Batching refers to processing a batch of packets at once, thereby greatly decreasing the per-packet overhead. Polling is a technique whereby the CPU repeatedly checks for incoming packets on the NIC as fast as possible, similar to busy-waiting. With interrupt-driven I/O the operating system has to call the interrupt handler which in turn calls the receiving process, introducing unnecessary overhead due to additional code and context switches. Polling uses 100% of CPU time increasing power consumption and preventing the CPU from doing other work.

3.2 libmoon

libmoon is a Lua wrapper around DPDK. It exposes many DPDK functions and features to Lua scripts.

Building libmoon yields a binary that serves as a replacement for LuaJIT and automatically loads libmoon's libraries into the interpreter.

[1] `https://dpdk.org/doc/nics`

These libraries include abstractions for DPDK NICs, many DPDK functions and structures as well as utility functions for allocating memory, creating and parsing protocol headers, etc.

libmoon automatically schedules each thread on the appropriate NUMA node/CPU core.

Chapter 4

snabb-libmoon-compat

4.1 Approach

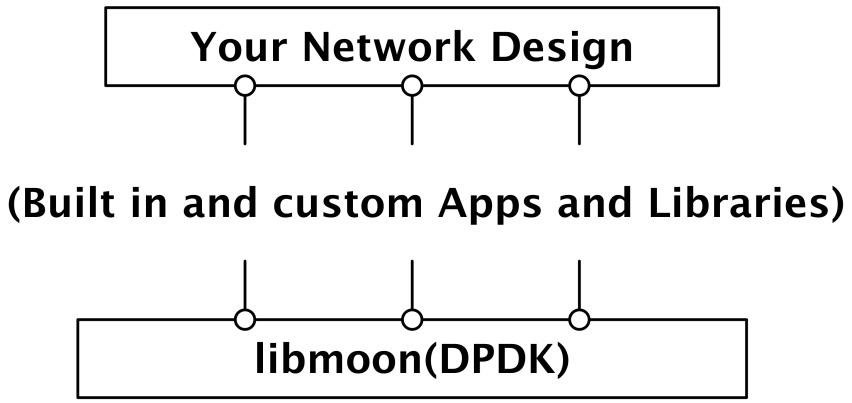

Figure 4.1: snabb-libmoon-compat structure (adapted from Figure 2.1)

The chosen approach to create a Snabb-API-compatible runtime is a three-part solution:

- Implementing a Snabb Core-compatible runtime
- Exposing DPDK NICs as Snabb apps
- Implementing a Snabb-API-compatible library

To achieve this snabb-libmoon-compat was created. snabb-libmoon-compat is both a fork and a rewrite of Snabb: It reuses some Snabb code with modifications while some modules were reimplemented entirely from scratch. It has a similar directory structure to Snabb but was adapted to be run using libmoon. Within the source directory there are directories for included apps (src/apps), included libraries (src/lib), included programs (src/program) as well as the runtime (src/core). Each app and program is contained in its own subdirectory. snabb-libmoon-compat contains no C code; everything is written in Lua. Therefore there is no need to compile it; it can simply be run using the LuaJIT embedded into the libmoon binary.

Upon startup snabb-libmoon-compat spawns a libmoon slave task that runs the actual snabb program while the master task does scheduling, facilitated by libmoon. The slave task then loads similar libraries into global scope as Snabb and launches the Snabb user program.

4.2 DPDKDevice

Due to Snabb's simple API DPDK devices were easily exposed as Snabb apps.

`DPDKDevice:new()` simply calls libmoon's device configuration function, thereby initializing the chosen NIC, setting up a single receive queue and a single transmit queue as well as allocating a memory pool for packets. `DPDKDevice:push()` simply reads packets from the app's `input.input` link and calls its transmit queue's `:sendN()` method. Similarly `DPDKDevice:pull()` just calls its receive queue's `:tryRecv()` method and forwards the received packets onto the `output.output` link.

Figure 4.2 shows the entire DPDKDevice implementation.

```
1  module(..., package.seeall)
2
3  local lm = require "libmoon"
4  local device = require "device"
5  local memory = require "memory"
6  local log = require "log"
7  local link = require "core.link"
8  local nreadable, receive, transmit =
9    link.nreadable, link.receive, link.transmit
10
11 DPDKDevice = {}
12
13 function DPDKDevice:new(dev_id)
14   local dev = device.config{
15     port = dev_id,
16     txQueues = 1,
17     rxQueues = 1,
18     disableOffloads = true
19   }
20   local obj = {
21     dev = dev,
22     tx = dev:getTxQueue(0),
23     rx = dev:getRxQueue(0),
24     bufs = memory.bufArray(link.max),
25   }
26   return setmetatable(obj, {__index = DPDKDevice})
27 end
28
29 function DPDKDevice:push()
30   local n = nreadable(self.input.input)
31   for i = 1, n do -- read all packets from the input link
32     self.bufs[i] = receive(self.input.input)
33   end
34   self.tx:sendN(self.bufs, n) -- forward all packets onto the wire
35 end
36
37 function DPDKDevice:pull()
38   -- receive a burst of packets
39   local rx = self.rx:tryRecv(self.bufs, 0)
40   for i = 1, rx do -- forward all packets onto the output link
41     transmit(self.output.output, self.bufs[i])
42   end
43 end
44
45 function DPDKDevice:report()
46   local stats = self.dev:getStats()
47   log:info("DPDKDevice '%s' sent %d packets and received %d packets",
48     self.appname, tonumber(stats.opackets), tonumber(stats.ipackets))
49 end
```

Figure 4.2: DPDKDevice implementation

4.3 Runtime

The runtime consists of several modules implementing links, packets, a basic utility library as well as the engine.

