Floem: A Programming System for NIC-Accelerated Network Applications

Phitchaya Mangpo Phothilimthana, ***University of California, Berkeley;*** **Ming Liu and Antoine Kaufmann,** ***University of Washington;*** **Simon Peter,** ***The University of Texas at Austin;*** **Rastislav Bodik and Thomas Anderson,** ***University of Washington***

https://www.usenix.org/conference/osdi18/presentation/phothilimthana

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Floem: A Programming System for NIC-Accelerated Network Applications

Phitchaya Mangpo Phothilimthana
University of California, Berkeley

Ming Liu
University of Washington

Antoine Kaufmann
University of Washington

Simon Peter
The University of Texas at Austin

Rastislav Bodik
University of Washington

Thomas Anderson
University of Washington

Abstract

Developing server applications that offload computation to a NIC accelerator is complex and laborious. Developers have to explore the design space, which includes semantic changes for different offloading strategies, as well as variations on parallelization, program-to-resource mapping, and communication strategies for program components across devices.

We therefore design FLOEM — a language, compiler, and runtime — for programming NIC-accelerated applications. FLOEM enables offload design exploration by providing programming abstractions to assign computation to hardware resources; control mapping of logical queues to physical queues; access fields of a packet and its metadata without manually marshaling a packet; use a NIC to memoize expensive computation; and interface with an external application. The compiler infers which data must be transferred between the CPU and NIC and generates a complete cache implementation, while the runtime transparently optimizes DMA throughput. We use FLOEM to explore NIC-offloading designs of real-world applications, including a key-value store and a distributed real-time data analytics system; improve their throughput by 1.3–3.6× and by 75–96%, respectively, over a CPU-only implementation.

1 Introduction

Network bandwidth is growing much faster than CPU performance [5], forcing many data-center applications to sacrifice application cycles for packet processing [9, 23, 37]. As a result, system developers have started to offload computation to programmable network interface controllers (NICs), dramatically improving the performance and energy efficiency of many data-center applications, such as search engines, key-value stores, real-time data analytics, and intrusion detection [12, 23, 26, 40]. These NICs have a variety of hardware architectures including FPGAs [12, 33, 48], specialized flow engines [6], and more general-purpose network processors [3, 32].

However, implementing data-center network applications in a combined CPU-NIC environment is difficult. It often requires many design-implement-test iterations before the accelerated application can outperform its CPU-only version. These iterations involve non-trivial changes: programmers may have to move portions of application code across the CPU-NIC boundary and manually refactor the program.

We propose FLOEM, a programming system for NIC-accelerated applications. Our current prototype targets a platform with the Cavium LiquidIO [3], a general-purpose programmable NIC that executes C code. FLOEM is based on a data-flow language that is natural for expressing packet processing logic and mapping *elements* (modular program components) onto hardware devices. The language lets developers easily move an element onto a CPU or a NIC to explore alternative offloading designs, as well as parallelize program components. Application developers can define a FLOEM element as a Python class that contains a C implementation of the element. To aid programming productivity, we provide a library of common elements.

Further examining how developers offload data-center applications to NICs, we have identified the following commonly encountered problems, which led us to propose abstractions and mechanisms amenable to a data-flow programming model that can solve these problems.

- Different offloading choices require different communication strategies. We observe that these strategies can be expressed by a **mapping of logical communication queues to physical queues**, so we propose this mapping as a part of our language.

- Moving computation across the CPU-NIC boundary may change which parts of a packet must be sent across the boundary. Marshaling the necessary packet fields

is tedious and error-prone. Thus, we propose **per-packet state** — an abstraction that allows a packet and its metadata to be accessed anywhere in the program — while FLOEM automatically transfers only required packet parts between a NIC and CPU.

- Using an in-network processor to cache application state or computation is a common pattern for accelerating data-center applications. However, it is non-trivial to implement a cache that guarantees the consistency of data between a CPU and NIC. We propose a **caching construct** for memoizing a program region, relieving programmers from having to implement a complete cache protocol.
- Developers often want to **offload an existing application** without rewriting the code into a new language. We let programmers embed C code in elements and allow a legacy application to interact with FLOEM elements via a simple function call, executing those elements in the host process of the legacy application.

We demonstrate that without significant programming effort, FLOEM can help offload parts of real-world applications — a key-value store and a real-time analytics system — improving their throughput by 1.3–3.6× and 75–96%, respectively, over a CPU-only configuration.

In summary, this paper makes the following contributions:

- Identifying *challenges* in designing of NIC-accelerated data-center applications (Section 2)
- Introducing *programming abstractions* to address these challenges (Sections 3 and 4)
- Developing a programming system that enables exploration of alternative offloading designs, including a *compiler* (Section 5) and a *runtime* (Section 6) for efficient data transfer between a CPU and NIC

2 Design Goals and Rationale

We design FLOEM to help programmers explore how to offload their server network applications to a NIC. The applications that benefit from FLOEM have computations that *may* be more efficient to run on the NIC than on the CPU because of the NIC's hardware-accelerated functions, parallelism, or reduced latency when eliminating the CPU from fast-path processing. These computations include packet filtering (e.g., format validation and classification), packet transformation (e.g., serialization, compression, and encryption), packet steering (e.g., load balancing to CPU cores), packet generation, and caching of application state. This list is not exhaustive. Ultimately, we would like FLOEM to help developers discover new ways to accelerate their applications.

The main challenge when designing programming abstractions is to realize a small number of constructs that let programmers express a large variety of implementation choices. This requires an understanding of common challenges within the application domain. We build FLOEM to meet the following design goals.

Goal 1: Expressing Packet Processing

As described above, computations suitable for NIC offloading are largely packet processing. Programming abstractions and systems for packet processing have long been studied, and the Click modular router [34] is widely used for this task. We adopt its data-flow model to ease the development of packet processing logic (Section 3).

Goal 2: Exploring Offload Designs

A data-flow model is suitable for mapping computations to desired hardware devices, as we have seen with many Click extensions that support offloading [24, 27, 46]. Similarly, FLOEM programmers implement functionality once, as a data-flow program, after which they can use code annotations to assign elements to desired devices and to parallelize the program. However, trivially adopting a data-flow model is insufficient to meet this design goal. By inspecting the design of a key-value store and a TCP stack offloaded with FlexNIC [23], we discover several challenges that shape the design of our language.

Logical-to-physical queue mapping (Section 4.1). One major part of designing an offloading strategy is managing the transfer of data between the host and accelerator. Various offloading strategies require different communication strategies, such as how to steer packets, how to share communication resources among different types of messages, and whether to impose an order of messages over a communication channel.

By examining hand-optimized offloads, we find that developers typically express communication in terms of logical queues and then manually implement them using the provided hardware communication mechanisms. A logical queue handles messages sent from one element to another, while a hardware communication channel implements one physical queue. As part of an offload implementation, developers have to make various mapping choices among logical and physical queues. The right mapping depends on the workload and hardware configuration and is typically realized via trial-and-error.

To aid this task, we design a queue construct with an explicit logical-to-physical queue mapping that can be controlled via parameters and by changing element connections. Existing frameworks [24, 27, 46] do not support this mapping. To control the number of physical

queues in these frameworks, programmers have to explicitly: (1) create more logical queues by demultiplexing the flow into multiple branches and making more elements and connections, or (2) merge logical queues by multiplexing multiple branches into one.

Per-packet state (Section 4.2). In a well-optimized program, developers meticulously construct a message by copying only the necessary parts of a packet to send between a CPU and NIC; this minimizes the amount of data transferred over PCIe. When developers move computation between the CPU and NIC, they may need to rethink which fields must be sent, slowing the exploration of alternative offloading designs.

