HotCocoa: Hardware Congestion Control Abstractions

Mina Tahmasbi Arashloo
Princeton University

Monia Ghobadi
Microsoft Research

Jennifer Rexford
Princeton University

David Walker
Princeton University

ABSTRACT

Congestion control in multi-tenant data centers is an active area of research because of its significant impact on customer experience, and, consequently, on revenue. Therefore, new algorithms and protocols are expected to emerge as the Cloud evolves. Deploying new congestion control algorithms in the end host's hypervisor allows frequent updates, but processing packets at high rates in the hypervisor and implementing the elements of a congestion control algorithm, such as traffic shapers and timestamps, in software have well-studied inaccuracies and CPU inefficiencies. In this paper, we argue for implementing the entire congestion control algorithm in programmable NICs. To do so, we identify the absence of hardware-aware programming abstractions as the most immediate challenge and solve it using a simple high-level domain specific language called HotCocoa. HotCocoa lies at a sweet spot between the ability to express a broad set of congestion control algorithms and efficient hardware implementation. It offers a set of hardware-aware COngestion COntrol Abstractions that enable operators to specify their algorithm without having to worry about low-level hardware primitives. To evaluate HotCocoa, we implement four congestion control algorithms (Reno, DCTCP, PCC, and TIMELY) and use simulations to show that HotCocoa's implementation of Reno perfectly tracks the behavior of a native implementation in C++.

1 INTRODUCTION

Today, congestion control (CC) algorithms play a central role in a data center network's efficiency and its tenants' quality of experience. Hence, a significant number of congestion control algorithms concentrate on data center networks, which greatly benefit from customizing their infrastructure to serve their specific workloads and tenants [3, 4, 8, 9, 23, 29, 31, 33]. This trend is likely to continue, given the impact of network congestion on data centers' revenue and their rapid adoption of new and evolving technologies [12]. Thus, as Cloud computing evolves and new approaches are introduced either by humans [11, 16] or machine learning techniques [13, 32], there is a growing need to enable programmability of CC algorithms.

Having no control over the CC algorithm inside VMs, operators may deploy their CC algorithms in the hypervisor [11, 16]. While this approach enables frequent updates to the CC implementation, it incurs well-studied CPU inefficiencies for doing congestion control and packet switching in software. Implementing traffic shaping in software can add 4% to CPU utilization [27]. Moreover, software-based rate control engines rely on software timers to timestamp packets. These timers are inaccurate as they drift orders of magnitude compared to hardware timers [20, 22, 23]. Worse yet, merely switching packets between the NIC and VMs at 10Gbps can utilize up to 45% of CPUs on a 12-core machine [16]. Thus, with 100Gbps NICs on the horizon, implementing traffic shaping, and similar per-packet stateful processing, at line rate in the hypervisor requires additional CPU cores and memory that could have otherwise been sold to tenants.

To free up CPU cycles on servers, several techniques have been developed for offloading various networking functions to the NIC (e.g., TCP Segmentation Offload [10] and Generic Receive Offload [2]). More recent technologies such as Single Root I/O Virtualization [17] enable VMs to bypass the hypervisor and send packets directly to the NIC, thus triggering efforts to offload VM network policies (e.g., tunneling, NAT, ACLs, etc.) to the NICs [15]. We take this idea to its extreme and propose to implement *the entire CC algorithm* itself in the programmable NICs [24, 26, 30], which are becoming widely deployed [14, 21].

Programming hardware, however, requires niche expertise; even then, it is challenging and time-consuming. The APIs for programmable NICs (e.g., Verilog, and on occasion P4 [7]) are extremely low level, and force network operators to think in terms of a constrained hardware pipeline rather than a high level algorithm. Thus, it takes significant effort to correctly develop and deploy a new CC algorithm using these APIs.

