In-Network Computation is a Dumb Idea Whose Time Has Come

Amedeo Sapio‡, Ibrahim Abdelaziz, Abdulla Aldilaijan,
Marco Canini, Panos Kalnis
KAUST

ABSTRACT

Programmable data plane hardware creates new opportunities for infusing intelligence into the network. This raises a fundamental question: what kinds of computation should be delegated to the network?

In this paper, we discuss the opportunities and challenges for co-designing data center distributed systems with their network layer. We believe that the time has finally come for offloading part of their computation to execute in-network. However, in-network computation tasks must be judiciously crafted to match the limitations of the network machine architecture of programmable devices. With the help of our experiments on machine learning and graph analytics workloads, we identify that aggregation functions raise opportunities to exploit the limited computation power of networking hardware to lessen network congestion and improve the overall application performance. Moreover, as a proof-of-concept, we propose DAIET, a system that performs in-network data aggregation. Experimental results with an initial prototype show a large data reduction ratio (86.9%-89.3%) and a similar decrease in the workers' computation time.

1 INTRODUCTION

The advent of flexible networking hardware [6] and expressive data plane programming languages [5, 29] have produced networks that are deeply programmable. The functionality of networks can now be enriched without hardware changes while retaining the capability of processing packets at very high rates, even above Terabits per second. Emerging programmable network devices are paving the way for new services to better support data center applications [9, 18] and improve network monitoring [13, 16, 24–26].

Programmable networks create the opportunity for in-network computation, i.e., offloading a set of compute operations from end hosts into network devices such as switches and smart NICs. In-network computation can offer substantial performance benefits, as it is for example the case with consensus protocols [9, 10] and in-network caches [20]. Although traditional networks are not capable of computation, the idea of using the network not just to move data, but also to perform computation on transmitted data is reminiscent of *Active Networks* [30], which proposed to replace packets with small programs called "capsules" that are executed at each traversed switch. However, for the past two decades the hardware capabilities were lacking. This appears to be changing.

The recently proposed RMT architecture [6] and its upcoming incarnation in the Barefoot Networks' Tofino [3] switch chip has a flexible parser and a customizable match-action engine. To process packets at high speed, this architecture has a multi-stage pipeline where packets flow at line rate. Each stage has a fixed amount of time to process every packet, allowing for lookups in memory (SRAM and TCAM), manipulating packet metadata and stateful registers, and performing boolean and arithmetic operations using ALUs. Other vendors are also introducing new classes of programmable chips with similar capabilities [7]. We believe that with this new generation of flexible data plane hardware it is worth revisiting a fundamental question: *as networks become capable of computation, what kinds of computation should networks perform?*

In this paper, we will consider this question in the scope of data center applications because it is likely that data centers will be early adopters of programmable networks and many of these applications have stringent performance requirements. On the one hand, in-network computations can be broadly useful in several performance-oriented contexts to reduce latency and/or increase throughput of certain operations. Furthermore, it can help reducing network traffic, so as to alleviate congestion, which is a major cause of application performance degradation. In particular, a computation that happens on-path and at line rate is appealing since it bears no cost to the application, which can spare CPU cycles for other tasks instead. On the other hand, despite recent technological advancements, network devices have limited compute power and little storage to support general computation. Moreover, systems designers are prescribed by the end-to-end principle [28] to avoid implementing application-specific logic in

‡Amedeo Sapio is also with Politecnico di Torino.

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA

ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152461

the network and are generally wary about raising the overall system complexity. Additionally, in-network computation must not affect the application correctness.

We posit that in-network computation must be used judiciously. Towards this goal, we seek to identify *what type of computation can be done in-network* such that: (*i*) network traffic is significantly reduced, (*ii*) only a minimal change at the application level is required and (*iii*) the correctness of the overall computation is not affected.

We find that a plausible class of applications that satisfy the above desiderata are those applications that follow a partition/aggregate workload pattern. These applications cover a wide spectrum of data-intensive frameworks including big data analytics as in MapReduce [12] and machine learning [1, 11], graph processing [14, 22] and stream processing [19]. Generally these frameworks scale applications by distributing data and computation across many worker servers. Each worker performs some computation on a data partition, which is followed by a communication phase to update shared state or finalize the computation. This process can be performed iteratively until a stopping condition is met.