Nevertheless, no existing system performs this optimization automatically. ClickNP [27] sends an entire packet, while NBA [24] and Snap [46] rely on developers to annotate each element with a packet's *region of interest*, specified as numeric offsets in a packet buffer. We design FLOEM to automatically infer what data to send across the CPU-NIC boundary and offer the *per-packet state* abstraction as if an entire packet could be accessed anywhere in the program. This abstraction resembles P4's per-packet metadata [10] and RPC IDLs (e.g., XDR [14] and Google's protobuf [18]). However, P4 allows per-packet metadata to be carried across multiple processing pipelines only within a single device, while RPC IDLs generate marshaling code based on interface descriptions, rather than automatically inferring.

Caching construct (Section 4.3). Caching application state or memoizing computation in an in-network processor is a common strategy to accelerate server applications [15, 22, 26, 30]. While the abstractions we have so far are sufficient to express this strategy, implementing a cache protocol still requires a significant effort to guarantee both data consistency and high performance when messages between a CPU and NIC may arrive out-of-order. Thus, we introduce a *caching construct*, a general abstraction for caching that integrates well with the data-flow model. This construct provides a full cache protocol that maintains data consistency between the CPU and NIC. Unlike FLOEM, existing systems support caching only of flow state [6, 27] — which typically does not require maintaining consistency between the CPU and NIC — but not caching of application state.

Goal 3: Integrating with Existing Applications

Prior frameworks were designed exclusively to implement network functions and packet processing [13, 16, 24, 27, 34, 36, 46], where computation is mostly stateless and simpler than in our target domain of server applications. While parts of typical server applications can be built by composing pre-defined elements, many parts cannot. In our target domain, developers often want to offload an application by reusing existing application code instead of writing code from scratch. Besides porting existing applications, some developers may prefer to implement most of their applications in C because a data-flow programming model may not be ideal for the full implementation of complex applications.

FLOEM lets developers combine custom and stock elements, embed C code in data-flow elements, and integrate a FLOEM program with an external program. As a result, developers can port only program parts that may benefit from offloading into the data-flow model. The impedance mismatch between the data-flow model and the external program's model (e.g., event-driven or imperative) raises the issue of interoperability. Our solution builds on the queue construct to decouple the internal part from the interface part, which appears to the external program as a function (**Section 4.4**). The external program can execute the function using its own thread to (1) retrieve a message from the queue and process it through elements in the interface part, or (2) process a message through the interface part and push it to the queue.

3 Core Abstractions

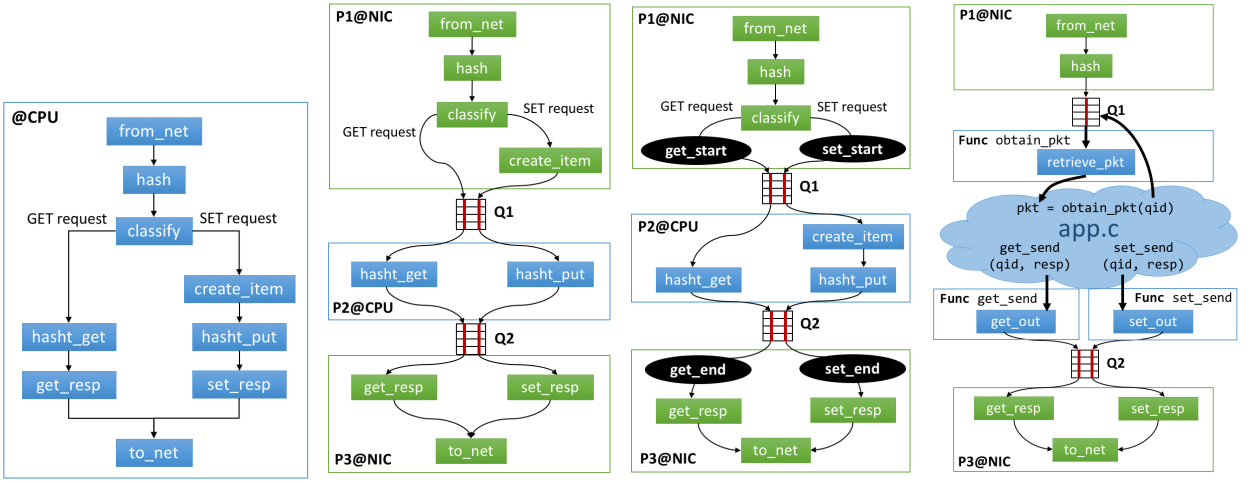
We use a key-value store application as our running example. Figure 1 displays several offloading designs for the applicaton: CPU-only (Figure 1a), split CPU-NIC (Figure 1b), and NIC as cache (Figure 1c). Figure 1d illustrates how to create an interface that an external program can use to interact with FLOEM. We show how to implement these offloads using our programming abstractions in this and the next sections.

Elements. FLOEM programs are composed of elements. Upon receiving inputs from all its input ports, an element processes the inputs and emits outputs to its output ports. The listing below illustrates how to create the `classify` element in our key-value store example, which classifies incoming requests by type (GET or SET).

```
class Classify(Element): # Define an element class
  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
```



(a) No offloading

(b) CPU-NIC split (see Section 4.1)

(c) NIC as cache (see Section 4.3)

(d) Interface to external program (see Section 4.4)

Figure 1: Several offloading strategies of a key-value store implemented in FLOEM

We specify input and output ports in the `configure` method. We express the logic for processing a single packet in the `impl` method by calling `run_c`, which accepts C code with special syntax to retrieve value(s) from an input port and emit value(s) to an output port.

To create the program shown in Figure 1a, we connect elements as follows:

```
from_net >> hash >> classify
classify.get >> hasht_get >> get_resp >> to_net
classify.set >> item >> hasht_put >> set_resp >> to_net
```

Note that `.get` and `.set` refer to the output ports of `classify`.

Queues. Instead of pushing data to the next element instantaneously, a queue can store data until the next element dequeues it. A queue can connect and send data between elements on both different devices (e.g., CPU and NIC) and on the same device.

Shared state. FLOEM provides a shared state abstraction that lets multiple elements share a set of variables that are persistent across packets. For example, elements `hasht_get` and `hasht_put` share the same state containing a hash table. FLOEM normally prohibits elements on different devices from sharing the same state. Instead, programmers must use message passing across queues to share information between those elements. Shared state lets programmers express complex stateful applications.

Segmented execution model. A *segment* is a set of connected elements that begins with from a *source* element, which is either a `from_net` element or a queue, and ends with *leaf* elements (elements with no output ports) or queues. A queue sends packets between segments. Our execution model is run-to-completion within a segment. A source element processes a packet and pushes it to subsequent elements until the packet reaches the end of the segment. When the entire segment finishes processing a packet, it starts on the next one. By default, one thread on a CPU executes each segment, so elements within a segment run sequentially with respect to their data-flow dependencies.

The program in Figure 1a has a single segment, while the program in Figure 1b has three. Note that not all elements in a segment must be executed for each packet. In our example, either `hasht_get` or `hasht_put` (not both) will be executed depending on the port where `classify` pushes a packet to.

Offloading and parallelizing. A segment is a unit of code migration and parallelization. Programmers map each segment to a specific device by supplying the `device` parameter. They can also assign multiple threads to run the same segment to process different packets in parallel using the `cores` parameter. Programmers cannot assign a segment to run on both the NIC and CPU in parallel; the current workaround is to create two identical segments, one for NIC and another for CPU. Figure 2 displays a FLOEM program that implements a sharded key-value store with the offloading strategy in Figure 1b.

