Evaluating the Power of Flexible Packet Processing for Network Resource Allocation

Naveen Kr. Sharma[*] *Antoine Kaufmann*[*] *Thomas Anderson*[*] *Changhoon Kim*[†]
Arvind Krishnamurthy[*] *Jacob Nelson*[‡] *Simon Peter*[§]

Abstract

Recent hardware switch architectures make it feasible to perform flexible packet processing inside the network. This allows operators to configure switches to parse and process custom packet headers using flexible match+action tables in order to exercise control over how packets are processed and routed. However, flexible switches have limited state, support limited types of operations, and limit per-packet computation in order to be able to operate at line rate.

Our work addresses these limitations by providing a set of general building blocks that mask these limitations using approximation techniques and thereby enabling the implementation of realistic network protocols. In particular, we use these building blocks to tackle the network resource allocation problem within datacenters and realize approximate variants of congestion control and load balancing protocols, such as XCP, RCP, and CONGA, that require explicit support from the network. Our evaluations show that these approximations are accurate and that they do not exceed the hardware resource limits associated with these flexible switches. We demonstrate their feasibility by implementing RCP with the production Cavium CNX880xx switch. This implementation provides significantly faster and lower-variance flow completion times compared with TCP.

1 Introduction

Innovation in switch design now leads to not just faster but *more flexible* packet processing architectures. Whereas early generations of software-defined networking switches could specify forwarding paths on a per-flow basis, today's latest switches support configurable per-packet processing, including customizable packet headers and the ability to maintain state inside the switch. Examples include Intel FlexPipe [26], Texas Instruments's Reconfigurable Match Tables (RMTs) [13], the Cavium XPliant switches [15], and Barefoot's Tofino switches [10]. We term these switches *FlexSwitches*.

FlexSwitches give greater control over the network by exposing previously proprietary switching features. They can be reprogrammed to recognize, modify, and add new header fields, choose actions based on user-defined match rules that examine arbitrary components of the packet header, perform simple computations on values in packet headers, and maintain mutable state that preserves the results of computations across packets. Importantly, these advanced data-plane processing features operate at line rate on every packet, addressing a major limitation of earlier solutions such as OpenFlow [22] which could only operate on a small fraction of packets, e.g., for flow setup. FlexSwitches thus hold the promise of ushering in the new paradigm of a *software defined dataplane* that can provide datacenter applications with greater control over the network's datapaths.

Despite their promising new functionality, FlexSwitches are not all-powerful. Per-packet processing capabilities are limited and so is stateful memory. There has not yet been a focused study on how the new hardware features can be used in practice. Consequently, there is limited understanding of how to take advantage of FlexSwitches, nor is there insight into how effective are the hardware features. In other words, do FlexSwitches have *instruction sets* that are sufficiently powerful to support the realization of networking protocols that require in-network processing?

This paper takes a first step towards addressing this question by studying the use of FlexSwitches in the context of a classic problem: network resource allocation. We focus on resource allocation because it has a rich literature that advocates *per-packet dataplane processing* in the network. Researchers have used dataplane processing to provide rate adjustments to end-hosts (congestion control), determine meaningful paths through the network (load balancing), schedule or drop packets (QoS, fairness), monitor flows to detect anomalies and resource exhaustion attacks (IDS), and so on. Our goal is to evaluate the feasibility of implementing these protocols on a concrete and realizable hardware model for programmable data planes.

We begin our study with a functional analysis of the set of issues that arise in realizing protocols such as the RCP [17] congestion control protocol on a FlexSwitch. At first sight, the packet processing capabilities of FlexSwitches appear to be insufficient for RCP and associated protocols; today's FlexSwitches provide a limited number of data plane operators and also limit the number of operations performed per packet and the amount of state maintained by the switch. While some of these limits can be addressed through hardware redesigns that support an enhanced set of operators, the limits associated with switch state and operation count are likely harder

[*]University of Washington
[†]Barefoot Networks
[‡]Microsoft Research
[§]University of Texas at Austin

to overcome if the switches are to operate at line rate. We therefore develop a set of building blocks designed to mask the absence of complex operators and to overcome the switch resource limits through the use of approximation algorithms adapted from the literature on streaming algorithms. We then demonstrate that it is possible to implement approximate versions of classic resource allocation protocols using two design principles. First, our approximations only need to provide an appropriate level of accuracy, taking advantage of *known workload properties* of datacenter networks. Second, we *co-design* the network computation with end-host processing to reduce the computational demand on the FlexSwitch.

Our work makes the following contributions:

- We identify and implement a set of building blocks that mask the constraints associated with FlexSwitches. We quantify the tradeoff between resource use and accuracy under various realistic workloads.
- We show how to implement a variety of network resource allocation protocols using our building blocks.
- In many instances, our FlexSwitch realization is an approximate variant of the original protocol, so we evaluate the extent to which we are able to match the performance of the unmodified protocol. Using ns-3 simulations, we show that our approximate variants emulate their precise counterparts with high accuracy.
- Using an implementation on a production FlexSwitch and emulation on an additional production FlexSwitch hardware model, we show that our protocol implementations are feasible on today's hardware.

2 Resource Allocation Case Study

We begin by studying whether FlexSwitches are capable of supporting RCP, a classic congestion control protocol that assumes switch dataplane operations. We first describe an abstract FlexSwitch hardware model based on existing and upcoming switches. We then examine whether we can implement RCP on the FlexSwitch model. We identify a number of stumbling blocks that would prevent a direct implementation.

FlexSwitch hardware model. Rather than supporting arbitrary per-packet computation, most FlexSwitches provide a *match+action* (M+A) processing model: match on arbitrary packet header fields and apply simple packet processing actions [13]. To keep up with packets arriving at line rate, switches need to operate under tight real-time constraints. Programmers configure how FlexSwitches process and forward packets using high-level languages such as P4 [12].

We assume an abstract switch model as described in [12] and depicted in Figure 1. On packet arrival, FlexSwitches parse packet headers via a user-defined parse graph. Relevant header fields along with packet meta-

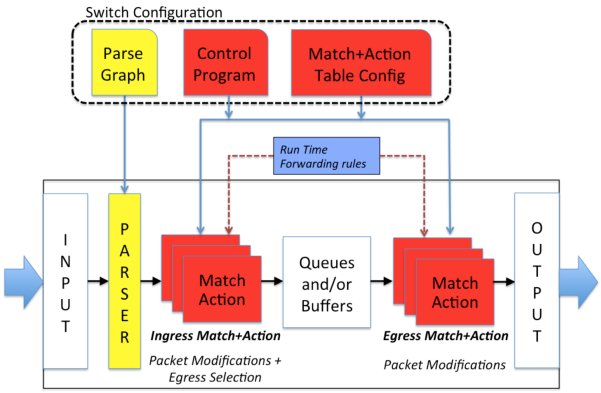


Figure 1: Abstract Switch Model (from [12]).

data are passed to an ingress pipeline of user-defined M+A tables. Each table matches on a subset of extracted fields and can apply simple processing primitives to any field, ordered by a user-defined control program. The ingress pipeline also chooses an output queue for the packet, determining its forwarding destination. Packets pass through an egress pipeline for destination-dependent modifications before output. Legacy forwarding rules (e.g., IP longest prefix match) may also impact packet forwarding and modification.

To support packet processing on the data path, FlexSwitches provide several hardware features:

Computation primitives that perform a limited amount of processing on header fields. This includes operations such as addition, bit-shifts, hashing, and max/min.

M+A tables generalize the abstraction of match+action provided by OpenFlow, allowing matches and actions on any user-defined packet field. Actions may add, modify, and remove fields.

A limited amount of *stateful memory* can maintain state across packets, such as counters, meters, and registers, that can be used while processing.

Switch meta-data, such as queue lengths, congestion status, and bytes transferred, augment stateful memory and can also be used in processing.

Timers built into the hardware can invoke the switch-local CPU to perform periodic computation such as updating M+A table entries or exporting switch meta-data off-switch to a central controller.