In this paper, we argue for a simple high-level domain-specific language (DSL) for specifying congestion control algorithms in hardware. In other words, our goal is to find a sweet spot that is expressive enough to capture a wide range of congestion control algorithms while being implementable given realistic hardware constraints. Reviewing the extensive literature on congestion control, we observe enough common structure across different CC algorithms to enable the definition of higher-level abstractions that give operators

HotNets '17, November 30–December 1, 2017, Palo Alto, CA, USA

ACM ISBN 978-1-4503-5569-8/17/11...$15.00
https://doi.org/10.1145/3152434.3152457

control over their CC algorithms without requiring them to dig into Verilog code or manage P4 table entries directly. More specifically, we identify four high-level elements involved in any congestion control solution (Figure 1):

- **CC Flock:** A set of packets that must be treated as a group for congestion control purposes. A flock can be a TCP flow, a group of packets originating from the same VM, or all the DNS packets from the same IP address.
- **Configurable Credit Manager:** The CC enforcement mechanism, e.g., a rate limiter or a sliding window, that ensures a CC flock only sends packets when it has the right to do so (i.e., when it has sufficient credit).
- **Accrediting State Machine (ASM):** The CC control algorithm that decides how much credit to allocate to the credit manager depending on the current state. Different CC flocks can be controlled by different ASMs.
- **Network Monitor:** A module that monitors network *events*, collects *metrics*, and triggers state changes in ASMs accordingly.

Note that we are only abstracting the congestion control aspect of data transport as VMs run their own transport protocol and can deal with flow control and generating retransmissions.

Inspired by this observation, we propose HotCocoa, a set of hardware-aware COngestion COntrol Abstractions. These abstractions enable operators to specify each of the above elements at a high level using a simple DSL, such that they can be efficiently implemented in hardware. Using HotCocoa, operators can define the metrics and events of interest that guide the CC algorithm,whether those events originate at traffic sources or destinations. They do not need to concern themselves with the low-level mechanisms by which information is communicated back-and-forth between system components.

To implement HotCocoa, we propose a compilation strategy that takes a HotCocoa program as input and generates configurations for FPGA-based NIC hardware pipelines at traffic sources and destinations. To evaluate our proposed language, we implement four well-known CC algorithms (Reno [5], DCTCP [3], PCC [13], TIMELY [23]) in HotCocoa [1]. Using simulations, we show that our implementation of Reno perfectly tracks its implementation in NS2.

2 HOTCOCOA

HotCocoa enables operators to specify the four elements of a congestion control solution at a high level of abstraction: (i) CC flocks, (ii) credit managers, (iii) network monitors, and (iv) ASMs. This section describes the design constraints that drive the development of HotCocoa's programming abstractions, (§2.1), and explains the abstractions themselves (§2.2).

2.1 Design Constraints

Design Constraint #1: Packets of a CC flock can only drive the update of their own flock's ASM state. ASMs are stateful, and update their internal state based on events and metrics collected by the network monitor. Each ASM controls a single CC flock. Thus, the implementation of a HotCocoa program on a NIC needs to keep per-flock state and update it at line rate upon packet arrival. However, performing memory reads and writes is expensive and spans multiple clock cycles; therefore, it cannot be done multiple times per packet at line rate. This is not a new problem. Studies have shown that end hosts in today's data centers may have up to a few million flows, but have only a few thousand active at any time [6]. Thus, to enable stateful processing on packets based on their flow's state, hardware designs for programmable NICs store active flows and their state in a fast on-chip cache that can be accessed with negligible overhead; furthermore, they only access the memory on the first packet of a flow to insert it into the cache. Therefore, these designs can achieve line rate stateful processing *as long as packets of a flow only access the state of their own flow*. Following a similar approach, our design ensures that in a HotCocoa program, a packet traveling from one end host to the other and updating user-defined metrics can only trigger updates to the ASM of the CC flock to which it belongs.