4 Advanced Offload Abstractions

This section presents programming abstractions that we propose to mitigate recurring programming challenges encounters when exploring different ways to offload applications to a NIC.

```
1  Q1 = Queue(channel=2, inst=3)
2  Q2 = Queue(channel=2, inst=3)
3
4  class P1(Segment):
5    def impl(self):
6      from_net >> hash >> queue_id >> classify
7      classify.get >> Q1.enq[0]  # channel 0
8      classify.set >> create_item >> Q1.enq[1] # chnl 1
9
10 class P2(Segment):
11   def impl(self):
12     self.core_id >> Q1.qid  # use core id as queue id
13     Q1.deq[0] >> hasht_get >> Q2.enq[0]
14     Q1.deq[1] >> hasht_put >> Q2.enq[1]
15
16 class P3(Segment):
17   def impl(self):
18     scheduler >> Q2.qid # scheduler produces queue id
19     Q2.deq[0] >> get_resp >> to_net
20     Q2.deq[1] >> set_resp >> to_net
21
22 P1(device=NIC, cores=[0,1]) # run on core id 0,1
23 P2(device=CPU, cores=[0,1,2])
24 P3(device=NIC, cores=[2,3])
```

Figure 2: FLOEM program implementing a sharded key-value store with the CPU-NIC split strategy of Figure 1b

4.1 Logical-to-Physical Queue Mapping

To achieve correctness and maximize performance, FLOEM gives programmers control over how the compiler instantiates logical queues for a particular offloading strategy. The queue construct `Queue(channel=n, inst=m)` represents n logical queues (n channels) using m physical queues (m instances). For example, `Q1` on line 1 of Figure 2 represents two logical queues — displayed as red channels in Figure 1b — using three physical queues. Different mappings of logical to physical queues lead to different communication strategies, as elaborated below.

Packet steering. Developers can easily implement packet steering by creating a queue with multiple physical instances. For example, in the split CPU-NIC version of the key-value store (Figure 1b), we want to shard the key-value store so that different CPU threads can handle different subsets of keys to avoid lock contention and CPU cache misses. As a result, we want to represent queue `Q1` by multiple physical queues, with each CPU thread having a dedicated physical queue to handle requests for its shard. The NIC then steers a packet to the correct physical queue based on its key. FlexNIC [23] shows that such key-based steering improves throughput of the key-value store application by 30–45%.

To implement this strategy, we create `Q1` with multiple physical queues (line 1 in Figure 2). Steering a packet is controlled by assigning the target queue instance ID to the `qid` field of *per-packet state* in the C code of any element that precedes the queue. In this example, we set `state.qid = hash(pkt.key) % 3`, where `state` refers to *per-packet state*.

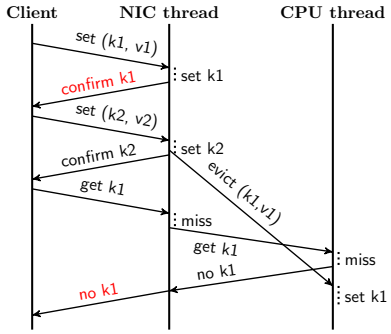


Figure 3: Inconsistency of a write-back cache if messages from NIC to CPU are reordered

Resource sharing. Developers may want to map multiple logical queues to the same physical queue for resource sharing, or vice versa for resource isolation. For example, they may want to consolidate infrequently used logical queues into one physical queue to obtain a larger batch of messages per PCIe transfer. In the sharded key-value store, we want to use the same physical queue to transport both the GET and SET requests of one shard so that the receiver's side processes these requests at the same rate as the sender's. To implement this, we use `Q1` to represent two logical queues (line 1 in Figure 2): one for GET and one for SET. Different degrees of sharing can vary application performance by up to 16% (Section 7.2).

Packet ordering. For correctness, developers may want to preserve the order of packets being processed from one device to another. For example, an alternative way to offload the key-value store is to use the NIC as a key-value cache, only forwarding misses to the CPU. To ensure consistency of the write-back cache, we must enforce that the CPU handles evictions and misses of the same key in the same order as the cache. Figure 3 shows an inconsistent outcome when an eviction and a miss are reordered. To avoid this problem, developers can map logical queues for evictions and misses to the same physical queue, ensuring in-order delivery.

The ability to freely map logical to physical queues lets programmers express different communication strategies with minimal effort in a declarative fashion. A queue can also be parameterized by whether its enqueuing process is lossless or lossy, where a lossless queue is blocking. Note that programmers are responsible for correctly handling multiple blocking queues.

4.2 Per-Packet State

FLOEM provides per-packet state, an abstraction that allows access to a packet and its metadata from any element without explicitly passing the state. To use this

abstraction, programmers define its format and refer to it using the keyword `state`. For our key-value store, we define the format of the per-packet state as follows:

```
class MyState(State): # define fields in a state
 hash = Field(uint32_t)
 pkt  = Field(pointer(kvs_message))
 key  = Field(pointer(void), size='state.pkt->keylen')
```

The provided element `from_net` creates a per-packet state and stores a packet pointer to `state.pkt` so that subsequent elements can access the packet fields, such as `state.pkt->keylen`. The element `hash` computes the hash value of a packet's key and stores it in `state.hash`, which is used later by element `hasht_get`. To handle a variable-size field, FLOEM requires programmers to specify its size, as with the key field above.

4.3 Caching Construct

With only minimal changes to a program, FLOEM offers developers a high-level caching construct for exploring caching on the NIC and storing outputs of expensive computation to be used in the future. First, programmers instantiate the caching construct `Cache` to create an instance of a cache storage and elements `get_start`, `get_end`, `set_start`, and `set_end`. Programmers then insert `get_start` right before the get query begins, and `get_end` right after the get query ends; a get query is computation we want to memoize. Programmers must also specify what to store as a key (input) and a value (output) in the cache; this can be done by assigning `state.key` and `state.keylen` (key and keylen fields of per-packet state) before the element `get_start`, and assigning `state.val` and `state.vallen` before `get_end`. If the application has a corresponding set query, elements `set_start` and `set_end` must be inserted, and those fields of the per-packet state must be assigned accordingly for the set query; a set query mutates application state and must be executed when a cache eviction occurs. Finally, programmers can use parameters to configure the cache with the desired table size, cache policy (either write-through or write-back), and a write-miss policy (either write-allocate or no-write-allocate).

For our key-value store example, we can use the NIC to cache outputs from hash table get operations by just inserting the caching elements, as shown in Figure 1c. Notice that queues `Q1` and `Q2` are parts of the expensive queries (between `get_start` and `get_end` and between `set_start` and `set_end`) that can be avoided if outputs are in the cache.

Requirements. The get and set query regions cannot contain any *callable segment* (see Section 4.4). Elements `get_start`, `get_end`, `set_start`, and `set_end` must be on the same device. Paths between `get_start` and `get_end`, and between `set_start` and `set_end`, must pass through the same set of queues (e.g., Figure 1c) to ensure the in-order delivery of misses and evictions of the same key. Multiple caches can be used as long as cached regions are not overlapped. The compiler returns an error if a program violates these requirements.

4.4 Interfacing with External Code

To help developers offload parts of existing programs to run on a NIC, we let them: (1) embed C code in elements, (2) implement elements that call external C functions available in linkable object files, and (3) expose segments of FLOEM elements as functions callable from any C program. The first mechanism is the standard way to implement an element. The second simply links FLOEM-generated C code with object files. For the last mechanism, we introduce a *callable segment*, which contains elements between a queue and an endpoint, or vice versa. An endpoint element may send/receive a value to/from an external program through its output/input port. A callable segment is exposed as a function that can be called by an external program to execute the elements in a segment.