Queues with strict priority levels can be assigned packets based on packet header contents and local state. Common hardware implementations support multiple queues per egress port with scheduling algorithms that are fixed in hardware. However, the priority level of any packet can be modified in the processing pipeline, and packets can be placed into any of the queues.

Packets may be sent to a *switch-local control plane CPU* for further processing, albeit at a greater cost. The CPU is able to perform arbitrary computations, for example on packets that do not match in the pipeline. It

cannot guarantee that forwarded packets are processed at line rate; instead, it drops packets when overloaded.

RCP. Rate Control Protocol (RCP) [17] is a feedback-based congestion control algorithm that relies on explicit network feedback to reduce flow completion times. RCP attempts to emulate processor sharing on every switch and assigns a single maximum throughput rate to all flows traversing each switch. In an ideal scenario, the rate given out at time t is simply $R(t) = C/N(t)$, where C is the link capacity and $N(t)$ is the number of ongoing flows, and the rate of each flow is the minimum across the path. Each switch computes $R(t)$ every *control interval*, which is typically set to be the average round-trip time (RTT) of active flows.

In RCP, every packet header carries a rate field R_p, initialized by the sender to its desired sending rate. Every switch on the path to the destination updates the rate field to its own $R(t)$ if $R(t) < R_p$. The receiver returns R_p to the source, which throttles its sending rate accordingly.

The packet header also carries the source's current estimate of the flow RTT. This is used by each switch to compute the control interval. For a precise calculation, per-flow RTTs need to be kept[1].

The original RCP algorithm computes the fair rate $R(t)$ using the equation

$$R(t) = R(t-d) + \frac{\alpha \cdot S - \beta \cdot \frac{Q}{d}}{\hat{N}(t)}$$

where d is the control interval, S is the spare bandwidth, Q is the persistent queue size and α, β are stability constants. $\hat{N}(t)$ is the estimated number of ongoing flows. This *congestion controller* maximizes the link utilization while minimizing queue depth (and drop rates). If there is spare capacity available (i.e., $S > 0$), RCP increases the traffic allocation. If queueing is persistent, RCP decreases rates until queues drain.

FlexSwitch constraints. Although FlexSwitches are flexible and reconfigurable in several ways, they do impose several restrictions and constraints. If these limits are exceeded, then the packet processing code cannot be compiled to the FlexSwitch target. We describe these limits, providing typical values for them based on both published data [13] and information obtained from manufacturers, and note how they impact the implementation of network resource allocation algorithms, like RCP.

Processing primitives are limited. Each switch pipeline stage can execute only one ALU instruction per packet field, and the instructions are limited to signed addition and bitwise logic. Multiplication, division, and floating point operations are not feasible. Hashing primitives, however, are available; this exposes the hardware that switches now use for ECMP and related load-balancing protocols. Control flow mechanisms, such as loops and pointers, are also unavailable, and entries inside M+A tables cannot be updated on the data path. This precludes the complex floating-point computations often employed by resource allocation algorithms from being used directly on the data path. It also limits the number of sequential processing steps to the number of M+A pipeline stages (generally around 10 to 20). Within a pipeline stage, rules are processed in parallel.

Available stateful memory is constrained. Generally, it is infeasible to maintain per-flow state across packets in both reconfigurable switches and their fixed function counterparts.[2] For example, common switches support SRAM-based exact match tables of up to 12 Mb per pipeline stage. The number of rules is limited by the size of the TCAM-based ternary match tables provided per pipeline stage (typically up to 1Mb). In contrast, it is common for a datacenter switch to handle tens of thousands to hundreds of thousands of connections. This allows for a negligible amount of per-connection state, which is likely not enough for most resource allocation algorithms to perform customized flow processing (e.g., for RCP to compute a precise average RTT).

State carried across computational stages is limited. The hardware often imposes a limit on the size of the temporary *packet header vector*, used to communicate data across pipeline stages. Common implementations limit this to 512 bytes. Also, switch metadata may not be available at all processing stages. For example, cut-through switches may not have the packet length available when performing ingress processing. This precludes that information from being used on the data path, severely crimping link utilization metering and flow set size estimation as needed by RCP.

FlexSwitch evolution. It is likely that the precise restrictions given in this section will be improved with future versions of FlexSwitch hardware. For example, additional computational primitives might be introduced or available stateful memory might increase. However, restrictions on the use of per-flow state, the amount of processing that can be done, and the amount of state that can be accessed per packet will have to remain to ensure that the switch can function at line rate.

3 A Library of Building Blocks

To recap, realizing RCP requires additional operations that are not directly supported by FlexSwitches. Specifically, an implementation of RCP requires the following basic features:

[1] However, it is possible to approximate the average RTT and only keep an aggregate of the number of flows.

[2] The dramatic growth in link bandwidths coupled with a much slower growth in switch state resources implies that this constraint will likely hold in future generations of datacenter switches.

- *Metering utilization and queueing:* For every outgoing port on a FlexSwitch, given a pre-specified control interval, we need a metering mechanism that provides the average utilization and the average queueing delay.
- *Estimating the number of flows:* For every outgoing port and a pre-specified control interval, we need an estimate of the number of flows that were transmitted through the port in the previous control interval—without keeping per-flow state.
- *Division/multiplication:* They are required to compute $R(t)$, but are not supported in today's FlexSwitches.

We performed a similar functional analysis of a broad class of resource allocation algorithms to identify their computational and storage requirements. Table 6 presents our findings. We can see that the requirements of most network resource allocation protocols can be distilled into a relatively small set of common *building blocks* that can enable the implementation of these algorithms on a FlexSwitch.

We outline the design of these and other related building blocks in this section. The building blocks are designed to overcome hardware limitations described in Section 2. In particular, we facilitate the following types of packet processing functionality: (a) flow-level measurements such as approximate maintenance of per-flow state and approximate aggregation of statistics across flows, (b) approximate multiplication and division using simpler FlexSwitch primitives, and (c) switch-level measurements such as metering of port-level statistics and approximate scans over port-level statistics.

A common theme across many of these building blocks is the use of approximation to stay within the limited hardware resource bounds. We apply techniques adapted from the streaming algorithms literature within the context of FlexSwitches. Streaming algorithms use limited state to approximate digests as data is streamed through. Often, there is a tradeoff between the accuracy of the measurement and the amount of hardware resources devoted to implementing the measurement. We draw attention to these tradeoffs and evaluate them empirically. We do so by measuring the resource use of faithful C/C++ implementations of the building blocks under various configuration parameters. In later sections, we use P4 specifications of these building blocks, evaluate performance using a production FlexSwitch, and measure resource requirements in an emulator for an additional production FlexSwitch.

3.1 Flow Statistics

Approximating Aggregate Flow Statistics: Many resource allocation algorithms require the computation of aggregate values such as the total number of active flows, the total number of sources and destinations communicating through a switch, and so on. Exact computation of these values would require large amounts of both state and per-packet computation. We therefore design a **cardinality estimator** building block that can approximately calculate the number of unique elements (typically tuples of values associated with a subset of header fields) in a stream of packets in a given time frame.

Our approach is adapted from streaming algorithms [28]. We calculate a hash of each element and count the number of leading zeros n in the binary representation of the result using a TCAM. Using this approach, we compute and store the maximum number $\max(n)$ of leading zeros over all elements to be counted. $2^{\max(n)}$ is then the expected number of unique elements. To implement this calculation on FlexSwitches, we use a ternary match table described by the string:

$$0^n 1 x^{N-n-1} \qquad 0 \leq n < N$$

where N is the maximum number of bits in the hash result. A stateful memory cell maintains the maximum number of leading zeros observed in the stream. This approach allows us to both update and retrieve the count n on the data plane.