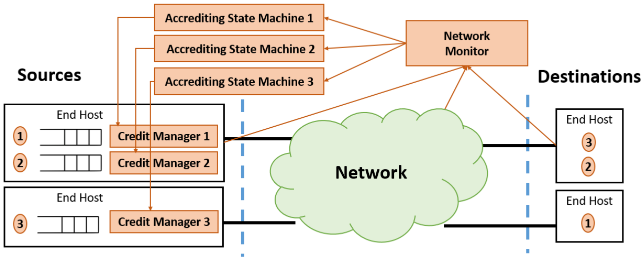


Figure 1: Programmable congestion control architecture

Design Constraint #2: The ASM of a CC flock should only rely on metrics collectable at either its source or its sinks to make credit allocation decisions. This is because the ASM itself will reside either on the source (and directly configure the credit manager) or on the destination (and send credit manager configurations back to the source to be applied). We want to make the metrics it needs available to it in, at most, an RTT for fast convergence. With the above constraint, the metrics can either be co-located with the ASM or reside on the other end of communication and be piggy-backed within an RTT on the flock's traffic. If the metrics reside anywhere else, we would need a large volume of control packets for a similar effect.

2.2 HotCocoa's Abstractions

This section describes HotCocoa's programming abstractions in more detail using the following example.

Running Example. Suppose each VM v in a data center network receives traffic from a set of end hosts S_v, and the operator wants to set the maximum allowed sending rate of each end host $i \in S_v$ in proportion to its current sending rate, such that the total traffic received at v stays within its bandwidth capacity c. Here, the CC flocks of interest are sets of packets going from the same source to the same destination. The network monitor should collect the rate at which each VM v receives traffic from each CC flock i, denoted as r_v^i, and keep track of the sum for all flocks in S_v ($\Sigma_{j \in S} r_v^j$). The control algorithm for each flock needs to know the values of r_v^i and the sum of the values. If the sum goes above c, the algorithm re-calculates i's maximum allowed sending rate as $\frac{C \times r_v^i}{\Sigma_{j \in S} r_v^j}$.

2.2.1 CC Flocks.

A CC flock is a set of packets that should be treated as a group for congestion control purposes. We distinguish between two types of CC flocks, Streams and Datagrams. Streams, such as TCP flows, are CC flocks in charge of the reliable transfer of bytestreams and therefore exchange control messages at the transport layer before, during, and after data transfer (TCP connection establishment, acknowledgments, and tear down). Datagrams, in contrast, consist of individual messages sent in separate packets and do not differentiate between data and control packets at the transport layer. Making this distinction is necessary for a congestion control programming interface, as some CC algorithms need control messages (e.g., acknowledgments), only available in Streams, to regulate the transmission of data packets.

A CC flock is just a set of packets. Thus, to define a CC flock, programmers should specify the set of header fields that distinguish the packets of that flock, i.e., header fields with the same specified value in all the packets of that flock. For instance, following is the definition of a UDP flow as a Datagram flock in HotCocoa:

```
Datagram udp1([srcip = 10.0.0.1, dstip = 10.0.0.2,
    ip_proto = UDP, srcport = 5000, dstport = 80]);
```

Defining Stream flocks is similar but requires additional arguments which specify both the format of control packets for connection setup and tear down and acknowledgements.

Specifying CC flocks one by one can be cumbersome. Therefore, we need a compact way of defining *a set* of CC flocks together. This can be done by specifying a set of values for each header field. For instance, the CC flocks of our running example can be defined in HotCocoa as follows:

```
Datagram ex_flock([srcip = s, dstip = d |
                   s <- all_ips, d <- all_ips, s != d]);
```

The above code snippet defines a set of CC flocks, each of which is a set of packets with a source and destination IP address from the set of all IP addresses, such that the source and destination IP addresses are not the same. We define these flocks as Datagrams because, in our example, they are supposed to be rate-limited per packet, independent of whether they are transferring bytestreams or individual messages.

2.2.2 Network Monitoring.

Suppose we have a central controller, with comprehensive knowledge about the network structure and traffic demands of all the endpoints at any point of time, that can choose the path of each packet. In such a setting, each endpoint can be scheduled to send packets into the network, such that a desired global utility function is maximized at all times [25]. Such a centralized approach to congestion control, however, is not likely to easily scale to data centers with hundreds of thousands of endpoints. As a result, we target the more common distributed congestion control solutions, in which each endpoint relies on *events* and *metrics* collected on the endpoints of communication (receipt of acks, timestamps, etc.) and/or feedback from the network (ECN, H_Feedback [19], etc.) to schedule packet transmissions. More specifically, we need to enable CC algorithms to closely monitor packets of each flock as they go from their source to their destination and collect their metrics of interest.