In Figure 1d, we implement simple computation, such as hashing and response packet construction, in FLOEM, but we leave complex functionality, including the hash table and item allocation, in an external C program. The external program interacts with the FLOEM program to retrieve a packet, send a get response, and send a set response via function `obtain_pkt`, `get_send`, and `set_send`, respectively. The following listing defines the function `obtain_pkt` using a callable segment. This function takes a physical queue ID as input, pulls the next entry from the queue with the given ID, executes element `retrieve_pkt` on the entry, and returns the output from `retrieve_pkt` as the function's return value.

```
class ObtainPkt(CallableSegment):
  def configure(self):
    self.inp = Input(int)      # argument is int
    self.out = Output(q_entry) # return value is q_entry

  def impl(self):
    self.inp >> Q1.qid
    Q1.deq >> retrieve_pkt >> self.out

ObtainPkt(name='obtain_pkt')
```

The external program running on the CPU calls `obtain_pkt` to retrieve a packet that has been processed by element `hash` on the NIC and pushed into queue `Q1`.

5 The FLOEM Compiler

The FLOEM compiler contains three primary components that: (1) translate a data-flow program with elements into C programs, (2) infer minimal data transfers across queues, and (3) expand the high-level caching construct into primitive elements, as depicted in Figure 4.

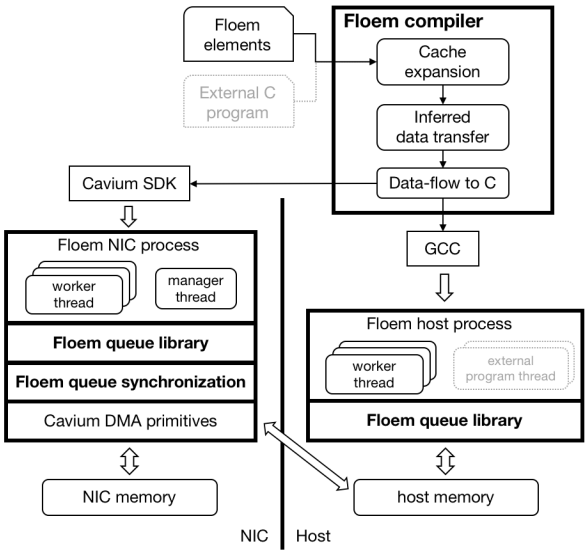
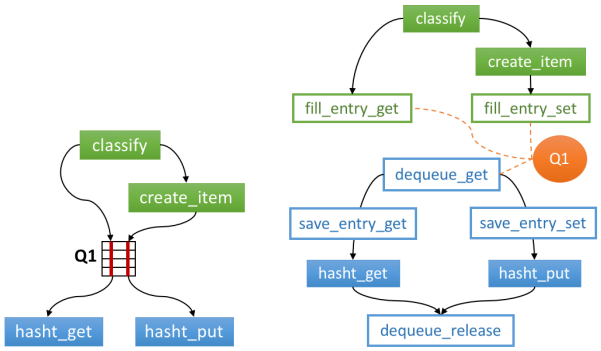


Figure 4: FLOEM system architecture



(a) Before transformation (b) After transformation

Figure 5: The key-value store's data-flow subgraph in the proximity of queue `Q1` from the split CPU-NIC version

5.1 Data-Flow to C

FLOEM compiles a data-flow program into two executable C programs: one running on the CPU and the other on the NIC. Our code generator compiles a segment of primitive elements into a chain of function calls, where one element corresponds to a function. The compiler replaces an output port invocation with a function call to the next element connected to that output port. The calling element passes an output value to the next element as an argument to the function call. Earlier compiler passes transform queues (Section 5.2) and caching constructs (Section 5.3) into primitive elements.

5.2 Inferred Data Transfer

In this section, we explain how the FLOEM compiler infers which fields of a packet and its metadata must be sent across each queue, and how it transforms queues into a set of primitive elements.

Liveness analysis. The compiler infers per-packet state's fields to send across each logical queue (each queue's channel) using a classical liveness analysis [7]. The analysis collects used and defined fields at each element and propagates information backward to compute a *live* set at each element (i.e., a set of fields that are used by the element's successors). For each segment, the compiler also collects a *use* set of all fields that are accessed in the segment.

Transformation. After completing the liveness analysis, the compiler transforms each queue construct into multiple primitive elements that implement enqueue and dequeue operations. In the split CPU-NIC version of the key-value store example, the compiler transforms queue `Q1` in Figure 5a into the elements in Figure 5b.

To enqueue an entry to a logical queue at a channel `X`, we first create element `fill_entry_X` to reserve a space in a physical queue specified by `state.qid`. We then copy the *live* per-packet state's fields at channel `X` into the queue. To dequeue an entry, element `dequeue_get` locates the next entry in a specified physical queue, classifies which channel the entry belongs to, and passes the entry to the corresponding output port (i.e., demultiplexing). Element `save_entry_X` allocates memory for the per-packet state on the receiver's side to store the *use* fields and a pointer to the queue entry so that the fields in the entry can be accessed later. Each `save_entry_X` is connected to the element that was originally connected to that particular queue channel. Finally, the compiler inserts a `dequeue_release` element to release the queue entry after its last use in the segment. These generated elements utilize the built-in queue implementations described in Section 6.

5.3 Cache Expansion

The compiler expands each high-level caching construct into primitive elements that implement a cache policy using the expansion rules shown in Figure 6. Each node in the figure corresponds to a subgraph of one or more elements. For a write-through cache without allocation on write misses, the compiler expands the program graphs that handle get and set queries in the left column into the graphs in the middle column. For a write-back policy with allocation on write misses, the resulting graphs are shown in the right column. For get-only applications, we skip the set expansion rule.

We apply various optimizations to reduce response time. For example, when a new allocation causes an

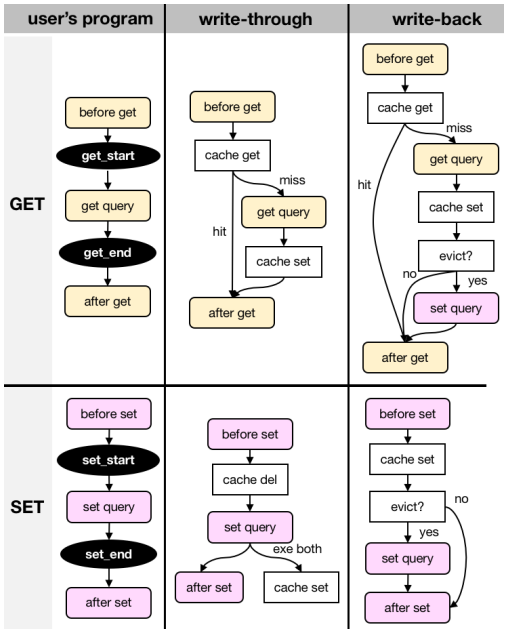


Figure 6: Cache expansion rules

eviction in a write-back cache, we write back the evicted key asynchronously. Instead of waiting for the entire `set query` to finish before executing `after get` (e.g., sending the response), we wait only until the local part of `set query` (on a NIC) reaches a queue to the remote part of `set query` (on a CPU). Once we successfully enqueue the eviction, we immediately execute `after get`.

5.4 Supported Targets

We prototype FLOEM on a platform with a Cavium LiquidIO NIC [3]. We use GCC and Cavium SDK [2] to compile C programs generated by FLOEM to run on a CPU in user mode and on a NIC, respectively. If a FLOEM program contains an interface to an external C program, the compiler generates a C object file that the external application can link to in order to call the interface functions.

Intrinsics, libraries, and system APIs of the two hardware targets differ. To handle these differences, FLOEM lets programmers supply different implementations of a single element class to target x86 and Cavium via `impl` and `impl_cavium` methods, respectively. If `impl_cavium` is not implemented, the compiler refers to `impl` to generate code for both targets. To generate programs with parallelism, FLOEM uses *pthread* on the CPU for multiple segments and relies on the OS thread scheduler. On the NIC, we directly use hardware threads and assign each segment to a dedicated NIC core. Consequently, the compiler prohibits creating more segments on the NIC than the maximum number of cores (12 for LiquidIO).