This basic approach is space-efficient but exhibits high variance for the estimate. The variance can be reduced by using multiple independent hashes to derive independent estimates of the cardinality. These independent estimates are averaged to produce a final estimate. An alternative is to divide the incoming stream into k disjoint buckets. For example, we can use the last $log(k)$ bits of the hash result as a bucket index. We can estimate the cardinality of each bucket separately and then combine them to derive the final estimate by taking the harmonic mean [28].

Different traffic properties can be estimated by providing different header fields to the hash function. For example, if the header fields are the 4-tuple of source IP, destination IP, source port, and destination port, then we can estimate the number of active flows traversing a switch. Similarly, we can calculate the number of unique end-hosts/VMs/tenants traversing the switch or collect traffic statistics associated with anomalies.

The parameters are the number h of hash functions to use, the maximum number N of bits in each hash result, and the number k of hash buckets to use. Given these parameters, we can ask: how much switch state is required to achieve a given level of accuracy and how does accuracy trade off with switch state?

Prior datacenter studies [4, 31], measure the number of concurrent flows per host to be 100-1000. This means a ToR switch can have 1,000-50,000 flows. Assuming we use a single hash function with $N = 32$, and 1 byte of memory for counting the number of leading zeroes in each bucket, Figure 2 shows the tradeoff of accuracy versus memory usage for different flow set sizes. We vary

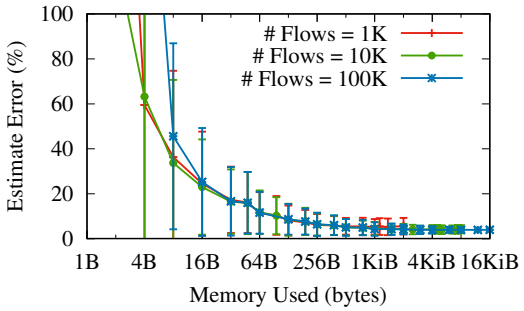


Figure 2: Average tradeoff between accuracy and memory used by the cardinality estimating building block for different set sizes of random flows over 1000 measurements (error bars show standard deviation).

memory use, which equals $h \times k$ bytes, for $k \in [1, 2^{16}]$ and $h \in [1, 8]$ and show the tradeoff for the best choice of h and k for a particular memory size.

Accuracy improves with increased memory, but the flow set size does not impact this tradeoff significantly. Using less than 64 bytes produces error rates above 20%. Memory between 64 bytes and 1 KB has average error rates between 5% and 15%. Using more than 1 KB of memory only improves average error rates marginally (to 4%), but it can improve worst-case error rates.

We do lose accuracy for very small ($\#Flows \approx k$) and very large ($\#Flows \approx 2^N$) flow set sizes. For very small flow sets, several buckets will remain empty. This introduces distortions which can be corrected by calculating the estimate as $\log(k/\#emptyBuckets)$ instead.

In summary, the cardinality estimation building block can work for a wide range of packet property sets with low error and moderate resources, while providing a configurable tradeoff between accuracy and resource use.

Approximating Per-Flow Statistics: Resource limits prevent maintaining accurate per-flow state, so this class of building blocks approximates flow statistics. We discuss two building blocks. The first building block provides per-flow counters. It can be used, for example, to track the number of bytes transmitted by individual flows to identify elephant flows. The second building block tracks timestamp values for a particular flow, e.g. to detect when a flow was last seen for flowlet detection.

Per-flow Counters: For this building block we use a count-min sketch [16] to track element counts in sub-linear space. It supports two operations: `update(e,n)` adds `n` to the counter for element `e` and `read(e)` reads the counter for element `e`. This building block requires a table of r rows and c columns, and r independent hash functions. `update(e,n)` applies the r hash functions to `e` to locate a separate cell in each row, and then adds `n` to each of those r cells. `read(e)` in turn uses the r hash functions to find the same r cells for `e` and returns the minimum. The approximation of the count returned by `read` is always greater than or equal to the exact count. To keep the sketch from saturating, we use the switch CPU to periodically multiply all values in the counter table by a factor < 1 (i.e., decay the counts across multiple intervals), or reset the whole table to 0, depending on the use-case.

N	False-positive	r	c	$r \times c$
1,000	10%	4	512	2,048
	5%	3	1,024	3,072
	1%	3	2,048	6,144
10,000	10%	2	8,192	16,384
	5%	3	8,192	24,576
	1%	4	16,384	65,536
100,000	10%	3	65,536	196,608
	5%	2	131,072	262,144
	1%	4	131,072	524,288

Table 1: Required count-min sketch size to achieve false-positive rate below the specified threshold. The workload consists of N flows that send a total of 10 M packets, 80% of which belong to 20% of the flows—the heavy hitters (HH)— and the HH-threshold is set based on the expected number of packets sent from a HH in this scenario.

Per-Flow Timestamps: To track the timestamp of the last received packet along a flow, we slightly modify the use of the count-min sketch to derive a *min-timestamp* sketch. Instead of adding `n` (the current time) to the selected cells, `update(e,n)` now just overwrites these cells with `n`. `read` remains unmodified and as such returns a conservative estimate for the last time an element has been seen.

One use-case for this block is flowlet switching, wherein packets of a live flowlet should all be routed along the same route but new flowlets can be assigned a new route. Flowlet switching requires two pieces of information per flow: the timestamp of the previous packet, and the assigned route. Our timestamp building block tracks flow timestamps, but storing the route information requires an extension of the building block.

The extended building block supports the following two operations: `update(e,n,r)` sets the timestamp of element `e` to `n` and, if the flowlet timestamp had expired, sets the route to `r`; and `read(e)` returns both the last timestamp and the route for element `e`. We assume that the chosen route for a flow can be represented as a small integer that represents its index in a deterministic set of candidate routes for the flow. Both `update` and `read` still use the same mechanism for finding r cells for a particular `e`, but we extend those cells to store both the timestamp and a route integer. `read` returns the minimum of the timestamps and *the sum* of the route integers from the cells. `update` still updates the timestamps in all selected cells, but will only modify the route information in cells where the timestamp is older than the con-

figured flowlet timeout. If no cell has a timestamp older than the timeout, then the flowlet is considered live and thus the route is not modified. Any non-live flowlet will have at least one cell that has a timestamp older than the timeout, thus the route values stored in these cells can be modified so that the sum of all route values equals the specified `r`. (The pseudocode for this building block is shown in Figure 3.1 in the Appendix.) The key observation for correctness is that the route value in a cell with a live timestamp will never be modified, which guarantees that none of the route information for a live flowlet is modified, because the flowlet's cells will all have live timestamps.

3.2 Approximating Arithmetic

We design a class of building blocks that provide an exact implementation of complex arithmetic where possible and approximations for the general case. We can exactly compute multiplication and division with a constant operand using bit shifts and addition. We resort to approximation only when operands cannot be represented as a short sequence of additions. Our approximate arithmetic supports two variable operands and relies on lookup tables and addition.

Bit shifts: Multiplication and division by powers of 2 can be implemented as bit shifts. If the constant operand is not a power of 2, adding or subtracting the results from multiple bit shifts provides the correct result, e.g., $A \times 6$ can be rewritten as $A \times 2 + A \times 4$. Division can be implemented by rewriting as multiplication with the inverse. Resource constraints (such as the number of pipeline stages) limit the number of shifts and additions that can be executed for a given packet.

Logarithm lookup tables: Where multiplication and division cannot be carried out exactly (e.g., if both operand are variables), we use the fact that $A \times B = \exp(\log A + \log B)$. Given log and exp, we can reduce multiplication and division to addition and subtraction. Because FlexSwitches do not provide log and exp natively, we approximate them using lookup tables.

