Floem: Programming System for NIC-Accelerated Network Applications

Phitchaya Mangpo Phothilimthana
University of California, Berkeley

Ming Liu, Antoine Kaufmann, Ras Bodik, Tom Anderson
University of Washington

Simon Peter
UT Austin
Ethernet vs. CPU

Data-Center Network
Ethernet vs. CPU

Data-Center Network

100x

1 Gbps → 100 Gbps
Ethernet vs. CPU

Data-Center Network

100X
1 Gbps → 100 Gbps

< 2X
3 GHz → 5 GHz
Network Card (NIC)

**Wimpy multi-core processor**
- Cavium LiquidIO
- Netronome Agilio
- Mellanox BlueField

**Field-programmable gate array (FPGA)**
- Microsoft Catapult
- NetFPGA

**Reconfigurable Match Table (RMT)**
- Bosshart et al. 2013
- Kaufmann et al. 2015
NIC Offload

Offload Computation

- Fast path processing: filtering, classifying, caching, etc.
- Transformation: encryption/decryption, compression, etc.
- Steering
- Congestion control
Offloading computation to a NIC requires a large amount of effort.
No cache coherence.
NIC can access CPU memory via PCIe.

**Cavium LiquidIO**
- Slower cores
- Lower power
- No floating-point
- Encryption co-processor
- L1/L2 cache, DRAM, host memory

**Intel Xeon**
- Faster cores
- Higher power
- Floating-point support
- HW-accelerated instructions
- L1/L2/L3 cache, DRAM, disk
**Space of Offload Designs**

**Example: Key-value store**

- Hash & steer pkt
- Prepare response

**Key-based steering**
- 30-45% higher throughput
- Require: multiple CPU cores
- [Kaufmann et al. 2016]

**Using NIC as Cache**
- 3x power efficiency
- Require: enough memory on NIC
- [Li et al. 2017]
No one-size-fit-all offload.
Non-trivial to predict which offload is best.
Challenge: Packet Marshaling

Example: Key-value store

// Define what fields to send
struct set_request_entry {
    uint16_t flags;
    uint16_t len;
    uint32_t hash;
    uint32_t keylen;
    uint32_t vallen;
    uint8_t other[];
}

// Copy those fields
extra = it->keylen + it->vallen;
entry = queue_alloc(sizeof(*entry) + extra, SET);
entry->hash = hash;
entry->keylen = it->keylen;
entry->vallen = it->vallen;
memcpy(entry->other, it->key, extra);

struct set_request_entry {
    uint16_t flags;
    uint16_t len;
    uint64_t item;
}

entry = queue_alloc(sizeof(*entry), SET);
entry->item = ialloc_to_offset(item);

Example: Key-value store

hash, key, value, keylen, vallen

create item

update table

item pointer

hash

create item

update table
Challenge: Communication Strategies

Example: Key-value store

- **Hash & Steer pkt**
  - **No steering**
  - **Key-based steering**
  - **Separate GET & SET**

- **KVS**

---

12
Exploring different offload designs requires a large amount of effort.
**Floem**

**DSL** makes it easy to explore alternative offloads. **Compiler** minimizes communication and generates efficient code. **Runtime** manages data transfer over PCIe.
Language Overview