6 PCIe I/O Communication

To efficiently communicate between the NIC and CPU over PCIe, FLOEM provides high-performance, built-in queue implementations, which rely on the queue synchronization layer (sync layer) to efficiently synchronize data between NIC and CPU. Figure 4 depicts how these components interact with the rest of the system. Currently, we support only a one-way queue with fixed-size entries, parameterized during compile-time.

6.1 Queue Synchronization Layer

Because DMA engines on the NIC are underpowered, they must be managed carefully. If we implemented the queue logic together with data synchronization, the queue implementation would be extremely complicated and difficult to troubleshoot. Hence, we decouple these layers. The sync layer can then additionally be used for other queue implementations, such as a queue with variable-size entries.

Our sync layer provides the illusion that the NIC writes directly to a circular buffer in host memory, where one buffer represents one physical queue. The layer keeps shadow copies of queues in local NIC memory, asynchronously synchronizes these copies with master copies in host memory, batches multiple DMA requests, and overlaps DMA operations with other computation.

To use this layer, a queue implementation must: (1) maintain a status flag in each entry to indicate its availability, and (2) provide basic queue information and queue entry's status checking functions. In turn, the sync layer provides `access_entry` and `access_done` functions to the queue implementation; the queue implementation must call `access_entry` and `access_done` before and after accessing/modifying any queue entry, respectively.

6.2 Maintaining Coherent Buffers

The queue synchronization layer relies on FLOEM's NIC runtime to maintain coherence between buffers on the NIC and the CPU by taking advantage of the circular access pattern of reads followed by writes. We do not explicitly track a queue's head and tail; instead, we use a status flag in each entry to determine if an entry is filled or empty. We choose this design to synchronize both the queue entry's content and status using one DMA operation instead of two. Thus, the runtime continuously checks the state of every queue entry and performs actions accordingly.

Typically, a queue entry on the NIC cycles through *invalid*, *reading*, *valid*, *modified*, and *writing* states, as shown in Figure 7. An *invalid* entry contains stale content and must be fetched from host memory. An asyn-

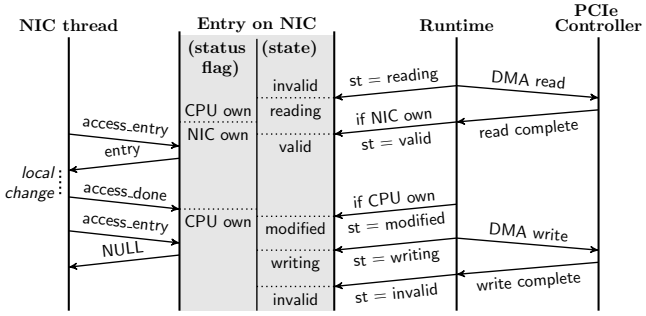


Figure 7: Transitions of a queue entry's status by a NIC worker thread and a NIC runtime manager thread

chronous DMA read transitions an entry from *invalid* to *reading* state. Once the read completes, and the entry is NIC owned (indicated by the status flag), the entry transitions to *valid* state. It may transition back to *invalid* if it is still CPU owned, for example, when the NIC attempts to dequeue an entry that the CPU has not finished enqueuing. The runtime uses the status checking functions provided by the queue implementation to check an entry's status flag. The program running on the NIC can access only *valid* entries; function `access_entry` returns the pointer to an entry if it is in *valid* state; otherwise, it returns NULL.

An entry transitions from *valid* to *modified* once the queue implementation calls function `access_done` to indicate that it is finished accessing that entry. An asynchronous DMA write then transitions the entry to *invalid* state, based on the assumption that the CPU side will eventually modify it, and the NIC must read it from the CPU. This completes a typical cycle of states through which an entry passes.

Note that the CPU side does not need this sync layer or track these states because, unlike the NIC, it does not issue DMA operations.

6.3 I/O Batching

In the actual implementation, we do not track the state of individual queue entries due to high overhead. Instead, we use five pointers to divide a circular queue buffer into five portions with the five states. When a pointer advances, we effectively change the states of a batch of entries that the pointer has moved past. The runtime has a dedicated routine to advance each pointer, and executes these routines in round-robin fashion, overlapping DMA read/write routines with other routines. To achieve DMA batching, the DMA read routine issues a DMA read for the next batch of entries instead of a single entry, as does the DMA write routine. We use a configurable number of dedicated NIC cores (manager threads) to execute the runtime. Each core manages a disjoint subset of queues.

More details about our queue implementation and queue synchronization layer beyond this section can be found in Section 3.6 of the first author's thesis [38].

7 Evaluation

We ran experiments on two small-scale clusters to evaluate the benefit of offloading on servers with different generations of CPUs: 6-core Intel X5650 in our *Westmere* cluster, and 12-core Intel E5-2680 v3 in our *Sandy Bridge* cluster (more powerful). Each cluster had four servers; two were equipped with Cavium LiquidIO NICs, and the others had Intel X710 NICs. All NICs had two 10Gbps ports.

We evaluated CPU-only implementations on the servers with the Intel X710 NICs, using DPDK [4] to send and receive packets bypassing the OS networking stack to minimize overheads. We used the servers with the Cavium LiquidIO NICs to evaluate implementations with NIC offloading. The Cavium LiquidIO has a 12-core 1.20GHz cnMIPS64 processor, a set of on-chip/off-chip accelerators (e.g., encryption/decryption engines), and 4GB of on-board memory.

7.1 Programming Abstraction

We implemented in FLOEM two complex applications (key-value store and real-time data analytics) and three less complex network functions (encryption, flow classification, and network sequencer).

Hypothesis 1 FLOEM *lets programmers easily explore offload strategies to improve application performance.*

The main purpose of this experiment is to demonstrate that FLOEM makes it easier to explore alternative offloading designs, *not* to show when or how one should or should not offload an application to a NIC.

For the complex applications, we started with a CPU-only solution as a baseline by porting parts of an existing C implementation into FLOEM. Then, we used FLOEM to obtain a simple partition of the application between the CPU and NIC for the first offload design. In both case studies, we found that the first offloading attempt was unsuccessful because an application's actual performance can greatly differ from a conceptual estimate. However, we used FLOEM to redesign the offload strategy to obtain a more intelligent and higher performing solution, with minimal code changes, and achieved 1.3–3.6× higher throughput than the CPU-only version.

For the less complex workloads, FLOEM let us quickly determine whether we should dedicate a CPU core to handle the workload or just use the NIC and save CPU cycles for other applications. By merely changing FLOEM's device mapping parameter, we found that

it was reasonable to offload encryption and flow classification to the NIC, but that the network sequencer should be run on the CPU. The rest of this section describes the applications in our experiment in greater detail.

Case Study: Key-Value Store

In this case study, we used one server to run the key-value store and another to run a client generating workload, communicating via UDP. The workload consisted of 100,000 key-value pairs of 32-byte keys and 64-byte values, with the Zipf distribution ($s = 0.9$) of 90% GET requests and 10% SET requests, the same workload used in FlexNIC [23]. We used a single CPU core with a NIC offload (potentially with multiple NIC cores); this setup was reasonable since other CPU cores may be used to execute other applications simultaneously. Figure 8 shows the measured throughput of different offloading strategies, and Table 1 summarizes the implementation effort.

CPU-only (Figure 1a): We ported an existing C implementation, which runs on a CPU using DPDK, into FLOEM except for the garbage collector of freed key-value items. This effort involved converting the original control-flow logic into the data-flow logic, replacing 538 lines of code with 334 lines. The code reduction came from using reusable elements (e.g., `from_net` and `to_net`), so we did not have to set up DPDK manually.