To do so, HotCocoa breaks a packet's journey into its significant stages and triggers events as packets enter and exit each stage. Programmers can register user-defined callback functions for these events to collect metrics or update the state of ASMs to reconfigure the credit managers. Note that to allow efficient hardware implementation, the scope of possible computation within a call-back function in HotCocoa is inevitably limited. However, we argue that the programming model is general enough to allow programmers to specify virtually any desired packet-level statistics, as the set of primitive events cover all the significant transitions of the packet that are visible from the communication endpoints. The rest of this section describes these events and explains how they can be used in HotCocoa to collect metrics.

Events. Primitive events are those triggered as packets of a CC flock travel through different stages, as shown in Figure 1:

- **Source.** Each CC flock has its own packet queue at the source. A packet enters this stage by being enqueued when the CC flock wants to send it out (`pkt_enqueued`), and exits by being dequeued when the flock's credit manager has enough available credit for it (`pkt_dequeued`).
- **Network.** The packet enters this stage once it is sent out toward the destination (`pkt_sent`). The packet travels across the network, during which it can be dropped, delayed, or tagged with extra information about the network; it exits this stage when it arrives at its destination.
- **Destination.** The packet enters this stage when it arrives at the NIC at the destination end host (`pkt_rcvd`).

For Streams, similar events are triggered when acks travel from the destination back to the source to support the broad set of CC algorithms that use acks to regulate packet transmissions [3, 5, 13, 23]. Moreover, CC algorithms use timers to detect packet loss, pace credit allocation, etc. Thus, HotCocoa allows programmers to define their own timers and register call-back functions for their `timeout`s.

Primitive events are useful for calculating metrics, such as maximum acked sequence number or received rate, that are updated on a per-packet basis. However, to ensure stability, most CC algorithms update their state to reconfigure credit managers at a much coarser level of granularity, e.g., after the transmission of a window of packets, or when the RTT rises above a threshold. Thus, HotCocoa provides programmers with ways to define their own events with their desired level of granularity using metrics collected from the network. Suppose a HotCocoa program defines the metric `avg_rtt`. Programmers can create an event that will be triggered when this metric dips below (rises above) a certain level, say 5ms, using `falling(avg_rtt, 5)` (`falling(avg_rtt, 5)`). Moreover, they can use boolean expressions over metrics to make an existing event conditional (e.g., `pkt_rcvd if avg_rtt < 5ms`). Finally, events can be combined using conjunction (`&&`) and disjunction (`||`). For instance, `(e1 || e2) && e3` is triggered if either `e1` or `e2` is triggered alongside `e3`.

Metrics. Metrics are statistics collected as packets travel from their source to their destination. When a packet transitions from one stage to the next, the callback functions for the metrics registered for that event are executed to update the values of those metrics. This, in turn, could trigger a chain of other events or updates to ASMs' states. Recall from our first design constraint that for efficient hardware implementation, packets of a flock should only access the state of their own flock. Thus, HotCocoa's metrics are collected per-flock, but as will be discussed later, they can also be aggregated, with certain limitations, across flocks that have the same source or destination

Metrics associated with a single flock are called *simple metrics*. A simple metric can be defined by specifying (i) the metric state, (ii) the metric initialization code, (iii) the set of events that trigger metric update, and (iv) the callback function for each event that is a simple imperative program without loops, i.e., a sequence of assignments and conditionals. These functions are known to be easily pipelined and efficiently implemented in hardware at line rate [28]. A simple metric in our running example is the rate at which each flock's traffic is received at the destination:

```
Metric rcvd_rate(Flock f){
  Int val, cnt, start, interval = 200;
  Timer t;

  def init(){
     val = 0; _cnt = 0; start = now; t.set(interval);}
  def update(f.pkt_rcvd e){cnt += e.pkt.len;}
  def update(t.timeout e){
    val = cnt / (now - start);
    cnt = 0; start = now; t.set(interval);}
  def Val() {return val;}}
```

Other common metrics, such as send and receive rate in terms of bytes and packets, number of sent and received bytes and packets, queue length at the source, and packet drops, can be similarly implemented and provided as a built-in library.