Links reuse Snabb's implementation with some small modifications. Therefore their performance is the same as Snabb.

Packets were implemented similarly to Snabb with only minor modifications being necessary. Snabb represents packets as simple structures consisting of a length field and a pointer to the packet's contents. libmoon uses DPDK's more complex `rte_mbuf` structure and offers setter and getter functions for most fields. An `rte_mbuf` contains additional metadata besides the packet's length and data such as a timestamp. Since libmoon already provides means for allocating and freeing packet buffers snabb-libmoon-compat's packet module is actually smaller and less complex than Snabb's version.

Only functions necessary for Snabb's protocol stack were copied from Snabb's codebase into snabb-libmoon-compat's `core.lib` module.

The engine implementation was adapted from Snabb's engine. The engine parses the given configuration and creates a graph with apps being nodes and links being edges. From this graph it computes the necessary setup actions, performs them and then runs the main loop. Large parts of Snabb's engine could be reused such as the graph computation and app configuration functions.

4.4 Library

Snabb's standard library is large, containing a multitude of protocol implementations, drivers and utility functions. Implementing all of these is out of scope for this thesis. Instead only a subset was ported or implemented allowing basic Snabb programs to be run on snabb-libmoon-compat with little modification.

Snabb's protocol stack could be reused almost entirely without modification. These modules contain functionality for assembling packets from headers using simple constructors. In addition to that libmoon's protocol stack is also entirely available to users.

Generally snabb-libmoon-compat users have access to libmoon's entire functionality except multithreading. Currently multithreading is not possible due to the snabb-libmoon-compat runtime running in a libmoon slave thread. libmoon only allows the master thread to spawn new threads. It should be possible to move the runtime into libmoon's master thread (see 6.2).

4.5 Compatibility

Figure 4.3 is a snabb-libmoon-compat-compatible version of Chapter 2's Figure 2.3. It required no modification at all although through snabb-libmoon-compat it can take advantage of libmoon's more advanced logging module.

```
1  module(..., package.seeall)
2
3  local log = require("log")
4  local link = require("core.link")
5
6  Fwd = {}
7
8  function Fwd:new()
9    return setmetatable({transmitted = 0}, {__index = Fwd})
10 end
11
12 function Fwd:pull()
13   if not self.output.output then
14     log:fatal("Fwd: output link not created")
15   elseif not self.input.input then
16     log:fatal("Fwd: input link not created")
17   end
18   local n = link.nreadable(self.input.input)
19   for _ = 1, n do
20     link.transmit(self.output.output, link.receive(self.input.input))
21   end
22   self.transmitted = self.transmitted + n
23 end
24
25 function Fwd:report()
26   log:info("Fwd '%s' transmitted %d packets",
27     self.appname, self.transmitted)
28 end
```

Figure 4.3: snabb-libmoon-compat forward app example

Figure 4.4 is a snabb-libmoon-compat-compatible version of Chapter 2's Figure 2.4. It, too, is almost identical with only the Intel82599-App having been replaced with the DPDKDevice-App. Snabb requires the NIC to be specified by its PCIe-address whereas DPDK uses simple numbering.

```
module(..., package.seeall)

local fwd = require("apps.fwd.fwd")
local dpdkdev = require("apps.dpdk_device.dpdk_device")
local log = require("log")

function run(parameters)
  log:info("running echo")

  if #parameters > 3 or #parameters < 1 then
    log:warn("Usage: echo <dev-id> [chain-length] [duration]\nexiting...")
    main.exit(1)
  end

  local devid = tonumber(parameters[1])
  local chainlen = tonumber(parameters[2]) or 1

  if chainlen < 1 then
    log:warn("chain-length < 1, defaulting to 1")
    chainlen = 1
  end

  local c = config.new()

  config.app(c, "dpdk", dpdkdev.DPDKDevice, devid)

  for i = 1, chainlen do
    config.app(c, "fwd" .. i, fwd.Fwd)
  end

  for i = 1, chainlen - 1 do
    config.link(c, string.format("fwd%d.output -> fwd%d.input", i, i + 1))
  end

  config.link(c, "dpdk.output -> fwd1.input")
  config.link(c, string.format("fwd%d.output -> dpdk.input", chainlen))

  -- no need to bind to the appropriate CPU core
  -- libmoon does that for us
  engine.configure(c)
  engine.main{duration = tonumber(parameters[3])}
end
```

Figure 4.4: snabb-libmoon-compat echo program example

Snabb module	compatibility level
`core.config`	full
`core.app`	partial
`core.link`	full
`core.packet`	full
`core.memory`	none
`core.shm`	partial
`core.counter`	full
`core.histogram`	none
`core.lib`	partial
`core.worker`	none
`main`	full
`lib.protocol`	full
`apps.pcap`	full

Table 4.1: snabb-libmoon-compat API compatibility

Most basic Snabb apps, that don't depend on C code, and some Snabb libraries should run out of the box or require minimal porting effort. Snabb programs will have to be modified to replace Snabb's driver apps (such as Intel82599 or Intel10G) with snabb-libmoon-compat's DPDKDevice app. Obviously Snabb's own apps that depend on not yet ported libraries won't work out of the box.