These applications are sensitive to network performance and the communication cost can be one of the dominant scalability bottlenecks as large volume of data need to be moved routinely in many to many patterns. Already several distributed frameworks like MapReduce [12], Pregel [22] and DryadLINQ [32] allow for *user-defined aggregation functions*. These functions enable application developers to reduce the network load (e.g., by summing all individual messages to a common graph vertex) and consequently, the job execution time. However, the aggregation functions are only applied at the worker-level, missing the opportunity of achieving better traffic reduction ratios when applied at the network level.

These aggregation functions have several characteristics that make them appealing and suitable to be partially executed in-network. First, they usually *reduce the amount of data* (e.g., sum the inputs, or find the minimum). Thus, it is beneficial to apply these functions as early as possible to decrease the amount of network traffic and lessen congestion. Second, they are usually characterized by *simple arithmetic/logic operations*, which make them amenable to parallelization and execution on programmable switches. Third, in many algorithms [14, 22], they are *commutative and associative functions*, which implies that they can be applied separately on different portions of the input data, disregarding the order, without affecting the correctness of the final result. Fourth, they are *often readily available*, meaning that they could be transparently supported without requiring the developer to write new application logic. As we will show, implementing certain aggregation functions inside the network is possible and beneficial since programmable switches can aggregate intermediate data, thus reducing the traffic as well as the processing load at the destination. However, the limited resources, restricted compute power and stringent constraints on packet processing time create several challenges and call for a judicious system design.

To contribute a concrete point in the design space, we propose DAIET, a system for data aggregation in-network. While our design is still incomplete and likely to change, this represents an example of a system that can be built using P4 to offload computation to the data plane. Our experimental results with an initial prototype supporting a MapReduce application show that this approach provides a large data reduction (86.9%-89.3%) and a similar decrease in worker's computation time.

A number of recent research efforts [8, 15, 21] have proposed in-network aggregation techniques for a variety of applications. However, these systems either required to change the network architecture [8, 21] or build a switch chip with a fixed set of aggregation functions [15]. We demonstrate that similar benefits can be reaped using flexible and programmable data planes. That said, we envision that practical deployments for our proposal might be better suited within clusters and racks specialized for certain workloads such as deep learning or data analytics where the benefits of in-network aggregation are substantial without requiring data center-wide adoption.

We note that aggregation functions, though they are a generic primitive applicable to a number of applications, are not the sole type of in-network computation possible and we hope that our work will trigger a broader discussion around the driving question behind our work: what should networks compute?

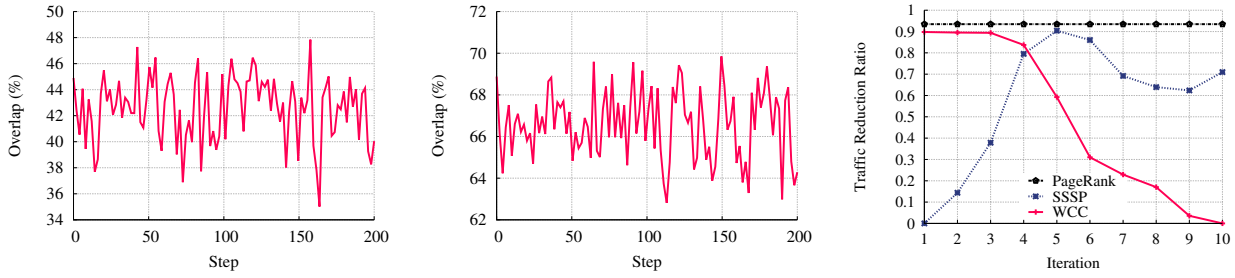
2 JUDICIOUS NETWORK COMPUTING

We focus on in-network computation enabled by the recent developments in reconfigurable, protocol-independent switch ASICs such as RMT [6]. Their network machine architecture is based on a multi-stage pipeline of packet processing logic [4]. Computing on these devices corresponds to executing streaming algorithms that have stringent constraints on the number and type of operations that can be performed. This is due to the following limitations:

Limited memory size. Packet processing at high speed requires a very fast memory, such as TCAMs or SRAM, which is expensive and usually available in small capacities. As an example, the upcoming Barefoot Tofino [3] switch chip is expected to process a remarkable 6.5 Tb/s while still providing the flexibility of data plane programmability. To match this processing speed, packets can be processed by a limited number of lookup tables and the expected available SRAM is in the range of few tens of MBs.