An *aggregate metric* combines a simple metric across a set of flocks. For instance, in our running example, we need to measure the total rate at which traffic is received by each VM from all flocks communicating with it. Using the code snippet below, we can group CC flocks in `ex_flock` by destination IP address and combine `rcvd_rate` of CC flocks in each group.

```
AggrMetric t_rr(sum, rcvd_rate, ex_flock, [], [dstip]);
```

Aggregate metrics cannot be used with `falling` and `rising` to define events. To see why, suppose we define an event `rising(t_rr, 1Gbps)` and use it in the ASM of the CC flocks in `ex_flock`. A packet from flock `x` in `ex_flock` can cause `_rr` to go above 1Gbps; as a result, the ASM of multiple flocks in `ex_flock` may require updating. Thus, aggregate metrics can only be used in boolean expressions to condition other events (e.g., `pkt_rcvd if t_rr > 1Gbps`). Moreover, because of our second design constraint, an aggregate metric should group CC flocks based on source IP or destination IP or both so that if it is used in the ASM of a CC flock, it can be located at either the traffic source or the sink.

2.2.3 Credit Managers. Every CC algorithm has a credit manager that enforces its decisions by releasing the queued packets of a CC flock only if it has enough credits (Figure 1). A credit manager needs to (i) have internal variables to keep track of a flock's credits, (ii) decide when there is enough credit to send a packet out (`deq_criteria`), (iii) register for relevant primitive events such as enqueue and dequeue of packets and receipt of acks to manage credits as packets fly by, and (iv) expose parameters that ASM can configure. Our running example, for instance, uses rates to control packet transmissions. The following example illustrates a credit manager based on a token bucket rate limiter, that can be used to enforce these rates. This credit manager exposes the rate and capacity parameters to ASMs for configuration.

```
CreditManager Token_Bucket (Flock f) {
  expose Int rate, cap;
  Int toks; Timer t;

  def init() {t.set(200);}
  def deq_criteria(Packet pkt){return pkt.len <= toks;}
  def on_enqueue(Packet pkt){}
  def on_deqeue(Packet pkt){toks -= pkt.len;}
  def on_timeout<t>(){
    toks += rate * 0.2; t.set(200);
    if (toks > cap){toks = cap;}}}
```

We have implemented other common credit management schemes, such as sliding windows for Streams, and can provide these as a library for HotCocoa programmers.

2.2.4 Accrediting State Machines. The core of a congestion control solution is the control algorithm that allocates credit to the credit manager of each flock. Looking at the CC algorithms in the literature, we observe that they consist of a set of states reflecting their beliefs about the possible states of the network, and transition between states based on their observations of the network. In each state, simple rules determine how much credit is allocated to the credit manager. Thus, HotCocoa abstracts the CC algorithm as an Accrediting State Machine (ASM) that executes a simple imperative program in each state to configure the credit manager of a flock (`fl.CM`), and transitions between states on certain events. The following code illustrates how to implement the ASM in our running example.

```
ASM ex_asm (Flock f, Params p){
  State Normal{};
  State Congestion{f.cm.rate = f.rcvd_rate/f.t_rr;};
  Init Normal; // Declare initial State

  Transitions {
  | Normal  on f.pkt_rcvd if f.t_rr > p.max_rate
         ==> Congestion;
  | Congestion on f.pkt_rcvd if f.t_rr <= p.max_rate
         ==> Normal
  };}
```

The ASM shown above loops in the `Normal` state without changing the corresponding token bucket as long as the total received rate stays below the `max_rate`. If a packet is received but the total received rate is above `max_rate`, it enters the `Congestion` state, calculates a new rate for its corresponding flock, and configures the token bucket accordingly.