Table 4.1 shows snabb-libmoon-compat's API compatibility. All modules not mentioned are not implemented. `core.shm` and `core.counter` are not thread-safe and have only been included for compatibility with existing Snabb apps. Please note that, should Snabb Pull Request #1320[1] be accepted, `apps.pcap` will only be partially compatible with Snabb.

Additionally a set of example programs and apps has been included with snabb-libmoon-compat. These include simple pcap replay, a basic packet capture utility as well as a simple but slow packet generator.

4.6 License

snabb-libmoon-compat will be released under the Apache 2.0 license[2][3], same as Snabb. All files and functions taken from Snabb have been marked as such. Modified files have been marked with "Modifications by Fabian Bonk." These are

- src/core/link.lua

[1] `https://github.com/snabbco/snabb/pull/1320`
[2] `https://www.apache.org/licenses/LICENSE-2.0`
[3] LICENSE file in snabb-libmoon-compat's source tree

- src/core/app.lua
- src/core/packet.lua
- src/core/config.lua
- src/core/lib.lua
- all of src/lib/protocol
- src/lib/lua/class.lua

snabb-libmoon-compat is available on GitHub[4] and the author's homepage[5].

[4] https://github.com/Reperator/snabb-libmoon-compat/
[5] https://fabianbonk.de/snabb-libmoon-compat/

Chapter 5

Performance

5.1 Hardware

Snabb's and snabb-libmoon-compat's performance was compared on three test systems. The systems were chosen because they represent old consumer, high-end workstation, low-end server and high-end server CPUs. See Table 5.1 for an overview of the test systems.[1]

System A consisted of an Intel Xeon E5-2620 v3 [8] CPU and both an Intel XL710 [9] 40GbE NIC (snabb-libmoon-compat only) and an Intel X520-T2 [10] 10GbE NIC as well as 32 GiB of DDR4-2133 quad-channel ECC memory.

System B consisted of dual Intel Xeon E5-2630 v4 [11] CPUs as well as the same NICs as system A. Each CPU has 64 GiB of DDR4-2133 quad-channel ECC memory for a total of 128 GiB.

System C consisted of an Intel i7-3770K [12] (overclocked to 4.5 GHz) and 16 GiB of DDR3-2000 dual-channel non-ECC memory.

System D consisted of an AMD Ryzen Threadripper 1950X [13] and 32 GiB of DDR4-3200 quad-channel non-ECC memory.

Systems C and D used Intel X520-DA2 [14] and X520-DA1 [15] (dual-port/single-port versions of the same NIC) 10GbE NICs respectively.

All systems ran Debian Linux 4.9.0 and slight modifications of the latest development versions of Snabb[2] and libmoon[3]. The modifications were necessary to add important testing functionality.

[1] The given CPU frequencies are the maximum single core turbo frequencies. AMD's XFR technology can boost the 1950X to up to 4.2 GHz.

[2] `https://github.com/Reperator/snabb`; commit `bee107d9f80b18dae4b88612363406eaf049083e`

[3] `https://github.com/Reperator/libmoon`; commit `9c093d111cd8037b24ecd51bfdc1f35ef2aeb3e4`

System	CPU	CPU freq.	L3 cache	NIC
A1	Intel Xeon E5-2620 v3	3.2 GHz	15 MiB	Intel XL710
A2	Intel Xeon E5-2620 v3	3.2 GHz	15 MiB	Intel 82599ES
B1	2 Intel Xeon E5-2630 v4	3.1 GHz	25 MiB	Intel XL710
B2	2 Intel Xeon E5-2630 v4	3.1 GHz	25 MiB	Intel 82599ES
C	Intel i7-3770K	4.5 GHz	8 MiB	Intel 82599ES
D	AMD Ryzen Threadripper 1950X	4.2 GHz	32 MiB	Intel 82599ES

Table 5.1: Test system specifications

5.2 Methodology

Both Snabb and snabb-libmoon-compat were tested by generating a large amount of small Ethernet packets (60 byte payload, 80 byte including preamble, start of frame delimiter and interpacket gap) on a separate machine using libmoon's examples/pktgen.lua. Traffic was always generated at a higher rate than Snabb or snabb-libmoon-compat could forward by using multiple CPU cores on the generating machine while the programs under test always ran on just one core.

Systems A and B as well as systems C and D generated traffic for each other.

The tested Snabb programs were chains of forward apps (see Figures 2.3 and 4.3) of length 1 through 100 with one DPDKDevice app (snabb-libmoon-compat) or one Intel82599 app (Snabb) at the ends of the chain. Figures 2.4 and 4.4 contain the snabb programs that were used. Snabb's `engine.busywait` feature was activated, causing Snabb to also employ polling. Snabb was manually bound to the best-performing core. Figure 5.1 shows the structure of the snabb program that ran on the test systems.

The final results are the peak packet rate measurements created by libmoon's `stats` feature on the generating machine after ~10 seconds of full load. The peak packet rate is usually within 1% of the average packet rate of the run.

Snabb does not support any 40GbE NICs.

Figure 5.1 details the testing setup.