***Split CPU-NIC (Figure 1b)*:** We tried a simple CPU-NIC partition, following the offloading design of FlexKVS [23], by modifying 296 lines of the CPU-only version; this offload strategy was carefully designed to minimize computational cycles on a CPU. It required many changes because the NIC (`create_item` element) creates key-value items that reside in CPU memory. Unexpectedly, this offload strategy lowered performance (the second bar). Profiling the application revealed a major bottleneck in the element that prepares a GET response on the NIC. The element issued a blocking DMA read to retrieve the item's content from host memory. This DMA read was not part of queue `Q2` because that queue sent only the pointer to the item, not the item itself. Therefore, the runtime could not manage this DMA read; as a result, this strategy suffered from this additional DMA cost.

***NIC caching (Figure 1c)*:** We then used FLOEM to explore a completely different offload design. Since the Cavium NIC has a large amount of local memory, we could cache a signification portion of the key-value store on the NIC. This offload design, previously explored, was shown to have high performance [26]. Therefore, we modified the CPU-only version by inserting the caching construct (43 lines of code) as well as creating segments and inserting queues (62 lines of code). For a baseline comparison, code relevant to communication on the CPU side alone was already at 240 lines in a manually-written C implementation of FlexKVS with a software NIC emulation. This translated to fewer than 15 lines of code in FLOEM. These numbers show that implementing a NIC-offload application without FLOEM requires significantly more effort than with FLOEM.

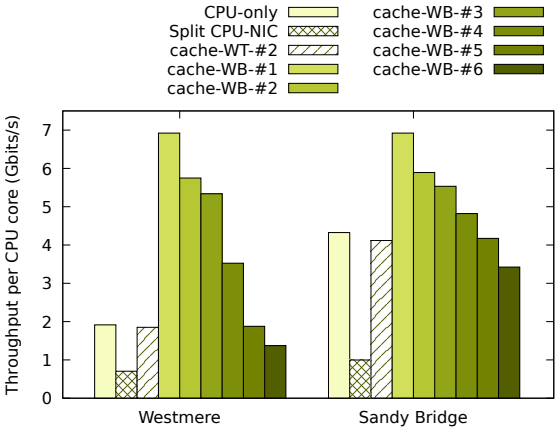


Figure 8: Throughput per CPU core of different implementations of the key-value store. WB = write-back, WT = write-through. #N in "cache-WB-#N" is the configuration number. Table 2 shows the cache sizes of the different configurations and their resulting hit rates.

Version (obtained from)	Effort (loc)	Details
Existing	1708	Hand-written C program
CPU-only (Existing)	replace 538 with 334	Refactor C program into FLOEM elements.
Split CPU-NIC (CPU-only)	add 296	Create queues. NIC remotely allocates items on CPU memory.
Caching (CPU-only)	add 43	Create a cache. Assign key, keylen, val, vallen.
NIC caching (Caching)	add 62	Create queues and segments.

Table 1: Effort to implement key-value store. The last column describes specific modification details other than creating, modifying, and rewiring elements. As a baseline, code relevant to communication on the CPU side alone was 240 lines in a manual C implementation.

Regarding performance, the third bar in Figure 8 reports the throughput when using a write-through cache with 2^{15} buckets and five entries per bucket, resulting in a 90.3% hit rate. According to the result, the write-through cache did not provide any benefit over the CPU-only design, even when the cache hit rate was quite high. Therefore, we configured the caching construct to use a write-back policy (by changing the cache policy parameter) because write-back generally yields higher throughput than write-through. The remaining bars show the performance when using a write-back cache with different cache sizes, resulting in the different hit rates shown

Config.	#1	#2	#3	#4	#5	#6	#2 (WT)
# of buckets	2^{15}	2^{15}	2^{15}	2^{15}	2^{14}	2^{14}	2^{15}
# of entries	∞	5	2	1	1	1	5
hit rate (%)	100	97.2	88.4	75.3	65.0	55.2	90.3

Table 2: The sizes of the cache (# of buckets and # of entries per bucket) on the NIC and the resulting cache hit rates when using the cache for the key-value store. All columns report the hit rates when using write-back policy except the last column for write-through. ∞ entries mean a linked list.

in Table 2. This offloading strategy improved throughput over the CPU-only design by 2.8–3.6× on Westmere and 28–60% on Sandy Bridge when the hit rate exceeded 88% (configuration #1–3).

Notice that at high cache hit rates, the throughput for this offload strategy was almost identical on Westmere and Sandy Bridge regardless of the CPU technology. The NIC essentially boosted performance on the Westmere server to be on par with the Sandy Bridge one. In other words, an effective NIC offload reduced the workload's dependency on CPU processing speed.

Case Study: Distributed Real-Time Data Analytics

Distributed real-time analytics is a widely-used application for analyzing frequently changing datasets. Apache Storm [1], a popular framework built for this task, employs multiple types of workers. Spout workers emit tuples from a data source; other workers consume tuples and may emit more tuples. A worker thread executes one worker. De-multiplexing threads route incoming tuples from the network to local workers. Multiplexing threads route tuples from local workers to other servers and perform simple flow control. Our specific workload ranked the top *n* users from a stream of Twitter tweets. In this case study, we optimized for throughput per CPU core. Figure 9 and Table 3 summarize the throughput and implementation effort of different strategies, respectively.

CPU-only: We ported demultiplexing, multiplexing, and DCCP flow-control from FlexStorm [23] into FLOEM but kept the original implementation of the workers as an external program. We used *callable segments* (Section 4.4) to define functions `inqueue_get` and `outqueue_put` for workers (in the external program) to obtain a task from the demultiplexer and send a task to the multiplexer (in FLOEM). This porting effort involved replacing 1,192 lines of code with only 350 lines. The code reduction here was much higher than in the key-value store application because FlexStorm's original implementation required many communication queues, which were replaced by FLOEM queues. The best CPU-only configuration that achieved the highest throughput per core used three cores for three workers (one spout, one counter, and one ranker), one core for demultiplexing, and two cores for multiplexing.

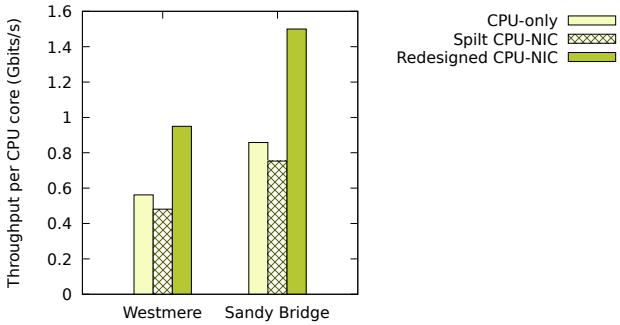


Figure 9: Throughput per CPU core of different Storm implementations

Version (obtained from)	Effort (loc)	Details
Existing	2935	Hand-written C program
CPU-only (Existing)	replace 1192 with 350	Refactor C program into FLOEMelements.
Split CPU-NIC (CPU-only)	modify 1	Change `device` parameter.
Redesigned (Split CPU-NIC)	add 23	Create bypass queues.

Table 3: Effort to implement Storm. The last column describes specific modification details other than creating, modifying, and rewiring elements.

Split CPU-NIC: As suggested in FlexNIC, we offloaded (de-)multiplexing and flow control to the NIC, by changing the device parameter (one line of code change). This version, however, lowered throughput slightly compared to the CPU-only version.