Once CC flocks and ASMs are defined, the programmer can specify which ASM should regulate the packet transmission of which CC flocks using the `CoCo` (COngestion COntrol) primitive: `CoCo(ex_flock, token_bucket, ex_asm, {max_rate : 1Gbps})`. This will program the credit manager of each flock in `ex_flock` as a `token_bucket`, and track the metrics used in its ASM, i.e. `rcvd_rate` and `t_rr`, to trigger state changes in the ASM.

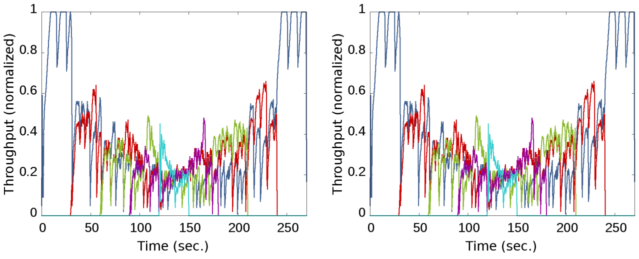
3 LANGUAGE EVALUATION

In this section, we evaluate HotCocoa's expressiveness and provide a proof of concept, using simulation, that a CC algorithm implemented in HotCocoa can safely replace its native implementation in general-purpose programming languages.

Expressiveness. We implemented four well-known CC algorithms, Reno, DCTCP [3], PCC[1] [13], and TIMELY [23], in HotCocoa (the programs are available at [1]). Each program includes the definition of all four components of HotCocoa: CC flocks, a network monitor, ASM, and a credit manager. For all the above CC algorithms, the CC flocks of interest are TCP flows. For Reno and DCTCP, the enforcement module is a sliding window, whereas for PCC and TIMELY, it is a token bucket rate limiter. Table 1 shows the number of lines of codes used to implement each element of each algorithm. Overall, all the algorithms can be expressed in HotCocoa in less than 150 lines of code.

Soundness. As a proof of the soundness of the CC algorithms implemented in HotCocoa, we wrote a simple C++ program to simulate our implementation of Reno in HotCocoa, incorporated it into NS2, and compared the result with NS2's implementation of Reno to show that our implementation perfectly follows the behavior of that of NS2 in terms of throughput and congestion window.

We created a topology with six end hosts, five as senders and one as receiver, all connected to a router. The round-trip time between the senders and receiver was 100ms. We started a single TCP flow from one of the hosts, and sequentially started and stopped other senders at 30 second intervals. We performed the experiment once with NS2's native Reno and once with HotCocoa's. Figures 2(a) and 2(b) depict the throughput of flows over time; as the figures show, they were perfectly matched in both runs of the experiment. We repeated the experiment with only one sender and observed that the calculated congestion windows for HotCocoa's implementation of Reno perfectly match that of NS2's native implementation. We take our results as a preliminary proof of concept that programs written in HotCocoa can safely replace their native implementation in general-purpose programming languages. We are planning to perform similar experiments with other CC algorithms, and once we prototype the compiler and can push these algorithms to hardware.



(a) Reno - NS2 (b) Reno - HotCocoa

Figure 2: Reno's implementation in HotCocoa *perfectly* tracks Reno's implementation in NS2. (a) and (b) compare the throughput for 5 flows sequentially started and stopped at 30 second intervals.

CC Algorithm	Monitoring #LoC	Control Algorithm #LoC	Credit Manager #LoC
Reno	64	45	12
DCTCP	97	45	12
PCC	39	92	17
TIMELY	9	34	17

Table 1: #LoC used for implementing different CC algorithms in HotCocoa.

4 COMPILATION STRATEGY

To compile HotCocoa programs to programmable NICs, we propose a logical pipeline with stages for the elements of CC algorithms, and a compilation strategy to map HotCocoa programs to the pipeline. We leave a full implementation of the logical pipeline on a programmable NIC and the compiler for future work.