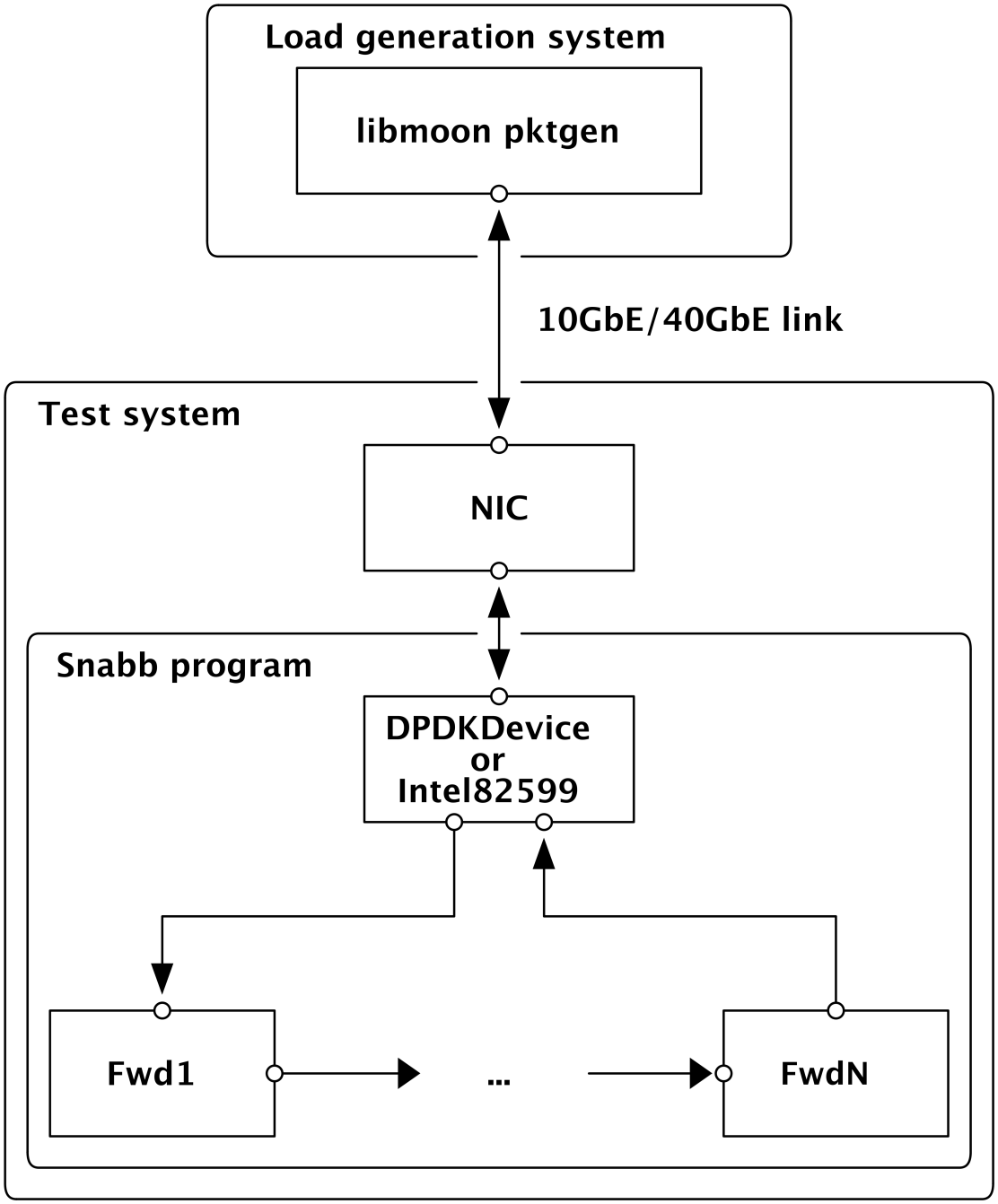

Figure 5.1: Performance measurement setup

5.3 Results

Snabb's results are green; snabb-libmoon-compat's results are blue (dark blue for 40GbE results). For each system there are two graphs: The first one showing the range from 1 through 100 forward apps; the second one detailing the more important range from 1 through 20 forward apps.

In our tests Intel XL710 NICs were not able to achieve packet rates higher than ~30 Mpps. Therefore this is the cutoff point of the graphs.

5.3.1 System A

See Figure 5.2.

On system A Snabb never reached 10GbE wire-rate. snabb-libmoon-compat was able to keep wire-rate at chain length 1 but dropped off soon afterwards. At around chain length 50 memory bandwidth became the limiting factor for both Snabb and snabb-libmoon-compat.

Using 40GbE NICs allowed for a maximum packet rate of 17.34 Mpps which equates to a ~16% improvement over 10GbE line rate (Snabb's maximum rate).

5.3.2 System B

See Figure 5.3.

System B behaved similarly to system A.

Going from a 10GbE NIC to a 40GbE NIC yielded an even larger performance increase than on system A. A maximum packet rate of 18.65 Mpps was measured, a ~25% performance increase over 10GbE wire-rate.

5.3.3 System C

See Figure 5.4.

System C's maximum performance was limited due to lack of support for Direct Cache Access (DCA). The NIC can only use Direct Memory Access (DMA) to write packets into system RAM from which the CPU needs to retrieve it manually. The other systems all support DCA, so their NICs write packets directly into the CPUs level 3 cache. Additionally system C's CPU is old (6 years at the time of writing) and only supports DDR3 memory instead of the newer DDR4 standard, limiting bandwidth even further.