Redesigned CPU-NIC: The split CPU-NIC version can be optimized further. A worker can send its output tuple to another local worker or a remote worker over the network. For the former case, a worker sends a tuple to the multiplexer on the NIC, which in turn forwards it to the target worker on the CPU. Notice that this CPU-NIC-CPU round-trip is unnecessary. To eliminate this communication, we created bypass queues for workers to send tuples to other local workers without involving the multiplexer. With this slight modification (23 lines of code), we achieved 96% and 75% higher throughput than the CPU-only design on the Westmere and Sandy Bridge cluster, respectively.

Other Applications

The following three applications are common network function tasks. Because of their simplicity, we did not attempt to partition them across the CPU and NIC. Figure 10 reports throughput when using one CPU core on a Sandy Bridge server or offloading everything to the Cav-

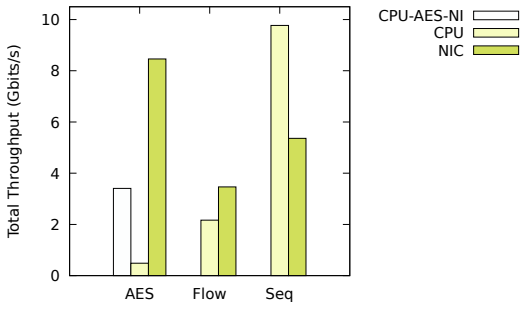


Figure 10: Throughput of AES encryption, flow classification, and network sequencer running on one CPU core and the LiquidIO NIC. 'CPU-AES-NI' uses AES-NI.

ium NIC. In our experiment, we used a packet size of 1024 bytes for encryption and network sequencer, and 80 bytes for flow classification.

Encryption is a compute-intensive stateless task, used for Internet Protocol Security. In particular, we implemented AES-CBC-128. We wrote two CPU versions: (1) using Intel Advanced Encryption Standard New Instructions (AES-NI), and (2) without AES-NI, which is available in only some processors. NIC Offloading improved throughput by 2.5× and 17.5× with and without AES-NI on CPU, respectively. Using AES-NI improved performance on the CPU but to a lesser degree than utilizing all encryption co-processors on the NIC. This result would be difficult to predict without an empirical test.

Flow classification is a stateful task that tracks flow statistics. We categorized flows using the header 5-tuple and used a probabilistic data structure (a count-min sketch) to track the number of bytes per flow. This application ran slightly faster on the NIC. Therefore, it seems reasonable to offload this task to the NIC if we want to spare CPU cycles for other applications.

Network sequencer orders packets based on predefined rules. It performs simple computation and maintains limited in-network state. This function has been used to accelerate distributed system consensus [29] and concurrency control [28]. Our network sequencer was 82% faster on the CPU core than on the NIC. Application throughput did not scale with the number of cores because of the group lock's contention; the number of locks acquired by each packet was 5 out of 10 on average in our synthetic workload, making this task inherently sequential. Therefore, using one fast CPU core yielded the best performance. We also tried running this program using multiple CPU cores, but throughput stayed the same as we increased the number of cores. On the NIC, using three cores offered the highest performance.

In summary, even for simple applications, it is not obvious whether offloading to the NIC improves or degrades performance. Using FLOEM lets us answer these questions quickly and precisely by simply changing the device parameter of the computation segment to either CPU or NIC. Comparing cost-performance or power-performance is beyond the scope of this paper. Nevertheless, one can use FLOEM to experiment with different configurations for a specific workload to optimize for a particular performance objective.

7.2 Logical-to-Physical Queue Mapping

Hypothesis 2 *Logical-to-physical queue mapping lets programmers implement packet steering, packet ordering, and different degrees of resource sharing.*

Packet steering. Storm, the second case study, required packet steering to the correct input queues, each dedicated to one worker. This was done by creating a queue with multiple physical instances and by setting `state.qid` according to an incoming tuple's type.

Packet ordering. The write-back cache implementation required in-order delivery between CPU and NIC to guarantee consistency (see Section 4.1).

Resource sharing. For the split NIC-CPU version of the key-value store, sending both GET and SET requests on separate physical queues offered 7% higher throughput than sharing the same queue. This is because we can use a smaller queue entry's size to transfer data for GET requests. In contrast, for our Storm application, sharing the same physical output queue between all workers yielded 16% higher throughput over separate dedicated physical queues. Since some workers infrequently produce output tuples, it was more efficient to combine tuples from all workers to send over one queue. Hence, it is difficult to predict whether sharing or no sharing is more efficient, so queue resource sharing must be tunable.

7.3 Inferred Data Transfer

Hypothesis 3 *Inferred data transfer improves performance relative to sending an entire packet.*

In this experiment, we evaluated the benefit of sending only a packet's live fields versus sending an entire packet over a queue. We measured the throughput of transmitting data over queues from the NIC to CPU when varying the ratio of the live portion to the entire packet's size (*live ratio*), detailed in Table 4. The sizes of live portions and packets were multiples of 64 bytes because performance was degraded when a queue entry's size was not a multiple of 64 bytes, the size of a CPU cache line. We used numbers of queues and cores that maximized throughput.

Live ratio	1/5	1/4	1/3	1/2	2/3	3/4	4/5
Live size (B)	64	64	64	64	128	192	256
Total size (B)	320	256	192	128	192	256	320
Speedup	3.1x	2.5x	2x	1.5x	1.3x	1.2x	1.2x

Table 4: Speedup when sending only the live portions when varying live ratios from a micro-benchmark. Sizes are in bytes (B).

As shown on the table, sending only live fields improved throughput by 1.2–3.1×. Additionally, we evaluated the effect of this optimization on the split CPU-NIC version of the end-to-end key-value store, whose queues from NIC to CPU transfer packets with a live ratio of 1/2. The optimization improved the throughput of this end-to-end application by 6.5%.

7.4 Queue Synchronization Layer

Hypothesis 4 *The queue synchronization layer enables high-throughput communication queues.*

We measured the throughput of three benchmarks. The first benchmark performed a simple packet forwarding from the NIC to CPU with no network activity, so its performance purely reflects the rate of data transfer over the PCIe bus rather than the rate of sending and receiving packets over the network. We used packet sizes of 32, 64, 128, and 256 bytes. The other two benchmarks were the write-back caching version of the key-value store and the redesigned CPU-NIC version of Storm.

Figure 11 displays the speedup when using the sync layer versus using primitive blocking DMA without batching (labeled "without sync layer"). The sync layer offered 9–15× speedup for pure data transfers in the first benchmark. Smaller packet sizes showed a higher speedup; this is because batching effectiveness increases with the number of packets in a batch. For end-to-end applications, we observed a 7.2–14.1× speedup for the key-value store and a 3.7× speedup for Storm. Note that the sync layer is always enabled in the other experiments. Hence, it is crucial for performance of our system.

7.5 Compiler Overhead

Hypothesis 5 *The* FLOEM *compiler has negligible overhead compared to hand-written code.*

We compared the throughput of code generated from our compiler to hand-optimized programs in C. To measure the compiler's overhead on the CPU, we ran a simple echo program, Storm, and key-value store. The C implementations of Storm and key-value store were taken from FlexStorm and one of FlexKVS's baselines [23]; these implementations are highly-optimized and perform better than the standard public implementations of Storm and memcached. On the NIC, we compared a simple

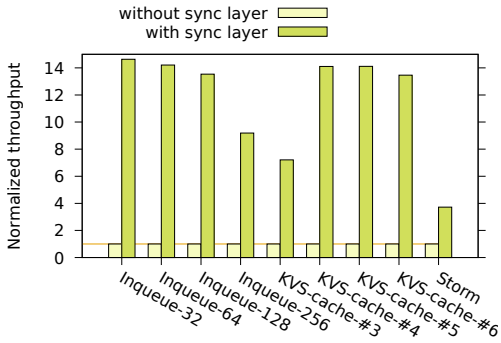


Figure 11: Effect of the queue synchronization layer. Throughput is normalized to that without the sync layer.

echo program, encryption, flow classification, and network sequencer. On average, the overhead was 9% and 1% on CPU and NIC, respectively. We hypothesize that the higher overhead on the CPU was primarily because we did not implement computation batching [24, 46], which was used for hand-optimized programs.