4.1 Logical Target Pipeline

Our proposed logical pipeline is depicted in Figure 3. It contains a DRAM and a fast cache in SRAM to keep HotCocoa's program state and packet queues (the trapezoid in the middle), a logical block for flock classification, logical blocks for implementing credit management logic and handling of events, metrics, and ASMs, and a TX scheduler that picks which flock should send a packet out next. There are separate ingress and egress pipelines that process packets, and a timer pipeline that handles timeout events, all of which can access the program state from the memory. Note that this is a preliminary proposed design, and several hardware mechanisms, including ensuring ASM consistency among handler blocks, need to be carefully designed.

Memory Module. The program state, including per-flock state as well as aggregate metrics, is stored in a memory module consisting of a DRAM and an SRAM cache for storing active flocks. The per-flock state includes the ASM state and the simple metrics it uses, its credit manager state, and its packet queue head and tail pointers.

Flock Classifier. There is a logical block at the beginning of both ingress and egress pipelines that classifies packets

[1] As HotCocoa currently does not support random number generation, we implemented PCC with deterministic testing of sending rates.

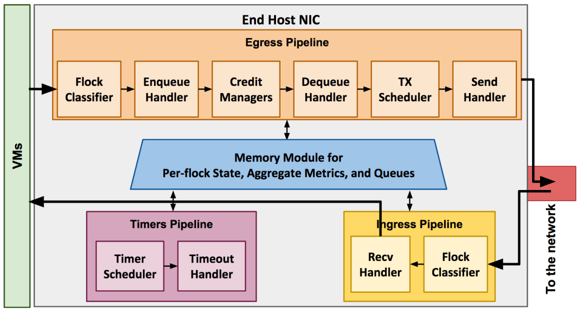


Figure 3: Proposed logical pipeline for HotCocoa programs on the NIC.

into their corresponding flock. It can be implemented with a TCAM with ternary matches.

Credit Manager and TX Scheduler. There is a logical block in the egress pipeline that implements the credit manager logic in the program and decides when to dequeue packets based on their available credit. The packets are then processed by the TX Scheduler that decides which flock can send its packet out next. Each credit manager has a *service queue* for configuration requests. It pulls these requests from its service queue and applies them to its parameters.

Handler Blocks. We have a *handler block* for each primitive event. An x handler block records the occurrence of primitive event x, and updates corresponding metrics and the ASM state if applicable. If a state transition happens in the ASM, the resulting credit manager configurations will be queued in the credit manager's *service queue*. We have handler blocks for enqueue, dequeue, send, and receive.

Timer Pipeline. The Timer pipeline keeps track of timers for timeout events. Timeouts could trigger updates to multiple flocks. Therefore, to make sure they do not stall the packet processing pipelines, we plan to design a scheduler for the Timer pipeline to pick a bounded number of flocks with expired timers at a time and handle their timeout events in the Timeout Handler.

4.2 Compiler

The logical pipeline is designed to have building blocks for different elements of the programming language. Therefore, the central technical challenge of the HotCocoa compiler is to decide whether to place a flock's associated events, metrics, and ASM at the source or the destination.

The placement for primitive events such as `pkt_sent` and `pkt_rcvd` is determined by definition (source and destination, respectively). However, a HotCocoa program can have a metric with multiple `update` functions, each triggered by the occurrence of a different primitive event. These events may not necessarily happen at the same location, and there is a choice to be made about where, at the source or destination, to maintain this metric's value. This choice affects the amount of meta data that is piggy-backed on the data packets of the CC flock to get the metric implementation the information it needs from events to update its value. Moreover, metrics can be used to define events themselves, and these events can, in turn, trigger updates to other metrics. These events and metrics can also be used in ASMs, which need to configure credit managers at traffic sources. Given these dependencies, the compiler needs to decide the optimal location for maintaining the metrics and ASM states, such that the metadata communicated between endpoints to correctly implement the program is minimized. We believe this can be accomplished using a constraint solver. Once the placement is decided, the compiler can generate configurations for each block in Figure 3 according to the logic specified in the input program.

5 RELATED WORK

Data Center Congestion Control Algorithms. There is a vast literature on congestion control algorithms in data centers [3, 4, 8, 9, 18, 23, 29, 31, 33] that use sliding windows or rate limiters to control packet transmission of flows and flow aggregates. HotCocoa provides abstractions for operators to specify these algorithms in a high-level program, and run them directly on the NIC.