Limited set of actions. Programmable devices support a small set of actions, usually simple arithmetic, data manipulation, and hashing operations. Some switches can also provide limited support for floating point operations [15].

Few operations per packet. To guarantee execution at line rate, programs have only tens of nanoseconds to process a single packet. As a result, they cannot use constructs that do not have an upper bound on the number of performed operations (e.g., loops). Some devices allow to recirculate a packet



(a) Stochastic Gradient Descent (b) Adam optimization (c) Graph Analytics Algorithms

Figure 1: Potential traffic reduction ratio for two machine learning applications and various graph analytics algorithms.

in the ingress queue for further processing, thus allowing to implement loops. But this comes at the cost of additional processing latency and lowers the forwarding capacity.

Furthermore, offloading functionality to the network faces several challenges. First, the underlying target imposes restrictions as discussed above and resources are limited. As such, in-network computation must live within these confines. Second, applications correctness is paramount. Data center networks have multiple paths and failures of links and devices are not uncommon. As such, in-network computation should provide benefits even if traffic follows different paths or an application experiences failures. Alternately, an application should be no worse than without in-network computation even when this is executing. Third, offloading functionality raises complexity not only because certain packet processing logic is executed in the network but also due to the required integration with applications or libraries. As such, in-network computation should focus on primitives that are broadly applicable to a class of applications and workloads, and identify reusable, high-level abstractions that promote easy adoption.

Since we are just at the onset of programmable data plane hardware, it is hard to gauge how far in-network computation can go and what possibilities the next technological enhancements will enable. As an analogy, the spectrum of applications of GPUs has evolved significantly since when they first became programmable, and today's GPGPUs have usages for deep learning and mining cryptocurrencies, that are far beyond the computer graphics domain.

In the rest of this paper, we follow through our earlier premise and explore in-network aggregation as a concrete example of in-network computation. Consequently, a main challenge to address when delegating data aggregation to the network is to account for all the constraints above while providing high data reduction ratios without affecting the correctness of the final results.

3 DATA AGGREGATION IN DATA CENTER APPLICATIONS

Data aggregation is a common task in several distributed data center applications [1, 12, 22, 31]. It also satisfies the characteristics discussed in Section 1. Therefore, it represents a good candidate for tasks that can be delegated to the network. In this section, we study a set of algorithms that can utilize aggregation functions to improve their performance. Specifically, we consider two classes of algorithms: machine learning and graph analytics. The goal of this analysis is to show the potential traffic reduction that can be achieved when aggregating the traffic inside the network.

For machine learning algorithms, we use TensorFlow [1] to run two applications: a Soft-Max Neural Network using mini-batch Stochastic Gradient Descent (SGD) and Adam optimization [17] (Adam). We use a mini-batch of size 3 for the former and 100 for the latter. In these experiments, we use the MNIST[1] database of handwritten digits. The model is trained to correctly identify the digits present in each image. We deployed TensorFlow on six machines: one acts as the parameter server while the other five machines run as many worker processes. Each machine is equipped with 128GB of RAM and two 2.20GHz Intel Xeon E5-2630v4 CPUs.

Workers are responsible for compute-intensive tasks while the parameter server stores and maintains a set of shared parameters that comprise the trained model. In this setting, each worker is training the same model on different mini-batches of the data. In each iteration, the worker sends its parameter updates to the server which aggregates the local updates from each worker. Then, the parameters at each worker are updated according to their values at the parameter server.

In TensorFlow, the parameters are tensors, which are represented as large n-dimensional arrays. Parameter updates are deltas that change only a subset of the overall tensor and can be aggregated by a vector addition operation. We evaluate the overlap of the tensor updates, i.e., the portion of tensor elements that are updated by multiple workers at the same time. This overlap is representative of the possible data reduction achievable when the updates are aggregated inside the network. A high overlap means that aggregating the local updates of each worker inside the network could reduce the network traffic significantly.