Therefore both Snabb and snabb-libmoon-compat were capped at ~8 Mpps, though snabb-libmoon-compat was able to keep this rate at higher chain lengths.

5.3.4 System D

See Figure 5.5.

System D had the best 10GbE performance of all test systems. snabb-libmoon-compat was able to keep wire-rate until a chain-length of 10. A probable cause for this may be the CPUs high frequency.

5.3.5 Evaluation

snabb-libmoon-compat was consistently able to match or outperform Snabb. DPDK's high-performance drivers yielded both improvements at 10GbE as well as opened up the possibility of using 40GbE NICs. 40GbE NICs were able to increase the maximum single core forwarding rate from 14.88 Mpps to a peak of 18.65 Mpps. This equates to a performance advantage of 25% over Snabb at a chain length of 1.

System D achieved a peak performance advantage over Snabb of ~40% at a chain length of 10.

At very long chain lengths of more than 50 memory bandwidth became the limiting factor for both Snabb and snabb-libmoon-compat. Very long chains are unusual but may be required for certain setups (see [16]).

Much larger performance improvements can in theory be achieved by running multiple instances of the app network in separate threads on separate CPU cores, though this requires major modifications to snabb-libmoon-compat (see 6.2).

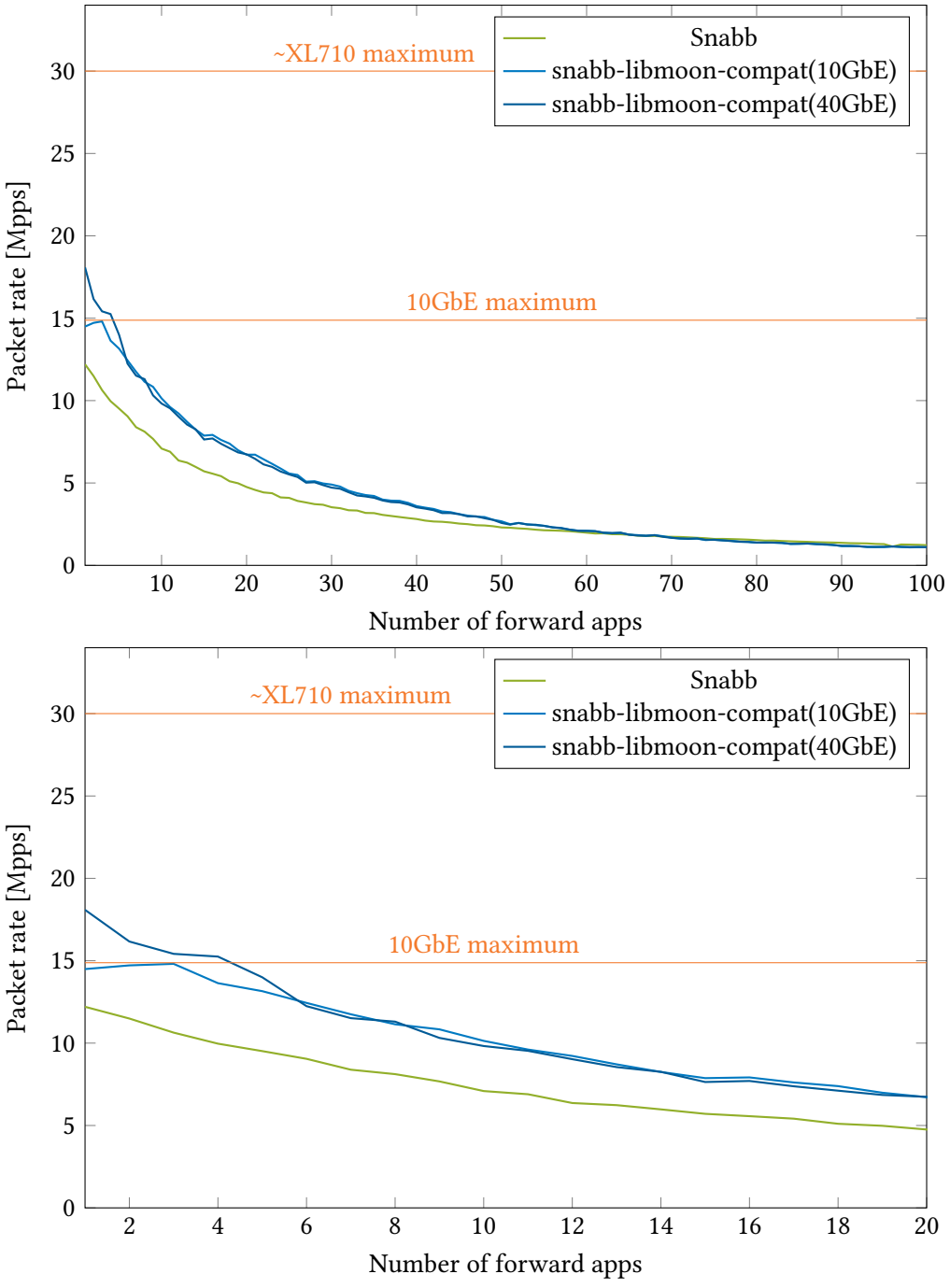

Figure 5.2: System A results

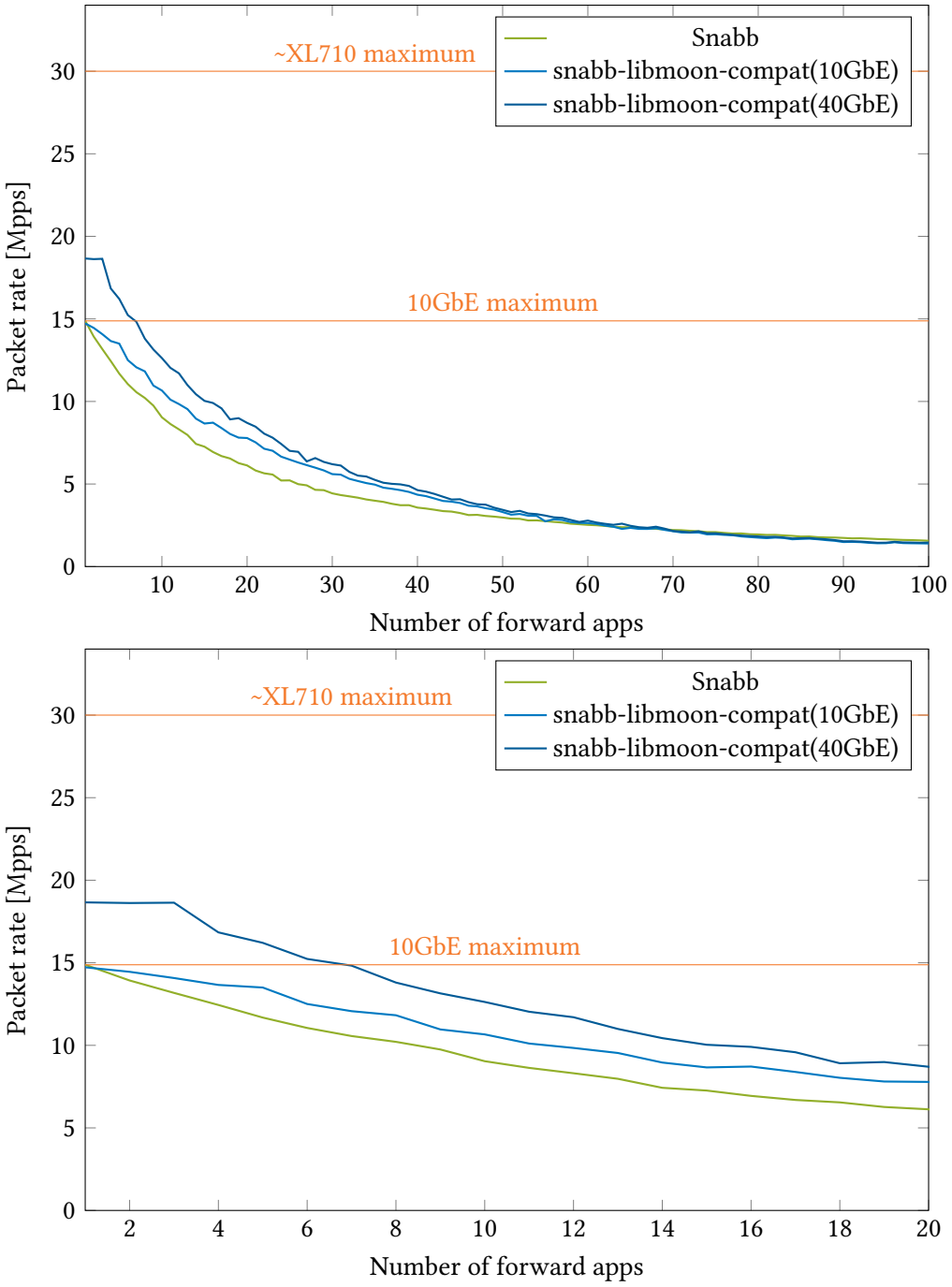

Figure 5.3: System B results

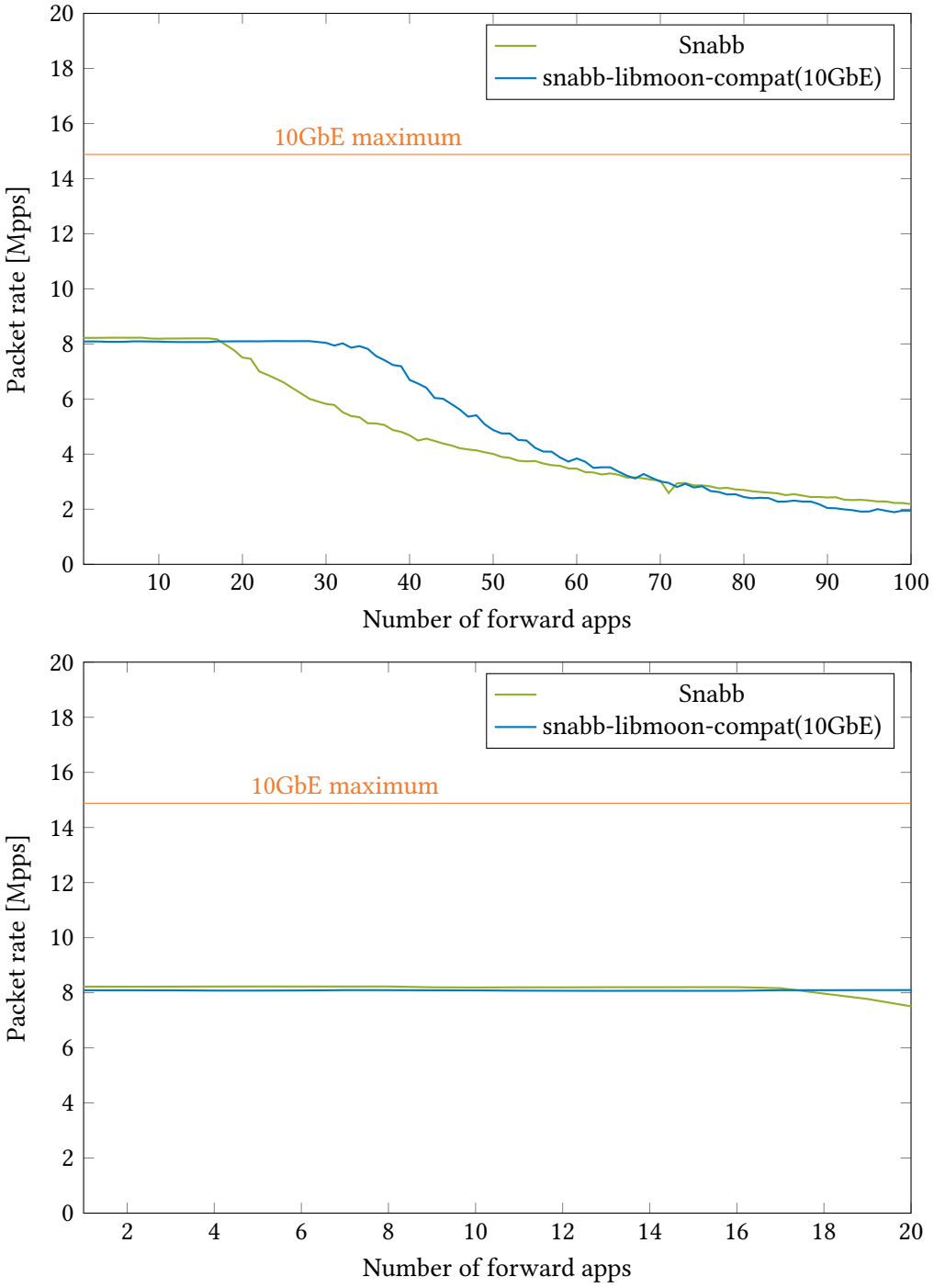

Figure 5.4: System C results

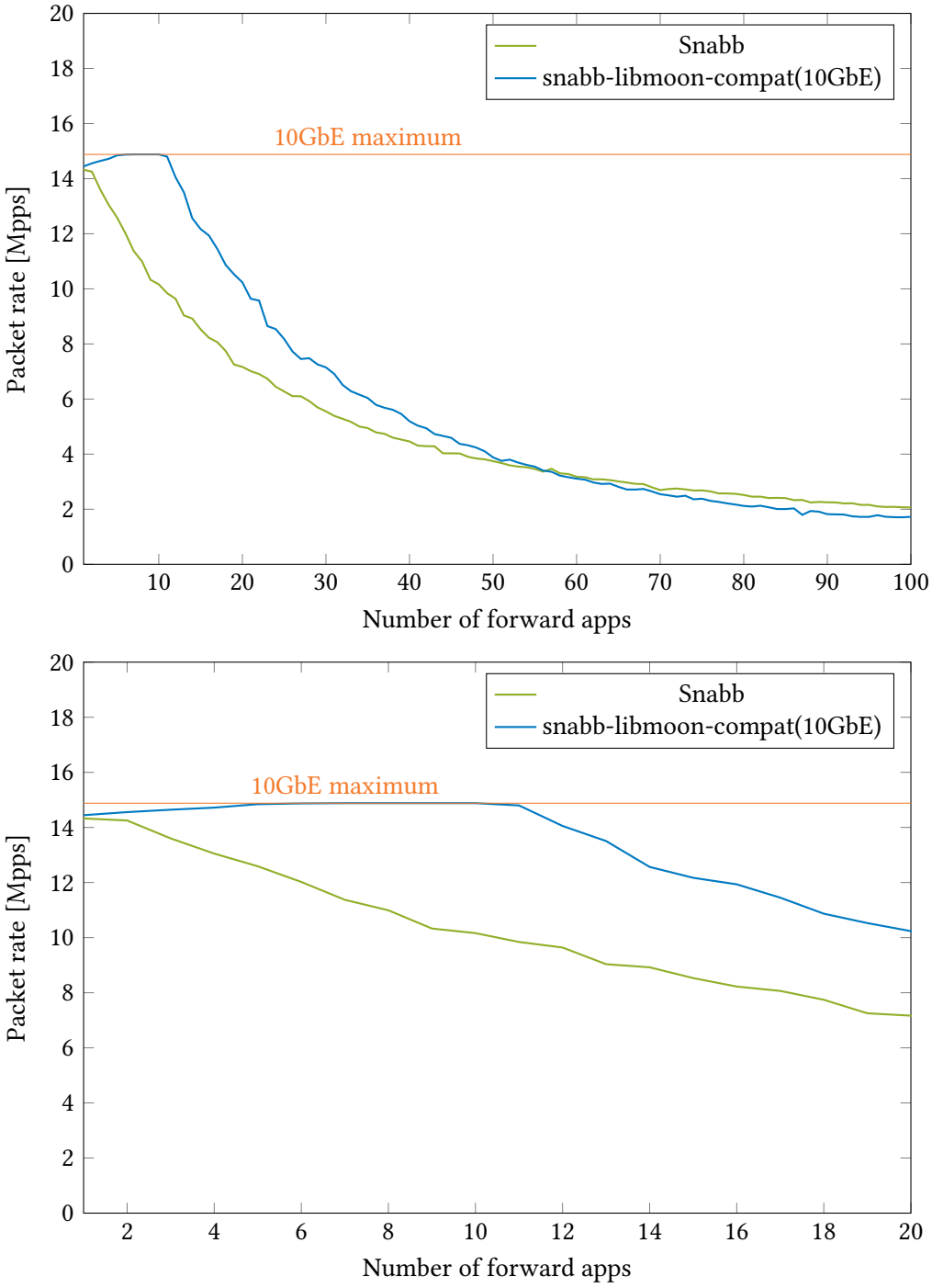