8 Discussion and Future Work

Multi-message packets. FLOEM can support a packet whose payload contains multiple requests via `Batcher` and `Debatcher` elements. Given one input packet, `Debatcher` invokes its one output port n times sequentially, where n is the number of requests in the payload. `Batcher` stores the first $n-1$ packets in its state. Upon receiving the last token, it sends out n packets as one value. The `Debatcher` element can inform the value of n to the `Batcher` element via the per-packet state. One can also take advantage of this feature to support computation batching, similar to Snap [46].

Multi-packet messages and TCP. Exploring the TCP offload with FLOEM is future work. FLOEM supports multi-packet messages via `Batcher` and `Debatcher` elements and could be used together with a TCP offload on the NIC, but our applications do not use TCP.

Shared data structures. In FLOEM, queues and caches are the only high-level abstractions for shared data structures between the NIC and CPU. However, advanced developers can use FLOEM to allocate a memory region on the CPU that the NIC can access via DMA operations, but they are responsible for synchronizing data and managing the memory by themselves.

Automation. Automatic program partitioning was among our initial goals, but we learned that it cannot be done entirely automatically. Different offloading strategies often require program refactoring by rewriting the graph and even graph elements. These program-specific

changes cannot be done automatically by semantics-preserving transformation rules. Therefore, we let programmers control the placement of elements while refactoring the program for a particular offload design. However, FLOEM would benefit from and integrate well with another layer of automation, like an autotuner or a runtime scheduler, that could select parameters for low-level choices (e.g., the number of physical queues, the number of cores, and the placement of each element) after an application has been refactored.

Other SmartNICs. The current FLOEM prototype targets Cavium LiquidIO but can be extensible to other SmartNICs that support C-like programming, such as Mellanox BlueField [32] and Netronome Agilio [6]. However, FPGAs [12, 33, 48] require compilation to a different execution model and the implementation of bodies of elements in a language compatible with the hardware.

9 Related Work

Packet processing frameworks. The FLOEM data-flow programming model is inspired by the Click modular router [34], a successful framework for programmable routers, where a network function is composed from reusable elements [34]. SMP Click [13] and RouteBricks [16] extend Click to exploit parallelism on a multi-processor system. Snap [46] and NBA [24] add GPU offloading abstractions to Click, while ClickNP [27] extends Click to support joint CPU-FPGA processing. Dragonet, a system for a network stack design, automatically offloads computations (described in data-flow graphs) to a NIC with fixed hardware functions rather than programmable cores [43, 44].

Other packet processing systems adopt different programming models. PacketShader [19] is among the first to leverage GPUs to accelerate packet processing in software routers. APUNet [17] identifies the PCIe bottleneck between the CPU and GPU and employs an integrated GPU in an APU platform as a packet processing accelerator. Domain-specific languages for data-plane algorithms, including P4 [10] and Domino [45], provide even more limited operations.

Overall, programming abstractions provided by existing packing processing frameworks are insufficient for our target domain, as discussed in Section 2.

Synchronous data-flow languages. Synchronous data-flow (SDF) is a data-flow programming model in which computing nodes have statically known input and output rates [25]. StreamIt [47] adopts SDF for programming efficient streaming applications on multicore architectures. Flextream [20] extends StreamIt with dynamic runtime adaptation for better resource utilization. More recently, Lime [21] provides a unified programming language based on SDF for programming heterogeneous computers that feature GPUs and FPGAs. Although some variations of these languages support dynamic input/output rates, they are designed primarily for static flows. As a result, they are not suitable for network applications, where the flow of a packet through a computing graph is highly dynamic.

Systems for heterogeneous computing. Researchers have extensively explored programming abstractions and systems for various application domains on various heterogeneous platforms [8, 11, 31, 35, 39, 41, 42]. FLOEM is unique among these systems because it is designed specifically for data-center network applications in a CPU-NIC environment. In particular, earlier systems were intended for non-streaming or large-grained streaming applications, whose unit of data in a stream (e.g., a matrix or submatrix) is much larger than a packet. Furthermore, most of these systems do not support a processing task that maintains state throughout a stream of data, which is necessary for our domain.

10 Conclusions

Developing NIC-accelerated network applications is exceptionally challenging. FLOEM aims to simplify the development of these applications by providing a unified framework to implement an application that is split across the CPU and NIC. It allows developers to quickly explore alternative offload designs by providing programming abstractions to place computation to devices; control mapping of logical queues to physical queues; access fields of a packet without manually marshaling it; cache application state on a NIC; and interface with an external program. Our case studies show that FLOEM simplifies the development of applications that take advantage of a programmable NIC, improving the key-value store's throughput by up to 3.6×.

Acknowledgments

This work is supported in part by MSR Fellowship, NSF Grants CCF–1337415, NSF ACI–1535191, NSF 16-606, and NSF 1518702, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, grants from DARPA FA8750–16–2–0032, by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) as well as gifts from Google, Intel, Mozilla, Nokia, Qualcomm, Facebook, and Huawei.

References

[1] Apache Storm. http://storm.apache.org. Accessed: 2017-11-15.

[2] Cavium Development Kits. http://www.cavium.com/octeon_software_develop_kit.html. Accessed: 2017-11-15.

[3] Cavium LiquidIO. http://www.cavium.com/LiquidIO_Adapters.html. Accessed: 2017-11-14.

[4] DPDK: Data Plane Develepment Kit. http://dpdk.org/. Accessed: 2017-11-07.

[5] IEEE P802.3bs 400 GbE Task Force. Adopted Timeline. http://www.ieee802.org/3/bs/timeline_3bs_0915.pdf. Accessed: 2017-11-16.

[6] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/. Accessed: 2017-11-14.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[8] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[9] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, 2014.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[11] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[12] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16, 2016.

[13] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, 2001.

[14] Cisco. Introduction To RPC/XDR. http://www.cisco.com/c/en/us/td/docs/ios/sw_upgrades/interlink/r2_0/rpc_pr/rpintro.html. Accessed: 2018-09-07.

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, 2017.

[16] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, 2009.

[17] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, 2017.

[18] Google. Protocol Buffers. http://developers.google.com/protocol-buffers/. Accessed: 2018-09-07.

[19] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '10, 2010.

[20] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[21] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In

Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP '08, 2008.

[22] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[23] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[24] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, 2015.

[25] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.

[26] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[27] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '16, 2016.

[28] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[29] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016.

[30] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[31] C. K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, 2009.

[32] Mellanox Technologies. BlueField Multicore System on Chip. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 1018. Accessed: 2018-04-25.

[33] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 1018. Accessed: 2018-04-25.

[34] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP '99, 1999.

[35] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the 22nd ACM Symposium Symposium on Operating Systems Principles*, SOSP '09, 2009.

[36] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016.

[37] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, 2014.

[38] P. M. Phothilimthana. *Programming Abstractions and Synthesis-Aided Compilation for Emerging Computing Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Sept 2018.

[39] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable Performance on Heterogeneous Architectures. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, 2014.

[41] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[42] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[43] P. Shinde, A. Kaufmann, K. Kourtis, and T. Roscoe. Modeling NICs with Unicorn. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, 2013.

[44] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We Need to Talk About NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, 2013.

[45] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '16, 2016.

[46] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, 2013.

[47] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, 2002.

[48] N. Zilberman, Y. Audzevich, G. Kalogeridou, N. Manihatty-Bojan, J. Zhang, and A. Moore. NetFPGA: Rapid Prototyping of Networking Devices in Open Source. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.