Data-flow programming model
Language Overview

Data-flow programming model

- Extend to support:
  - Heterogeneity
  - Parallelism

Contributions

**Goal: Explore offload designs**

1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

**Goal: Integration with existing app**

4. Interface to external programs

Extend to support:
- Heterogeneity
- Parallelism
Compiler & Runtime

- Cache expansion
- Infer data transfer
- Data-flow to C

Floem program

External C program

Floem compiler

NIC C code

Cavium SDK

Floem NIC process

worker thread

runtime thread

Floem queue library

Floem queue synchronization

Cavium DMA primitives

NIC memory

NIC

Floem host process

worker thread

external program thread

Floem queue library

Host

host memory

CPU C code

GCC

External C program

- Cache expansion
- Infer data transfer
- Data-flow to C
Data-Flow Model

Contributions

**Goal: Explore offload designs**
1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

**Goal: Integration with existing app**
4. Interface to external programs

Data-flow programming model

Extend to support:
- Heterogeneity
- Parallelism
Data-Flow Model: Key-Value Store

```
class Classify(Element):
    def configure(self):
        self.inp = Input(pointer(kvs_message))
        self.get = Output(pointer(kvs_message))
        self.set = Output(pointer(kvs_message))

    def impl(self):
        self.run_c(r'''
            // C code
            kvs_message *p = inp();
            uint8_t cmd = p->mcr.request.opcode;
            output switch {
                // switch --> emit one output port
                case (cmd == PROTOCOL_BINARY_CMD_GET): get(p);
                case (cmd == PROTOCOL_BINARY_CMD_SET): set(p);
            }''')

    classify = Classify()  # Instantiate an element
```
Data-Flow Model: Key-Value Store

```
class Classify(Element):
    def configure(self):
        self.inp = Input(pointer(kvs_message))
        self.get = Output(pointer(kvs_message))
        self.set = Output(pointer(kvs_message))

    def impl(self):
        self.run_c(r'''
        // C code
        kvs_message *p = inp();
        uint8_t cmd = p->mcr.request.opcode;
        output switch {
            // switch --> emit one output port
            case (cmd == PROTOCOL_BINARY_CMD_GET): get(p);
            case (cmd == PROTOCOL_BINARY_CMD_SET): set(p);
        }''')

classify = Classify()  # Instantiate an element
```
Data-Flow Model

```python
@CPU
class Seg1(Segment):
    def impl(self):
        from_net >> hash >> classify
classify.get >> hasht_get >> get_resp >> to_net
classify.set >> item >> hasht_put >> set_resp \n    >> to_net
s1 = Seg1()
```

Segment

GET request

SET request

```python
from_net
hash
classify
classify.get
hasht_get
get_resp
to_net
create_item
hasht_put
set_resp
to_net
```

```plaintext
GET request

SET request
```
Data Parallelism

@CPU

from_net

hash

classify

GET request

create_item

hash_get

get_resp

to_net

SET request

hasht_put

set_resp

Seg1(cores=[0,1,2,3])
Pipeline Parallelism

Multiple Insert queues

Q1
- from_net
- hash
- classify
- hash_get
- create_item
- GET request

Q2
- hash_put
- set_resp
- get_resp
- to_net

Multiple segments
NIC Offload

@NIC

- from_net
- hash
- classify
- create_item
- hasht_get
- get_resp
- to_net
- hasht_put
- set_resp
- @CPU
- @NIC

Seg1(device=NIC)

Seg2(device=CPU)

Seg3(device=NIC)
Inferred Data Transfer

Data-flow programming model

Extend to support:
• Heterogeneity
• Parallelism

Contributions

Goal: Explore offload designs
1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

Goal: Integration with existing app
4. Interface to external programs

Extend to support:
• Heterogeneity
• Parallelism
**Per-packet state:** a packet and its metadata can be accessed anywhere in the program.

**Compiler** infers which fields of packet and metadata to send.

---

**Solution: Infer Fields to Send**

- `from_net`
- `hash`
- `classify`
- `hasht_get`
- `hasht_put`
- `Q1`
- `Virtual Channel C1`
  - `uint32_t hash;`
  - `uint16_t keylen;`
  - `uint8_t key[];`
- `Virtual Channel C2`
  - `item* it;`
- `GET request` to `classify`
- `SET request` from `classify`
- `create_item` to `hasht_put`
Logical-to-Physical Queue Mapping

Data-flow programming model

Extend to support:
- Heterogeneity
- Parallelism

Contributions

**Goal: Explore offload designs**
1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

**Goal: Integration with existing app**
4. Interface to external programs

Extend to support:
- Heterogeneity
- Parallelism
Observation: Different communication strategies can be expressed by mapping logical queues to physical queues.

- Degrees of resource sharing
- Dynamic packet steering
- Packet ordering

Solution: Queue construct with explicit logical-to-physical queue mapping.