We use a ternary match table to implement logarithms. For N-bit numbers and a window of calculation accuracy of size m, with $1 \leq m \leq N$, table entries match all bitstrings of the form:

$$0^n 1 (0|1)^{min(m-1, N-n-1)} x^{max(0, N-n-m)} \qquad 0 \leq n < N$$

where x is the wildcard symbol. These entries map to the l-bit log value—represented as a fixed point integer—of the average number covered by the match. For example, for $N = 3$ and $m = 1$, the entries are $\{001, 01x, 1xx\}$, and for $m = 2$ the table entries are $\{001, 010, 011, 10x, 11x\}$. `exp` is calculated using an exact match table that maps logarithms back to the corresponding integer value. The parameters m and l control the space/accuracy tradeoff for this building block. For N-bit operands, table size is approximately $N \times 2^m$ for the log table and precisely 2^l for the exp table. Table 2 shows the minimal values of m and l to achieve a mean relative error below the specified thresholds. Note that even for 64-bit numbers a mean relative error below 1% can be achieved within 2048 exact match and ternary match table entries.

N	Error	m	l	exp (SRAM)	log (TCAM)
16	10%	3	6	64	59
	5%	4	7	128	111
	1%	6	9	512	383
32	10%	3	7	128	123
	5%	4	8	256	239
	1%	6	10	1024	895
64	10%	3	9	512	251
	5%	4	9	512	495
	1%	6	11	2048	1919

Table 2: Required number of lookup table entries for an approximation of multiplication/division for different size operands (N in bits) for the smallest configuration (m and l) with a mean relative error below the specified threshold.

3.3 Switch Statistics

Metering queue lengths: Network protocols often require information about queue lengths. FlexSwitches provide queue length information as part of the packet metadata, but only in the egress pipeline and only for the queue just traversed. This building block tracks queue lengths and makes them available through the entire processing pipeline. When a packet arrives in the egress pipeline we record the length of queue traversed in a register for that queue. Depending on the use-case, this building block can be configured to track the minimal queue length seen in a specified time interval, and using a timer to reset the register at the end of every interval. A variant of this building block is to calculate a continuous exponentially weighted moving average (EWMA) of queue lengths, and this utilizes the approximate arithmetic building block to perform the weighting.

Metering Rates: Similarly, data rates are an important metric in resource allocation schemes. FlexSwitches provide metering functionality in hardware, but the output is generally limited to colors that are assigned based on thresholds as described in the P4 language. This building block is able to measure rates for arbitrary input events in a more general fashion. We provide two configurations: One measures within time intervals, the other measures continuously. We describe the latter as the former is straightforward.

In order to support continuous measurement of the rate during the last time interval T, we use two registers to store the rate and the timestamp of the last update. When updating the estimate, we first calculate the time passed

since the last update as t_Δ. Because there were no events during the last t_Δ ticks, we can calculate how many events occurred during the last time interval T based on the previous rate R as $Y = R \times (T - t_\Delta)$ (or 0 if $t_\Delta > T$). Based on that and the number of events x to be added with the current operation we update the rate to $\frac{Y+x}{T}$, and also update the timestamp register with the current timestamp. The multiplication with R for calculating Y is implemented using the second method described in the approximate multiplication building block, while the division by T can be implemented using a right shift.

Approximate Port Balancing: A common task is to assign packets to one of multiple destinations (e.g., links or servers) to balance load between them. We provide a class of building blocks that implement different balancing mechanisms. The building blocks in this class fall in two separate categories, static balancing steers obliviously to the current load while dynamic balancing takes load into account. We describe the latter.

Making a load balancing decision while taking into account the current load avoids the imbalances common with static load balancing schemes. Computational limitations on FlexSwitches make it infeasible to pick the least loaded destination from a set of candidates larger than 2-4. Previous work [8] has shown that picking the least loaded destination from a small subset – even just 2 – of destinations chosen at random, significantly reduces load imbalances. Picking random candidates can be implemented on FlexSwitches by calculating two or more hashes on high-entropy inputs (timestamp, or other metadata fields). Information about the load of different destinations can be obtained from the *metering queue lengths* or *metering rates* building blocks.

3.4 Discussion

We note that our building blocks address both short and long-term limitations associated with FlexSwitches. For example, some of our building blocks emulate complex arithmetic using simpler primitives and help make the case for supporting some of these complex operations in future versions of FlexSwitches. The rest of the building blocks address fundamental constraints associated with switch state and operation count, thereby allowing the realization of a broader class of protocols that are robust to approximations (as we will see in the next section).

We provide the building blocks in a *template* form, parameterized by zero or more packet header fields. Each block either rewrites the packet header or maintains state variables that are available to subsequent building blocks in the pipeline. For example, the cardinality estimator can be parameterized by the 5-tuple or just the source address and exposes a single variable called 'cardinality'. This variable can be re-used when blocks are chained within a pipeline.

Tables 5 and 6 in Appendix A summarize our building blocks and the different classes of protocols we are able to support with them. We can realize a variety of schemes ranging from classical congestion control and scheduling to load balancing, QoS, and fairness. This shows our blocks are general enough to be reused across multiple protocols and sufficient to implement a broad class of resource allocation protocols.

4 Realizing Network Resource Allocation Protocols

In this section we describe and evaluate a number of network resource allocation protocols that can be built using the building blocks described in Section 3. The network protocols that we target fall into the following broad categories: congestion control, load balancing, QoS/Fairness, and IDS/Monitoring.

4.1 Evaluation Infrastructure

To show that we achieve our goal of implementing complex network resource allocation problems on limited hardware resources using only approximating building blocks, we first implement RCP on top of a real FlexSwitch. We use the Cavium Xpliant CNX880xx [14], a fully flexible switch that provides several of the features described in Section 2 while processing up to 3.2 Tb/s. We then evaluate the accuracy of our implementations versus the non-approximating originals using ns-3 simulations. Finally, we evaluate the resource usage of our implementations and discuss whether they fit within the limited resources that are expected of FlexSwitches.

To use the CNX880xx, we realize various building blocks on the Cavium hardware. This involved: 1) programming the parser to parse protocols, 2) creating tables for stateful memory, 3) configuring the pipeline to perform packet processing operations, and 4) coding the service CPU to perform periodic book-keeping.

To measure the performance of our RCP implementation against other protocols, we emulate a 2-level Fat-Tree topology consisting of 8 servers, 4 ToRs and 2 core switches by creating appropriate VLANs on the switch and directly interconnect the corresponding switch ports. This way, the same switch emulates all switches in the topology. All links operate at 10Gbps. We generate flows from all servers towards a designated target server with Poisson arrivals such that the ToR links are 50-60% utilized. The flows are Pareto distributed ($\alpha = 1.2$), with a mean size of 25 packets. We measure the flow completion times and compare it with the default Linux TCP-cubic implementation.

To evaluate the accuracy of our use cases, we implement them within the ns-3 network simulator [25] version 3.23. We simulate a datacenter network topology consisting of 2560 servers and a total of 112 switches.

The switches are configured in a 3-level FatTree topology with a total over-subscription ratio of 1:4. The core and aggregation switches each have 16×10Gbps ports while the ToRs each have 4×10Gbps ports and 40×1Gbps ports. All servers follow an on-off traffic pattern, sending every new flow to a random server in the datacenter at a rate such that the core link utilization is approximately 30%. Flows are generated using a Pareto distribution ($\alpha = 1.2$), and a mean flow size of 25 packets. Simulations are run for long enough that flow properties can be evaluated.

To evaluate the resource use of our use cases on an actual FlexSwitch, we implement our use cases in the P4 programming language and compile them to a production switch target. The compiler implements the functionality proposed in [19] and compiles to the hardware model described in Section 2. It reports the hardware resource usage for the entire use case, including memory used for data and code.

Our compiler-based evaluation serves to quantify the increased resource usage of our congestion control implementations when added to a baseline switch implementation based upon [34] that provides common functionality of today's datacenter switches. The baseline switch implementation provides basic L2 switching (flooding, learning, and STP), basic L3 routing (IPv4, IPv6, and VRF), link aggregation groups (LAGs), ECMP routing, VXLAN, NVGRE, Geneve and GRE tunneling, and basic statistics collection. We intentionally do not add more functionality to the baseline to highlight the additional resources consumed by our implementations.