Figures 1(a) and 1(b) show the amount of overlap among workers updating the same portion of tensors in the same iteration for SGD and Adam applications, respectively. Note that the overlap percentage is consistent among different iterations.

[1] http://yann.lecun.com/exdb/mnist/

The average overlap percentage is around 42.5% and 66.5% for SGD and Adam applications, respectively. Also note that the results in this experiment represent a lower bound of the possible overlap as the applications could be tuned to schedule communication to maximize overlap. In both applications, there are other tensors communicated over the network with a higher overlap percentage, or even fully communicated (100% overlap). We also experimented while increasing the number of workers from two to five (without changing the mini-batch size), and observed that the overlap increases.

We further consider graph analytics algorithms. We used the LiveJournal dataset,[2] which consists of 4.8M vertices and 68M edges. To run these algorithms, we deployed GPS [27] – an open-source Pregel clone – on four machines, each with 3.40GHz Intel Core i7-2600 CPU and 16GB of RAM. We consider three algorithms with various characteristics: PageRank, Single Source Shortest Path (SSSP) and Weakly Connected Components (WCC). The three algorithms are associated with a commutative and associative aggregation function.

Figure 1(c) shows the potential traffic reduction ratio for various graph algorithms using the LiveJournal graph. Each graph algorithm exhibits a different traffic volume. In PageRank, each vertex starts by sending its PageRank value to all its neighbours. Then, each vertex in the next iteration receives and sums the various values from its neighbours and calculates a new PageRank value. The traffic reduction ratio is calculated by combining all the messages sent to the same destination into a single message by applying the aggregation function used by the algorithm, i.e., *sum*, inside the network. In each iteration, all vertices are active and send messages to their neighbours; hence, the traffic reduction ratio is almost the same across all iterations. SSSP starts by sending a smaller number of messages from the source vertex. In the following iteration, the number of messages increases exponentially and hence a higher traffic reduction ratio is achieved. On the other hand, WCC starts by sending large number of messages from all vertices which decrease as the algorithms converges. The potential traffic reduction ratio in all the three applications ranges from 48% up to 93%.

In summary, applying in-network aggregation functions could significantly reduce the traffic of these applications.

4 SOLUTION SKETCH

As a proof-of-concept, we propose DAIET, a system for in-network aggregation designed to address the challenges presented in Section 2. While it has been designed with P4 and programmable ASICs in mind, it is general enough to be possibly implemented on different programmable data plane platforms. For the sake of presentation, we describe DAIET when applied to MapReduce-based applications. However, the proposed solution is generic enough and works well for various partition/aggregate data center applications.

To perform in-network aggregation, DAIET requires a close collaboration between the application and the network. Prior

[2]https://snap.stanford.edu/data/

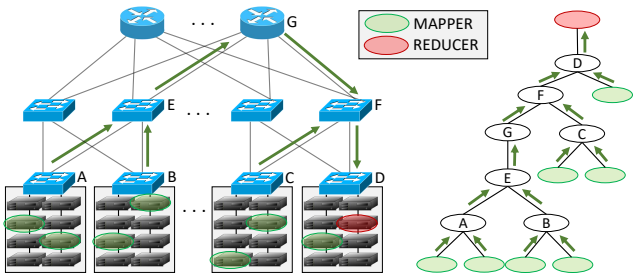


Figure 2: Aggregation Trees: example of physical and logical view for traffic aggregation in a data center network.

to starting a job, the master allocates the *map* and *reduce* jobs to the workers. This allocation information is exchanged with the network controller. Then, the controller defines the aggregation trees. An aggregation tree (Figure 2) is a spanning tree covering all the paths from all the mappers to a reducer. There is one tree rooted at each reducer. The network controller then configures the network devices, pushing a set of flow rules, to perform the per-tree aggregation and forward the traffic according to the tree. Specifically, each network device needs to know (*i*) the *tree ID* (i.e., *reducer ID*) that is embedded in packets addressed to one reducer, (*ii*) the associated output port to forward the traffic to the next node in the tree, and (*iii*) the specific aggregation function to perform. Moreover, for each tree a network device is configured with the number of children nodes it receives traffic from, so that the aggregated data are flushed to the next node when all the children have sent their intermediate results.