Figure 5.5: System D results

Chapter 6

Conclusion

6.1 Results

In this work we ported Snabb's runtime to libmoon/DPDK by partially reimplementing its components and libraries. Users are now able to benefit from libmoon's features as well as DPDK's larger hardware support. Programs can now use all 40GbE and 100GbE NICs supported by libmoon.

Currently most apps only work with some modification. snabb-libmoon-compat's packet structure can only be used via libmoon's accessors; Snabb's fields have different names. Theoretically this could be fixed by modifying libmoon or possibly by creating a pseudo-`rte_mbuf` structure with different field names.

The thesis' goal of accelerating Snabb programs was achieved: By adding support for faster NICs throughput improved by up to 25%. Additionally overhead in app networks up to 50 apps was slightly reduced. Real-world Snabb programs usually have additional functionality besides simple forwarding; this additional overhead cannot easily be reduced, making multi-core scaling a necessary next step.

6.2 Future work

Possible directions of future work include implementing a larger set of Snabb's libraries and making use of libmoon's native multithreading to improve performance.

API-compatibility can be improved by either modifying Snabb's code, by mapping functions onto libmoon or by reimplementing functionality entirely.

Theoretically large performance gains are possible by spawing a thread for each app or worker threads for a group of related apps. Most apps are independent of each other and can therefore run in separate threads without risk of race conditions. The