4.2 Simple use-cases

The simple use-cases typically apply building blocks in a direct way to achieve their goals. We provide here a few examples to give insight into how our building blocks apply to a wide variety of different network applications.

WCMP. Weighted Cost Multipath (WCMP) routing [39] is an improvement over ECMP that can balance traffic even when underlying network performance is not symmetric. WCMP uses *weights* to express path preferences that consequently impact load balance. To implement WCMP using FlexSwitches, we use the approximate port balancing building block over a next-hop hash table and replicate next-hop entries according to their weight. The WCMP weight reduction algorithm [39] applies in the same way to reduce table entries.

CoDel. This QoS mechanism [24] monitors the minimum observed per-packet queueing delay during a time interval and drops the very last packet in the interval if this minimum observed delay is above a threshold. If there is a packet drop, the scheme provides a formula for calculating a shorter interval for the next time period. This scheme can be easily implemented on a FlexSwitch using metering, a small amount of state for maintaining the time period, approximate arithmetic for computing the next interval, and timers.

TCP port scan detection. To detect TCP port scans, we filter packets for set SYN flags and use the cardinality estimation building block to estimate the number of distinct port numbers observed. If this estimate exceeds a set threshold, we report a scan event to the control plane.

NTP amplification attack detection. Similarly, to detect NTP amplification attacks, we detect NTP packets (UDP port 123) and use cardinality estimation of distinct source IP addresses. If the number is high, we conclude that a large number of senders is emitting NTP requests over a small window of time and report an attack event.

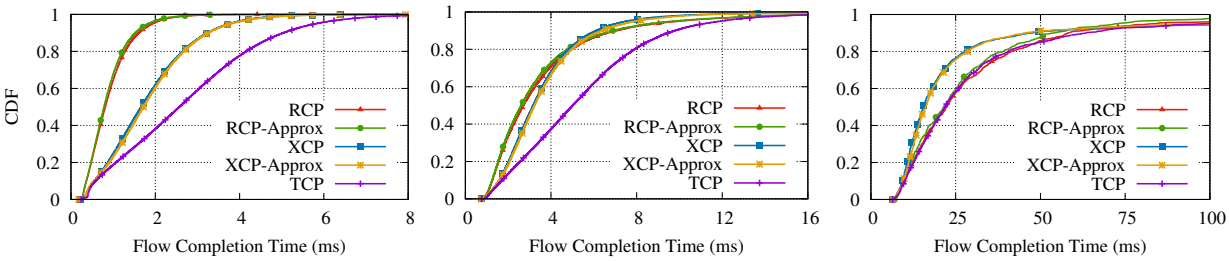
4.3 RCP

We now illustrate how to orchestrate several of the building blocks described in Section 3 to implement RCP. Staying true to our design principles, we employ approximation when necessary to make the implementation possible within limited resource bounds. We recall: To implement RCP we require a way of metering average utilization and queueing delay over a specified control interval. We also need to determine the number of active flows for each outgoing port over the same interval.

First, we estimate the spare capacity and the persistent queues built up inside the switch using the *metering utilization and queueing* building block. We use the building block to maintain per-link meta-data for the number of bytes received and the minimum queue length observed during a control period. When a packet arrives, we update Q by taking the minimum of the previous Q value and the current queue occupancy size as measured by our building block. Similarly, a counter B accumulates the number of bytes sent over the link. A timer is initialized to d, the expected steady state RTT of most flows. When the timer expires, we calculate the spare bandwidth as $S = C - B/d$, where C is the total capacity of the link. d is rounded down to a power of two so that the division operation can be replaced by a bit-shift operation. This use of a slightly smaller d is consistent with RCP's recommendation of using control periods that are roughly comparable to the average RTT.

RCP approximates the number of flows using a circular definition: $\hat{N}(t) = \frac{C}{R(t-d)}$. This essentially estimates the number of flows using the rate computed in the previous control interval. We use a more direct approximation of the number of unique flows by employing the *cardinality estimator* building block.

Finally, with estimates of utilization, queueing, and number of flows, we can use the RCP formula for calculating $R(t)$. Given that RCP is stable for a wide range of $\alpha, \beta > 0$, we pick fractional values that can be approximated by bit shift operations. For the division by $N(t)$, a



(a) Flows of size < 50 packets (b) Flows of size 50-500 packets (c) Flows of size > 500 packets

Figure 3: Cumulative distribution of FCTs of various flow sizes for TCP as well as both precise and approximate RCP and XCP.

general division is required because both sides are variables. We use the *approximate divider* building block to perform the division with sufficient accuracy.

Our hardware implementation of RCP uses several features of the XPliant CNX880xx switch. First, we use the configurable counters to build the cardinality estimator. An array of counters indexed by the packet hash is incremented on each packet arrival. The counter values are read periodically to estimate the number of ongoing flows. Next, we use the metering block to maintain port level statistics such as bytes transmitted and queue lengths. For periodically computing RCP rates, we utilize an on-switch service CPU to compute and store the rates inside the pipeline lookup memory. Finally, we program the reconfigurable pipeline to correctly identify and parse an RCP packet, extract the rate and rewrite it with the switch rate if it is lower.

We compared the performance of the above RCP implementation against TCP on the Cavium testbed and workload described in Section 4.1. We measure the flow completion times for various flow sizes and report them in Table 3. As expected, RCP benefits significantly from the switch notifying the endhosts the precise rate to transmit at. This avoids the need for slow-start and keeps the queue occupancy low at the switches, leading to lower flow completion times.

In order to measure the impact of approximation on the accuracy of our FlexSwitch implementation of RCP, we compare our implementation (RCP-Approx) to an original implementation (RCP) using the simulated workload described in Section 4.1. Figure 3 shows the result as a number of CDFs of flow completion time (FCT) for flows of various lengths. We can see that RCP-Approx matches the performance of RCP closely, for all three types of traffic flows. We also compare to the performance of TCP to validate that our simulation adequately models the performance of RCP. We see that this is indeed the case as RCP performance exceeds TCP performance for shorter flows, as shown in [17].

To measure the additional hardware resources used for RCP-Approx, we integrate RCP-Approx into our baseline switch implementation and compile using the compiler described in Section 4.1. Table 4 shows the additional hardware resources used compared to the baseline switch. We can see that additional resource use is small—not exceeding 6% for all resources but pipeline stages and requiring an additional stage.

Flow Size	**TCP** Mean	50^{th}%	95^{th}%	**RCP** Mean	50^{th}%	95^{th}%
Short	10.32	0.85	2.92	0.85	0.73	2.05
Medium	67.97	5.33	216.99	5.07	3.18	14.85
Long	649.25	59.73	3559.85	50.42	36.85	137.26

Table 3: Flow completion times for short, medium, and long flows (< 50, < 500, and ≥ 500 packets) in milli-seconds on a two-tier topology running RCP on Cavium hardware.

Resource	**Baseline**	**+RCP**		**+XCP**		**+CONGA**	
Pkt Hdr Vector	187	191	+2%	195	+4%	199	+6%
Pipeline Stages	9	10	+11%	9	+0%	11	+22%
Match Crossbar	462	473	+2%	471	+2%	478	+3%
Hash Bits	1050	1115	+6%	1058	+1%	1137	+8%
SRAM	165	175	+6%	172	+4%	213	+29%
TCAM	43	44	+2%	45	+5%	44	+2%
ALU Instruction	83	88	+6%	92	+11%	98	+18%

Table 4: Summary of resource usage for various use-cases.

We conclude that RCP can indeed be implemented with adequate accuracy and limited additional resource usage. This gives us confidence that other resource allocation algorithms might be implementable as well and we do so in the following subsections.

4.4 XCP

Like RCP, the eXplicit Control Protocol (XCP) [20] is a congestion control system that relies on explicit feedback from the network, but optimizes fairness and efficiency over high bandwidth-delay links. An XCP-capable router maintains two control algorithms that are executed periodically on each output port: a congestion controller and a fairness controller. The congestion controller is similar to RCP's—it computes the desired increase or decrease in the aggregate traffic (in bytes) over the next control interval as $\phi = \alpha \cdot d \cdot S - \beta \cdot Q$, where S, Q, and d are defined as before.