When the map phase is completed, each map task produces an intermediate data set consisting of key-value pairs, which is partitioned among the reducers. These partitions are sent to the reducer using UDP packets containing a small preamble and a sequence of key-value pairs. In the current prototype, we do not address the issue of packet losses, which we leave as future work. The abstraction of key-value pairs can be mapped on the messages exchanged in various data center applications; e.g., they can represent updates to shared parameters in a machine learning job or exchanged messages among vertices in graph processing.

The preamble specifies the number of pairs present in the packet and the tree ID the packet belongs to. We have carefully defined how the output of the map task is serialized in the local file, so that packets are transmitted without partial pairs. In fact, data cannot be deserialized during packetization, since it would greatly affect the execution time, therefore we use a fixed-size representation for the pairs, so that it is easy to calculate the offsets of pairs in the file and extract a number of complete pairs. This serialization and deserialization modification is not required in those applications that do not store intermediate results on disk. Finally, the end of the transmission is marked by a special END packet.

For each tree, network devices store two arrays, one for the keys and one for the values. These two arrays are managed as a hash table with buckets of only one element. Specifically, a hash function is used to convert a key to an index in the

Algorithm 1: Packet Processing Algorithm

```
Input: Network Packet P
Output: Aggregated Network Traffic
header ← parseHeader(P)
if header.type = DATA_PACKET then
    entries ← parsePayload(P, header.num_entries)
    foreach pair in entries do
        idx ← Hash(pair.key)
        if keyRegister[idx] is empty then
            keyRegister[idx] ← pair.key
            valueRegister[idx] ← pair.value
            indexStack.push(idx)
        else if keyRegister[idx] = pair.key then
            updateValue(valueRegister[idx], pair.value)
        else
            store(spilloverBucket,pair)
            if spilloverBucket is full then
                flushData(spilloverBucket)
else if header.type = END_PACKET then
    remaining_children = remaining_children − 1
    if remaining_children = 0 then
        flushData(keyRegister, valueRegister)
```

array. The index is used to access the two arrays and store the key and its corresponding value in the relative cells. If a collision is detected (a key generates the same hash of a different, already stored, item), then the new pair is not used for aggregation, and is stored in a different *spillover bucket*. This bucket is a queue of pairs with as many entries as the number of pairs that can fit in one packet. When this bucket is full, the entries are immediately sent to the next node in the tree. If the hash function distributes hash values evenly across the available range, this solution better employs the available memory (a scarce resource in data plane devices) without affecting the correctness of the final result. In fact this solution saves the allocation of multiple collision buckets that have a low probability to be used. The non-aggregated values in the spillover bucket are the first to be sent to the next node, so that they ware more likely to be aggregated if the next node is a network device and has spare memory. Additionally, an *index stack* is kept in the device memory to store the indices of the used cells in the two arrays. This facilitates flushing the results to the next node, avoiding a costly scan of the arrays.

Algorithm 1 summarizes the steps performed by the network device for each received packet. When a new packet containing key-value pairs is received, each pair in the packet is processed to update the local state. First, the hash function is applied to the key to obtain the corresponding index. This index is then used to access the keys array and check if: (*i*) the cell is empty, (*ii*) the same key has been received before, or (*iii*) a different key with the same hash value is already stored in the cell. In the first case, the new key-value pair is stored and the index is saved in the index stack. In the second case, the value is aggregated with the previously stored value and the result is stored in the array. In the latter case (i.e., a collision), the pair is stored in the spillover bucket. If this bucket is full, all its pairs are sent to the next node.

When an END packet is received, marking the end of one partition, the number of pending children (initialized by the controller) is decremented. When this value reaches zero, all the aggregated pairs in the two arrays can be sent to the next node towards the destination.

While usually a reducer receives the intermediate results from each mapper sorted according to the key, DAIET cannot preserve the order, thus the reducer receives unordered, aggregated, intermediate results. As a consequence, the intermediate results must be sorted at the reducer rather than at the mapper, which usually reduces the amount of parallelism. However, as shown in Section 5, the reduction in the amount of data to sort makes this overhead negligible.