single-producer, single-consumer structure of links should lend itself to multithreading. Within threads Snabb's simple and fast links could be used without synchronization; when connecting apps from different threads DPDK's `rte_ring` structure could be used. This structure is already available through libmoon's `pipe` module. If more apps can be bundled into one thread these apps will not suffer from synchronization overhead. Cache performance and possible dependencies between apps need to be taken into account but further exploration in this direction seems promising.

Another requirement is testing. Currently snabb-libmoon-compat is very much a prototype and as such it has no testsuite. Due to this prototype status it should not be used in production environments.

API Documentation is required. snabb-libmoon-compat's API differs in some aspects from Snabb and therefore documentation needs to be adjusted or rewritten.

Lastly snabb-libmoon-compat's user interface needs improvement. The allowed user program structure is less flexible than Snabb. Currently each user program needs to be in a separate directory with matching directory and file names.

Bibliography

[1] The Snabb Authors, *Snabb Reference Manual*, https://snabbco.github.io/.

[2] Luke Gorrie et al., "Snabb," https://github.com/snabbco/snabb.

[3] Mike Pall, "LuaJIT," https://luajit.org/.

[4] ——, *LuaJIT ffi.* API Functions*, https://luajit.org/ext_ffi_api.html.

[5] Paul Emmerich, "libmoon," https://github.com/libmoon/libmoon.