The fairness controller distributes the aggregate feedback ϕ among individual packets to achieve per-flow fairness. It uses the same additive increase, multiplica-

tive decrease (AIMD) [7] principle as TCP. If $\phi > 0$, it increases the throughput of all flows by the same uniform amount. If $\phi < 0$, it decreases the throughput of a flow by a value proportional to the flow's current throughput. XCP achieves AIMD control without requiring per-flow state by sending feedback in terms of change in congestion window, and through a formulation of the feedback values designed to normalize feedback to account for variations in flow rates, packet sizes, and RTTs.

In particular, given a packet i of size s_i corresponding to a flow with current congestion window $cwnd_i$ and RTT rtt_i, XCP computes the positive feedback p_i (when $\phi > 0$) and the negative feedback n_i (when $\phi < 0$) as:

$$p_i = \xi_p \cdot \frac{rtt_i^2 \cdot s_i}{cwnd_i} \qquad n_i = \xi_n \cdot rtt_i \cdot s_i$$

where ξ_p and ξ_n are constants for a given control interval. Observe that the negative feedback is simply a uniform constant across packets if all flows have the same RTT and send the same sized packets. As a consequence, flows that send more packets (and hence operate at a higher rate) will get a proportionally higher negative feedback and will multiplicatively decrease their sending rates in the next control interval. Similarly, the structure of p_i results in an additive increase as the per-packet feedback is inversely proportional to $cwnd_i$. Finally, the per-interval constants ξ_p and ξ_n are computed such that the aggregate feedback provided across all packets equals ϕ, with L being the set of packets seen by the router in the control interval:

$$\xi_p = \frac{\phi}{d \cdot \sum_L \frac{rtt_i \cdot s_i}{cwnd_i}} \qquad \xi_n = \frac{-\phi}{d \cdot \sum_L s_i}$$

FlexSwitch Implementation

The core of the XCP protocol requires each switch to: (i) calculate and store at every control interval the values ϕ, ξ_p, and ξ_n, and (ii) compute for every packet the positive or negative feedback values (p_i or n_i). p_i and n_i are communicated back to the sender, while $cwnd_i$ and rtt_i are communicated to other routers to allow them to compute ξ_p and ξ_n. Given the programmable parser in a FlexSwitch, it is straightforward to extend the packet header to include fields corresponding to $cwnd_i$, rtt_i, and either p_i or n_i. However, the XCP equations described above require complex calculations. We outline a sequence of refinements designed to address this.

Approximate computations: As with RCP, we make use of the *metering utilization and queueing* building block and then suitably choose the stability constants and the control interval period to simplify computations in the processing pipeline. First, we set the stability constants α, β to be negative powers of 2, such that ϕ can be calculated using bit shifts. XCP is stable for $0 < \alpha < \frac{\pi}{4\sqrt{2}}$ and $\beta = \alpha^2\sqrt{2}$, which makes this simplification feasible.

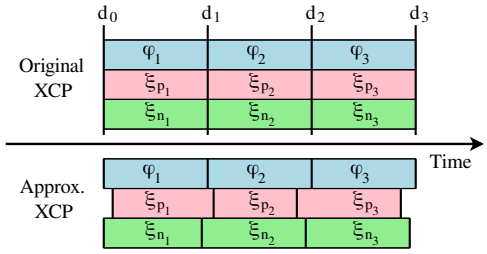


Figure 4: Staggered evaluation of XCP control parameters.

Next we approximate the control interval d to be a power of two that approximates the average RTT in the datacenter. We can then compute ϕ/d every d microseconds using integer counters and bit-shifts without incurring the loss in precision associated with the approximate division building block.

End-host computations: In XCP, end-hosts send rtt_i and $cwnd_i$ values with every packet and receive from the switches a per-packet feedback p_i, n_i. In our FlexSwitch implementation, we offload more of the computation to the end-hosts. First, we require the end-host to send to switches the computed value $\frac{rtt_i \cdot s_i}{cwnd_i}$. Second, instead of calculating the per-packet feedback p_i, n_i at the switch, we simply send the positive and negative feedback coefficients ξ_p, ξ_n to the end-host. The end-host can then calculate the necessary feedback by computing p_i, n_i based on its local flow state. This results in a subtle approximation as we can no longer keep track of aggregate feedback at the router. It is possible that we give out more aggregate feedback than the target ϕ if a large burst of packets arrives during a control interval. This can temporarily cause a queue buildup inside the switch, but XCP's negative feedback will quickly dissipate the queue. We did not see large queue buildups in our simulations.

Approximate control intervals: Finally, instead of computing the positive and negative feedback coefficients ξ_p, ξ_n along with ϕ at the end of every control interval, we compute them whenever the denominator reaches a value close to a power of 2. In our case, we approximate $\sum_L \frac{rtt_i \cdot s_i}{cwnd_i}$ or $\sum_L s_i$ to a power of 2 for positive and negative feedback coefficients respectively, both of which are in the XCP congestion header and are accumulated in integer counters inside switch memory. As a result, calculating ξ_p, ξ_n requires only a bit-shift operation. It also means that we are not calculating all XCP parameters synchronously at every control interval, but rather at staggered intervals as shown in Figure 4.

We again measure the impact of approximation on the accuracy of our FlexSwitch implementation by comparing it (XCP-Approx) to an original implementation of XCP using the simulated workload described in Section 4.1. From Figure 3 we can see that XCP-Approx also closely matches the performance of XCP, for all three types of traffic flows. Both implementations out-

perform RCP for long flows and perform worse than RCP for short flows. This validates our simulation as XCP is optimized for long flows over high bandwidth-delay links, while RCP is optimized for short flows.

Table 4 shows the additional hardware resources used by XCP-Approx when integrated into the baseline switch. XCP requires almost twice the computational resources of RCP—both in terms of ALU instructions and state carried among pipeline stages in the packet header vector. This is expected, as XCP-Approx computes 2 counter values for every packet and 3 parameters every control interval, while RCP carries out only 1 computation per interval and 1 per packet arrival. Conversely, SRAM use is diminished versus RCP, as we can carry out multiplication/division solely via bit-shifts, while we require more TCAM entries to identify when a variable is an approximate power of 2.

We conclude that XCP can also be implemented with adequate accuracy and limited additional resource usage using our building blocks. Given this experience, we now turn to a slightly broader resource management algorithm that combines some of the functionality required for RCP and XCP with a load balancing element.

4.5 CONGA

CONGA [3] is a congestion-aware load balancing system that splits TCP flows into flowlets and allocates them to paths based on congestion feedback from remote switches. Congestion is communicated among switches with each payload packet by embedding congestion information within unused bits of the VXLAN [21] overlay network header that is common in datacenters today.

CONGA operates primarily at leaf switches and does load balancing based on per-uplink congestion. Each leaf switch holds two tables that contain congestion information along all possible uplink choices from and to other leaf switches and are updated by information from these other leaf switches. To forward a packet, a leaf switch picks an outgoing link and records its choice in the packet header. Core switches record the maximum congestion along the path to the packet's destination by updating a header field with the maximum of their local congestion and that already in the field. Finally, the destination leaf switch updates its congestion-from-leaf table according to the recorded congestion along the sender's uplink port choice. To relay congestion information back to sender switches, each switch additionally chooses one entry in its congestion-from-leaf table for that switch and transmits it using another set of CONGA header fields within each payload packet.

To estimate local link congestion, all switches use a discounting rate estimator (DRE). DREs use a single register X to keep track of bytes sent along a link over a window of time. For each sent packet, the register is incremented by that packet's size. The register is decremented periodically with a multiplicative factor α between 0 and 1: $X \leftarrow X \times (1-\alpha)$.