5 PRELIMINARY EVALUATION

Our current implementation of DAIET is built using P4 and is available as open source[3]. As in a traditional SDN, the controller can configure a P4 data plane by pushing flow rules to a set of tables. These flow rules can match custom protocols and execute custom actions. We found that P4 imposes two main constraints affecting the implementation of DAIET: (*i*) a table can be applied at most once per packet, therefore it is not possible to apply the same table to all the headers in a stack of multiple headers of the same type, and (*ii*) the absence of variable-length data structures.

The first constraint, which is meant to avoid loops during packet processing, forces the programmer to manually perform loop unrolling, at the expenses of code readability and size. The second constraint is relevant in case of variable-length keys (e.g., strings). In fact, in this case the programmer is forced to reserve for each key as many bytes as the largest expected key, increasing the memory footprint, which in turn causes the allocation of arrays with fewer cells, thus increasing the possibility that a pair is not aggregated.

We present preliminary results from our prototype implementation. We focus on quantifying the reduction of traffic received by the root node of each tree (i.e., reducers) and the corresponding decrease in completion time at reducers. We believe this reduction, in a deployment with hardware switches, is expected to be proportional to the reduction in the job completion time, since each reducer will receive and process less data. However, as P4 hardware was not yet available to us, we obtain results using the bmv2 software switch,[4] which is not designed for line-rate packet processing. Thus, we cannot directly measure an improvement in job completion time but our results, which show around 88% median traffic reduction are still indicative of the expected benefits.

We run the experiments on a single server with two Intel Xeon E5-2680v2 CPUs with 40 logical cores in total and 768 GB of RAM. We run the bmv2 switch in a container on 4 dedicated cores, while 12 more containers, each with 2 dedicated cores, are used as workers to run 24 mappers (one per core) and 12 reducers (one per worker). An additional

[3] https://sands.kaust.edu.sa/daiet/

[4] https://github.com/p4lang/behavioral-model

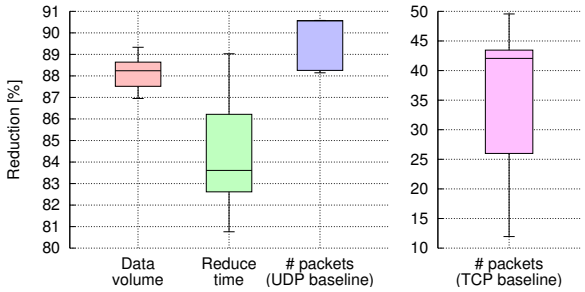


Figure 3: Reduction on the amount of data, running time and number of packets received at reducers.

container is used to run the master. The 12 workers execute a WordCount benchmark on an implementation of MapReduce adapted to send the map results using DAIET. The input dataset is a 500 MB file containing random words that are not causing hash collisions.[5] We do not test with a larger dataset because the network aggregation is performed by bmv2 using mainly a single core. We configure P4 registers to store 16K key-value pairs, so that, with words of maximum 16 characters and a 4 B integer value, the total SRAM required would be around 10 MB, which is a reasonable amount of memory for a hardware P4 switch. To quantify the reduction, we run the same benchmark in two other baseline scenarios without in-network aggregation: (*i*) using the original TCP-based data exchange and (*ii*) using UDP and the DAIET protocol, but without executing data aggregation in the switch.

Figure 3 shows a box plot of the reduction, across all the workers, in the total data volume and execution time at the reducers observed with DAIET compared to the first baseline. We observe that in-network aggregation provides a 86.9%-89.3% reduction of the amount of data received by the reducers. Because the smaller the data the less processing time at the reducer, we measured a median decrease of 83.6% in the execution time at the reducer, despite the received data are not sorted and require a complete sort operation.

Because current P4 hardware switches are expected to parse only around 200-300 B of each packet[6], we consider that one DAIET packet can contain at most 10 key-value pairs. Thus, our implementation generates more packets compared to the TCP baseline. However, the data volume reduction due to in-network aggregation is greater than the overhead caused by the additional packets. Figure 3 presents the reduction in the number of packets received by the reducer compared to the two baselines.We observe a median and maximum reduction of 90.5%, with a minimum of 88.1% compared with the baseline using UDP without in-network aggregation. Even considering the TCP baseline, we still measured a median 42% reduction in the number of packets. It is worth noting that an additional overhead in the data volume and number of packets is given by the fixed-size length of strings in our implementation, that forces a 16 B key even for smaller strings. This limitation will be removed in a future version of DAIET.