[6] Linux Foundation, "Data Plane Development Kit," https://dpdk.org/.

[7] Mike Pall, "DynASM," https://luajit.org/dynasm.html.

[8] "Intel Xeon Processor E5-2620 v3," https://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz.

[9] "Intel Ethernet Converged Adapter XL710-QDA2," https://ark.intel.com/products/83967/Intel-Ethernet-Converged-Network-Adapter-XL710-QDA2.

[10] "Intel Ethernet Server Adapter X520-T2," https://www.intel.de/content/dam/doc/product-brief/ethernet-server-adapter-x520-t2-brief.pdf.

[11] "Intel Xeon Processor E5-2630 v4," https://ark.intel.com/products/92981/Intel-Xeon-Processor-E5-2630-v4-25M-Cache-2_20-GHz.

[12] "Intel Core i7-3770K Processor," https://ark.intel.com/products/65523/Intel-Core-i7-3770K-Processor-8M-Cache-up-to-3_90-GHz.

[13] "AMD Ryzen Threadripper 1950X," https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-1950x.

[14] "Intel Ethernet Converged Adapter X520-DA2," https://ark.intel.com/products/39776/Intel-Ethernet-Converged-Network-Adapter-X520-DA2.

[15] "Intel Ethernet Converged Adapter X520-DA1," https://ark.intel.com/products/68669/Intel-Ethernet-Converged-Network-Adapter-X520-DA1.

[16] W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, and T. Wood, "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization," in *Proceedings*

of the 12th International on Conference on emerging Networking EXperiments and Technologies, CoNEXT 2016, Irvine, California, USA, December 12-15, 2016, 2016. [Online]. Available: http://faculty.cs.gwu.edu/timwood/papers/16-CoNext-flurries.pdf