CONGA's load-balancing algorithm is triggered for each new flowlet. It first computes for each uplink the maximum of the locally measured congestion of the uplink and the congestion feedback received for the path from the destination switch. It then chooses the uplink with the minimum congestion for the flowlet.

To recognize TCP flowlets, leaf switches keep flowlet tables. Each table entry contains an uplink number, a valid bit, and an age bit. For each incoming packet, the corresponding flowlet is determined via a hash on the packet's connection 5-tuple that indexes into the table. If the flowlet is valid (valid bit set), we simply forward the packet on the recorded port. If it is not valid, we set the valid bit and determine the uplink port via the load-balancing algorithm. Each incoming packet additionally resets the age bit of its corresponding flowlet table entry. A timer periodically checks the age bit of all flowlets before setting it. If a timer encounters an already set age bit, then the flowlet is timed out by resetting the valid bit. To accurately detect enough flowlets, CONGA switches have on the order of 64K flowlet entries in a table.

FlexSwitch Implementation

Relaying CONGA congestion information is straightforward in FlexSwitches: We simply add the required fields to the VXLAN headers. Since CONGA is intended to scale only within 2-layer cluster sub-topologies, congestion tables are also small enough to be stored entirely within FlexSwitches. To implement the remaining bits of CONGA, we need the following building blocks:

Measuring Sending Rates: We need to measure the rate of bytes sent on a particular port, which CONGA implements with DREs. We use the timer-based rate measurement building block for this task. We setup one block for each egress port to track the number of transmitted bytes along that port and set the timeout and multiplicative decrease factor to CONGA-specific values.

Maintaining congestion information: We can simply fix the size of congestion tables to a power of 2 and use precise multiplication via bit shifts and addition to index into the 2-dimensional congestion tables.

Flowlet detection: To identify TCP flowlets and remember their associated paths we use the flow statistics building block. We mimic CONGA's flowlet switching without a need for valid or age bits.

CONGA flowlet switching: To remember the initially chosen uplink for each flowlet, we store the uplink number as a tag inside the high-order bits of the flow's associated counter. We use this tag directly for routing. At the same time, we ensure that per-flow counters are reset fre-

quently enough so that the tag value will never be overwritten due to the flow packet counter growing too large. To do so, we simply calculate the maximum number of min-sized packets that can traverse an uplink within a given time frame and ensure that this number stays below the allocated counter size (32 bits), minus the space reserved for the tag (2 bits in current ToR switches). For 100Gb/s uplinks, more than 5 seconds may pass sending min-sized packets at line-rate before counter overflow.

CONGA's load balancing requires a complex calculation of minimums and maximums—too many to be realized efficiently on a FlexSwitch. Rather than faithfully replicating the algorithm and determining the best uplink port upon each new flowlet, we keep a running tally of the minimally-congested uplink choice for each leaf switch. Upon a new flowlet, we simply forward along the corresponding minimally-congested uplink.

Our running tally needs to be updated each time a corresponding congestion metric is updated. If this results in a new minimum, we simply update the running minimum. However, if an update causes our current minimum to fall out of favor, we might need to find the new current minimum. This would require re-computation of the current congestion metric for all possibilities—an operation we want to avoid. Instead, we simply update our current "minimum" to the value we have at hand as a best guess and wait for further congestion updates from remote switches to tell us the true new minimum. In our current implementation we also do not update our tally when local rate measurements are updated. This spares further resources at minimal accuracy loss. We ran ns3 simulations comparing CONGA and approximated CONGA implemented using our building blocks to confirm that our approximations don't cause significant deviation in performance (see Appendix B).

Table 4 shows the resources used when our CONGA implementation is added to our baseline switch implementation. We can see that an additional 29% of SRAM to store the additional congestion and flowlet tables is the main contribution. Also, we require 2 extra pipeline stages and 18% more ALU instructions to realize CONGA computationally, for example to approximate multiplication and division. Other resource use is increased minimally, below 10%.

We conclude that even complex load balancing protocols can be implemented on FlexSwitches using our building blocks. The additional resource use is moderate, taking up less than a third of the baseline amount of stages, SRAM bytes, and ALU instructions and less than 10% of the baseline for the other switch resources.

5 Related Work

Our work demonstrates that many resource allocations protocols can be implemented on a FlexSwitch using various approximation techniques. To achieve our goal, we leverage several algorithms from the streaming literature and apply them to a switch setting. For example, we show how to implement the HyperLogLog [28] algorithm and count-min sketch [16] on a FlexSwitch to approximate the number and frequency of distinct elements in traffic flows. Our flow timestamps and flowlet detection building blocks are related to *Approximate Concurrent State Machines* [11], but we are able to design simpler solutions given that we don't need general state machines to implement the functionality. Other related efforts are OpenSketch [38] and DREAM [23] that propose software-defined measurement architectures. OpenSketch uses sketches implemented on top of NetFPGA, while DREAM centralizes the heavy-hitter detection at a controller while distributing the flow monitoring tasks over multiple traditional OpenFlow switches. Both works trade off accuracy for resource conservation. We build on these ideas to implement a broad set of building blocks within the constraints of the hardware model and implement resource allocation algorithms using them.

Other work has proposed building blocks to aid programmable packet scheduling [35] and switch queue management [36]. While some of this work has been implemented on NetFPGA, we believe their solutions are likely to be applicable within a FlexSwitch model albeit with some approximations. Complementary to our work are proposals for enhancing the programmability of a reconfigurable pipeline [33].

6 Conclusion

As switching technology evolves to provide flexible M+A processing for each forwarded packet, it holds the promise of making a software-defined dataplane a reality. This paper evaluates the power of flexible M+A processing for in-network resource allocation. While hardware constraints make the implementation of resource allocation algorithms difficult, we show that it is possible to approximate several popular algorithms with acceptable accuracy. We develop a number of building blocks that are broadly applicable to a wide range of network allocation and management applications. We use implementations on a production FlexSwitch, compilation statistics regarding hardware resources allocated, and simulation results to show that our protocol implementations are feasible and effective.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Rodrigo Fonseca for their valuable feedback. This research was partially supported by the National Science Foundation under Grants CNS-1518702 and CNS-1616774.

References

[1] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010).

[2] ALIZADEH, M., ATIKOGLU, B., KABBANI, A., LAKSHMIKANTHA, A., PAN, R., PRABHAKAR, B., AND SEAMAN, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE standardization. In *Proceedings of the 46th Annual Allerton Conference on Communication, Control, and Computing* (2008).

[3] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[4] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference* (2010).

[5] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less Is More: Trading a Little Bandwidth for ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012).

[6] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[7] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681, 2009.

[8] AZAR, Y., BRODER, A. Z., KARLIN, A. R., AND UPFAL, E. Balanced Allocations. *SIAM Journal on Computing 29*, 1 (1999), 180–200.

[9] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *12th USENIX Symposium on Networked Systems Design and Implementation* (2015).

[10] BAREFOOT NETWORKS. Tofino Programmable Switch. `https://www.barefootnetworks.com/technology/`.

[11] BONOMI, F., MITZENMACHER, M., PANIGRAH, R., SINGH, S., AND VARGHESE, G. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *Proceedings of the ACM SIGCOMM Conference* (2006).

[12] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review 44*, 3 (July 2014), 87–95.

[13] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[14] CAVIUM. CNX880XX_PB_p1 Rev1 - Cavium. `http://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf`.

[15] CAVIUM. XPliant Ethernet Switch Product Family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[16] CORMODE, G., AND MUTHUKRISHNAN, S. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[17] DUKKIPATI, N. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2007.

[18] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J., AND AKELLA, A. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[19] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015).

[20] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the ACM SIGCOMM Conference* (2002).

[21] MAHALINGAM, M., DUTT, D., DUDA, K., AGARWAL, P., KREEGER, L., SRIDHAR, T., BURSELL, M., AND WRIGHT, C. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, 2014.