Queue(**channels=2**, **instances=3**)
class Seg2(Segment):
    def impl(self):
        Q1.deq[0] >> hash_get >> ...
        Q1.deq[1] >> hash_put >> ...
Key-Based Steering

Strategy:

Program:

```
class Seg2(Segment):
    def impl(self):
        Q1.deq[0] >> hash_get >> ...
        Q1.deq[1] >> hash_put >> ...
```

…

GET request

CREATE item

…

hash & steer pkt

Q1

…

hash_get

hash_put

KVS

KVS

Key-based steering
Key-Based Steering

Strategy:

Program:

```python
class Seg2(Segment):
    def impl(self):
        self.core_id >> Q1.qid
        Q1.deq[0] >> hasht_get >> ...
        Q1.deq[1] >> hasht_put >> ...
```

Key-based steering:
```
state.qid = state.hv % 2
```
Caching Construct

Data-flow programming model

Extend to support:
• Heterogeneity
• Parallelism

Contributions

Goal: Explore offload designs
1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

Goal: Integration with existing app
4. Interface to external programs
Difficult to implement a complete cache protocol:
- Maintain consistency of data on NIC and CPU
- High performance
Get Expansion

User’s program

Write-through

Write-back

before get

get_start

get query

get_end

after get

before get

cache get

hit

get query

after get

miss

get query

cache set

after get

before get

cache get

miss

get query

before get

set query

before get

cache get

hit

set query

before get

cache set

miss

evict?

no

hit

yes

OR
Set Expansion

User’s program:
- before set
- set_start
- set query
- set_end
- after set

Write-through:
- before set
- cache del
- set query
- after set
- exe both

Write-back:
- before set
- cache set
- set query
- after set
- evict?
  - no
  - yes

OR
Runtime & Communication
Queue Implementation Challenge

For performance, require:
• I/O batching
• overlapping DMA operations with useful computation

NIC memory

DMA

PCIe

Host memory

Q1  Q2

Q1  Q2
Queue Implementation Challenge

Queue library +
data sync

Queue library

DMA primitives

DMA

PCIe

Host memory

NIC memory

Q1

Q2

Q1

Q2

P1

P3

P2
Queue Implementation Challenge

Queue library
Queue sync
DMA primitives

NIC memory

Q1

Q2

DMA

PCIe

Host memory

Q1

Q2

Queue library
Evaluation

Does Floem help programmers explore different offload designs?
Server Setup

With Smart NIC
- Cavium LiquidIO NIC
  - two 10Gbps ports
  - 12-core 1.20GHz cnMIPS64 processor
  - 4GB memory

Without Smart NIC
- Intel X710 NICs
  - two 10Gbps ports
  - DPDK (bypass OS networking stack)

6-core Intel X5650
Case Study: Key-Value Store

Workload
- 100,000 key-value pairs
- 32-byte keys and 64-byte values
- Zipf distribution ($s = 0.9$)
- 90% GET and 10% SET
Case Study: Key-Value Store

Higher is better.

Throughtput (Gbits/s)

Split CPU-NIC

Code relevant to communication

C program
+ 240 lines

Floem program
15 lines

CPU-only

@NIC
from_net
hash
classify
GET request
create_item
@CPU
hash_get
hash_put
GET request
@NIC
set_resp
to_net
Q1
Q2

Code relevant to communication

C program
+ 240 lines

Floem program
15 lines
Higher is better.

Throughput (Gbits/s)

- CPU-only
- split CPU-NIC

**Case Study: Key-Value Store**

### Code relevant to communication

- C program + 240 lines
- Floem program 15 lines

**Split CPU-NIC**

- from_net
- hash
- classify
- create_item
- SET request
- @NIC
- hash_get
- hash_put
- @CPU
- get_resp
- set_resp
- to_net
- @NIC
- Q1
- Q2
Higher is better.

Case Study: Key-Value Store

Add cache in Floem
40 lines
Case Study: Key-Value Store

Higher is better.

Caching on NIC

Add cache in Floem
40 lines
Case Study: Key-Value Store

Higher is better.

Change policy parameter
1 line

Caching on NIC

@NIC
from_net
hash
GET request
classify
SET request
get_start
set_start
Q1
@CPU
create_item
hasht_get
hash_get
hasht_put
set_start
Q2
to_net
get_end
get_resp
set_end
set_resp
Distributed real-time data analytics (Storm)
• First offload: worse than CPU-only
• Second offload: 96% improvement with 23 lines of code

Encryption
AES-CBC-128

Flow classification
Use a count-min sketch on the header 5-tuple

Network sequencer
Use a group lock
Conclusion

**Takeaway:** high-level programming abstractions
- control implementation strategies
- avoid low-level details

**Result:** minimal changes to explore different designs

[github.com/mangpo/floem](http://github.com/mangpo/floem)