[5]Our current prototype does not manage collisions.

[6]According to private conversations with a P4 hardware vendor.

6 RELATED WORK

NetAgg [21] is a software middlebox platform that provides an on-path aggregation service. NetAgg middleboxes are deployed on servers directly attached to network switches through high-bandwidth links, composing an aggregation tree in the network. This requires changes in the network architecture. Furthermore, for computation-bound applications the middleboxes can become a performance bottleneck. SHArP [15] is designed to offload MPI collective operation processing to the network. Reduction operations are performed on the data as it traverses a reduction tree in the network, reducing the volume of data as it goes up the tree. SHArP only supports a limited set of combiners, since they are directly implemented in the switch ASIC. Unlike NetAgg and SHArP, DAIET does not modify the network architecture and provides more flexibility to support a variety of applications. Similar to NetAgg, Camdoop [8] also supports on-path aggregation for MapReduce-based applications. It leverages the capabilities of Camcube [2] which uses direct-connect protocols where all traffic is forwarded between servers without switches. Thus, it requires a custom topology and it is incompatible with a common data center infrastructure.

Besides data aggregation, IncBricks [20] is an in-network caching fabric with basic computing primitives. It comprises of programmable switches and smart NICs. It uses a key-value store as the application interface and allows to offload common compute operations on key-value pairs; e.g., increment, compare and update. Their design shifts the computation towards smart NICs since switches have limited storage. A specialized, in-switch key-value store for network measurement collection and aggregation also appears in Marple [25].

7 OUTLOOK

Programmable network hardware is finally emerging and provides the opportunity to revisit the idea of performing computation inside the network. Given ever more stringent requirements for data center applications facing hardware scalability bottlenecks and the end of Moore's law, programmable hardware appears to be the next frontier for achieving higher levels of efficiency and speed. Google's Tensor processing unit and Microsoft's Catapult projects are just two examples of this ongoing trend. We believe that the time has come to entrust network devices with part of the tasks typically executed by software. However, programmable networking devices have a distinct network machine architecture with stringent constraints. Determining the kinds of in-network computation, streaming algorithms and workloads that are going to be feasible under these architectural model is a major open challenge. As in the case of TCP offloading [23], we might need to see a period where variants are proposed, tested, evolved, and sometimes discarded. Data aggregation appears as a natural fit for in-network computation and our results are promising. But we view our work merely as an initial step towards the larger goal of judicious in-network computing.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We are grateful to Colin Dixon, Changhoon Kim, Jeongkeun Lee, Jeff Mogul, KyoungSoo Park and Amin Vahdat for their valuable comments and suggestions. We further thank Jeff for inspiring the title of this paper.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic Routing in Future Data Centers. In *SIGCOMM*, 2010.

[3] Barefoot Networks. Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.

[4] Barefoot Networks. The World's Fastest & Most Programmable Networks. https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[5] 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.*, 44(3), July 2014.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[7] Cavium. XPliant Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[8] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI*, 2012.

[9] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2), May 2016.

[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *SOSR*, 2015.

[11] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[13] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data Plane Performance Diagnosis of TCP. In *SOSR*, 2017.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[15] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *COM-HPC*, 2016.

[16] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band Network Telemetry via Programmable Dataplanes. In *SOSR*, 2015.

[17] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[18] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *NSDI*, 2016.

[19] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *NSDI*, 2016.

[20] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ASPLOS*, 2017.

[21] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-Specific On-path Aggregation in Data Centres. In *CoNEXT*, 2014.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, 2010.

[23] J. C. Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS*, 2003.

[24] S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *SOSR*, 2017.

[25] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

[26] D. A. Popescu, G. Antichi, and A. W. Moore. Enabling Fast Hierarchical Heavy Hitter Detection using Programmable Data Planes. In *SOSR*, 2017.

[27] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[28] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2, Nov. 1984.

[29] 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.

[30] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 26(2), Apr. 1996.

[31] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *SIGMOD*, 2014.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.