[22] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review 38*, 2 (Mar. 2008), 69–74.

[23] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[24] NICHOLS, K., AND JACOBSON, V. Controlling Queue Delay. *Queue 10*, 5 (May 2012).

[25] NS-3 CONSORTIUM. ns-3 Network Simulator. `http://www.nsnam.org/`.

[26] OZDAG, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[27] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[28] PHILIPPE FLAJOLET AND ÉRIC FUSY AND OLIVIER GANDOUET AND FRÉDÉRIC MEUNIER. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *HAL CCSD; Discrete Mathematics and Theoretical Computer Science* (June 2007), 137–156.

[29] POPA, L., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the Network in Cloud Computing. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011).

[30] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (1999).

[31] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[32] SHIEH, A., KANDULA, S., GREENBERG, A., KIM, C., AND SAHA, B. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011).

[33] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[34] SIVARAMAN, A., KIM, C., KRISHNAMOORTHY, R., DIXIT, A., AND BUDIU, M. DC.P4: Programming the Forwarding Plane of a Data-center Switch. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research* (2015).

[35] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[36] SIVARAMAN, A., WINSTEIN, K., SUBRAMANIAN, S., AND BALAKRISHNAN, H. No Silver Bullet: Extending SDN to the Data Plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013).

[37] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM Conference* (2011).

[38] YU, M., JOSE, L., AND MIAO, R. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013).

[39] ZHOU, J., TEWARI, M., ZHU, M., KABBANI, A., POUTIEVSKI, L., SINGH, A., AND VAHDAT, A. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems* (2014).

A Building Blocks and their Use in Various Protocols

Building Block	Functionality	Techniques
Cardinality Estimator	Estimate #unique elements in a stream	Linear Counting, Hyperloglog
Flow Counters	Estimate per-flow bytes/packets	Count-min sketch
Flow Timestamps	Last packet sent timestamp	Sketch with timestamps
Metering queues/utilization	Estimate rates	EWMA
Port Balancing	Pick a port based on some metric	Power-of-2 choices
Approx Arithmetic	Division/Multiplication	Logarithm tables, Bit-shits

Table 5: Summary of the proposed building blocks and techniques we use to implement them.

Functionality	Protocol	Building Blocks Required	Implementation
Congestion Control, Scheduling	RCP [17]	Arithmetic, Cardinality, Metering	Section 4.3.
	XCP [20]	Arithmetic, Metering	Section 4.4.
	QCN [2]	Arithmetic, Metering	Meter queue sizes and calculate feedback value from switch queue length based on sampling probability.
	HULL [5]	Arithmetic, Metering	Implement *Phantom queues* by metering and mark packets based on utilization levels.
	D^3 [37]	Flow Statistics, Metering	RCP like feedback, but prioritizes near-deadline flows.
	PIAS [9]	Flow Statistics, Balancing	Emulates shortest job next scheduling by dynamically lowering priority based on packets sent.
Load Balancing	CONGA [3]	Arithmetic, Flow Timestamps, Metering	Section 4.5.
	WCMP [39]	Balancing	Section 4.2.
	Ananta [27]	Flow Counters, Metering, Balancing	Use flow counters to realize VIP map and flow table. Use metering and balancing for packet rate fairness and to detect heavy hitters.
	Hedera [1]	Flow Counters, Balancing	Detect heavy hitters and balance them.
	Presto [18]	Flow Statistics, Balancing	Create flowlets and balance them
QoS & Fairness	Seawall [32]	Arithmetic, Cardinality, Metering	Collect traffic statistics and provide RCP-like feedback to determine the per-link, per-entity share.
	FairCloud [29]	Arithmetic, Flow Counters	Meter number of source/destination flows and queue them into approximately prioritized queues.
	CoDel [24]	Arithmetic, Metering	Section 4.2.
	pFabric [6]	Arithmetic, Flow Counters	Queue packets into approximately prioritized queues using remaining flow length value in the header.
Access Control	Snort IDS [30]	Flow Counters, Cardinality	Section 4.2.
	OpenSketch [38]	Flow Counters, Metering	Our approximate flow statistics are based on the same underlying count-min sketch.

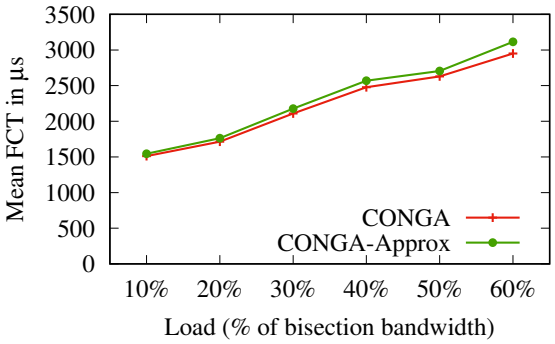
Table 6: Network functions that can be supported using FlexSwitch building blocks. A deeper discussion of several of the algorithms is given in Section 4.

```
sketch = {ts, route_id}[2][N]

elements(five_tuple):
  h1, h2 = hashes(five_tuple)
  e1 = sketch[0][h1 % N]
  e2 = sketch[1][h2 % N]

read(five_tuple):
  e1, e2 = elements(five_tuple)
  ts = min(e1.ts, e2.ts)
  route = e1.route_id + e2.route_id
  return (ts, route)
```

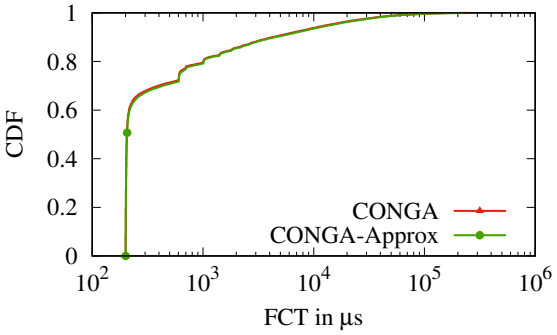
```
update(five_tuple, ts, route_id):
  e1, e2 = elements(five_tuple)
  cutoff = ts - TIMEOUT
  if (e1.ts < cutoff && e2.ts < cutoff)
    e1.route_id = rand()
    e2.route_id = route_id - e1.route_id
  else if (e1.ts < cutoff)
    e1.route_id = route_id - e2.route_id
  else if (e2.ts < cutoff)
    e2.route_id = route_id - e1.route_id

  e1.ts = e2.ts = ts
```

Figure 5: Pseudocode for the flowlet switching building block presented in Section 3.1. This particular sketch uses two hash functions and two rows of N counters each, but generalizes to multiple hash functions in a straightforward manner.

B Approximate CONGA ns3 simulation



(a) Average FCT with increasing network load.



(b) CDF of FCT with network load at 60%.

Figure 6: Performance comparison between CONGA and approximate CONGA using ns3 simulations.

We compare the performance of approximate CONGA implemented using our building blocks to the original version of CONGA. We simulate the same 4-switch (2 leaves, 2 spines), 64 server topology in [3] with $50\mu s$ link latency and the default CONGA parameters: $Q = 3, \tau = 160\mu s$, and flowlet gap $T_{fl} = 500\mu s$. We run the enterprise workload described in the same paper, and vary the arrival rate to achieve a target network load, which we measure as a percentage of the total bisection bandwidth.

First, we measure the change in average flow completion time as we increase the load in the network. Figure 6a shows that our approximate implementation closely follows the original protocol. We sometimes see a slightly higher FCT primarily because we perform an approximate *minimum* over the ports when we have to assign a new flowlet. Current restrictions on FlexSwitches don't allow us to scan all ports to pick the least loaded one, so we keep a running approximate minimum. This results in some flowlets not getting placed optimally to the least loaded link. In all other cases, we implement CONGA's protocols accurately.

Figure 6b shows the CDF of flow completion times for all flows when the network is loaded at 60%. Again, the approximate implementation of CONGA matches very closely to that of original CONGA. A majority of the flows are short and hence are not affected by the approximate minimum selection of ports.