Congestion Control in Hypervisor. AC⚡DC [16] and vCC [11] make the case that data center operators need to deploy their own congestion control algorithms in multi-tenant data centers and easily update them based on state-of-the-art. However, they propose these algorithms to be implemented in software in the hypervisor. As we argued earlier, software implementation of traffic shaping schemes, and the timestamps upon which they depend, are inaccurate and CPU inefficient [20, 22, 23, 27], and are not expected to scale at a reasonable CPU cost for next generation NICs. Thus, we enable the deployment of congestion control algorithms directly in programmable NICs by providing a high-level DSL as a solution to the high barrier for hardware programming.

6 CONCLUSIONS

Given the impact of congestion control on tenants' experience in a multi-tenant data center, the ever-evolving nature of the Cloud and its congestion control algorithms, and the need for efficiency and accuracy in packet processing as we move to higher network speeds, we propose implementing congestion control algorithms directly in programmable NICs. To overcome the high barrier for programming hardware, we propose HotCocoa, a high-level DSL for expressing congestion control algorithms that can directly run in hardware. We propose a high-level target pipeline and a compiling strategy to translate HotCocoa programs to pipeline configurations.

ACKNOWLEDGMENTS

This work is supported by DARPA Contract No. HR001117C0047 and NSF grant CNS-1703493. We thank the anonymous reviewers, Victor Bahl, Doug Burger, Adrian Caulfield, Derek Chiou, Daniel Firestone, Roch Guerin, Changhoon Kim, Larry Luo, Ratul Mahajan, Jitendra Padhye, Andrew Putnam, Shachar Reindel, Vishal Shrivastav, and Anirudh Sivaraman for helpful feedback.

REFERENCES

[1] HotCocoa Github Repository. https://github.com/minmit/CoCoA.git.

[2] Generic Receive Offload. https://lwn.net/Articles/358910/, 2008.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[5] M. Allman, V. Paxon, and W. Stevens. TCP Congestion Control, 4 1999. RFC 2581.

[6] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. IMC '10, 2010.

[7] 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 Comput. Commun. Rev.*, 2014.

[8] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 2016.

[9] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling Mix-Flows in Commodity Datacenters with Karuna. In *SIGCOMM*, 2016.

[10] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder. Offload of TCP Segmentation to a Smart Adapter, 1999. US Patent 5,937,169.

[11] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized Congestion Control. In *SIGCOMM*, 2016.

[12] David Maltz. Keeping Cloud-Scale Networks Healthy. https://video.mtgsf.com/video/4f277939-73f5-4ce8-aba1-3da70ec19345.

[13] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. NSDI'15, 2015.

[14] D. Firestone. SmartNIC: Accelerating Azure's Network with FPGAs on OCS servers. https://ocpussummit2016.sched.com/event/68u4/.

[15] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*, 2017.

[16] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*, 2016.

[17] Intel LAN Access Division. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

[18] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[19] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[20] H. Marouani and M. R. Dagenais. Internal Clock Drift Estimation in Computer Clusters. *Journal of Computer Systems, Networks, and Communications*, 2008.

[21] C. Metz. Microsoft Bets its Future on a Reprogrammable Computer Chip. https://www.wired.com/2016/09/microsoft-bets-future-chip-reprogram-fly/, 2016.

[22] D. L. Mills. Precision Synchronization of Computer Network Clocks. *SIGCOMM Comput. Commun. Rev.*, 1994.

[23] R. Mittal, V. The Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[24] Netronome SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[25] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *SIGCOMM*, 2014.

[26] 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 *ISCA*, 2014.

[27] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*, 2017.

[28] 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 *SIGCOMM*, 2016.

[29] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *SIGCOMM*, 2012.

[30] Will Chu. Intelligent Networks by Cavium. http://www.cavium.com/newsevents_Caviumnetworks_CoredgeNetworks.html.

[31] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[32] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *SIGCOMM*, 2013.

[33] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.