NetREC: Network-wide in-network REal-value Computation

Matthews Jose, Kahina Lazri, Jérôme François, Olivier Festor

NetREC: Network-wide in-network REal-value Computation

Matthews Jose*†, Kahina Lazri*, Jérôme François† and Olivier Festor†
*Orange, Chatillon, France, email: [matthews.jose,kahina.lazri]@orange.com
†Inria, LORIA, University of Lorraine, email: [jerome.francois,olivier.festor]@inria.fr

***Abstract*—The current generation of networks empowers the use of programmable switches whose behaviour can be defined using languages like P4. Nevertheless, these languages do not support network-wide deployment of stateful real-value functions. This paper presents NetREC, an extension of RMT programmable data planes designed to enable stateful real-value functions computation across multiple switches.**

NetREC first decomposes the real-value functions into a dependency graph of elementary operations that are distributed among the network. This distribution is carried out by dynamically generating and solving an integer linear program. We deploy a prototype of NetREC on a network of Tofino switches and demonstrate its capability of computing recursive real-value functions like exponential weighted moving average.

***Index Terms*—RMT, Floating Point Numbers, P4, Data plane Programming, SDN**

I. Introduction

The introduction of re-configurable switch architectures, *e.g.* Reconfigurable Match-Action (RMT), and network dataplane programming languages (like P4 [1]) allow to dynamically define packet headers and custom actions when processing packets. Different types of functions can be programmed and deployed in network without requiring middleboxes, for example to mitigate DDoS (Distributed Denial-of-Service) attacks [2]–[4] or to apply load-balancing [5].

Recently, there has been much interest in performing real-value computation on programmable switches. Functions used for aggregation, network telemetry and security that are traditionally computed on middleboxes [6] and on the SDN controller have been successfully deployed on the switch [7]–[10]. However, performing real-value operations in-network is very resource heavy. A simple addition operation with floating point numbers uses one-third of the resources in a pipeline [11]. On the other hand, network functions, increasingly using machine learning, rely on more computational steps [12]. In a network, coordinating several programmable switches could actually enable larger and more complex functions to be deployed in-network. Despite this, most solutions are designed for implementing functions on a single programmable switch.

Although the type of functions which can be deployed in-network is limited and mostly stateless, time-series are a class of functions that is widely used in network management and are recursive functions. Such functions are extremely difficult to implement due to architectural constraints in the stateful memory of programmable switches [13]. Several workarounds have been proposed [14]–[16] that mostly revolve around probabilistic approximations and packet recirculation mechanisms. However, recirculating a packet that takes microseconds whereas processing the original packet is in the order of nanoseconds. Furthermore, these proposals are application-specific.

We therefore propose NetREC. It automates the process of building pipelines for user-defined stateful real-value functions across multiple programmable switches and supports recursive functions. NetREC uses a tailor made representation for real numbers in combination with mathematical lookup tables and leverage integer linear program to distribute the network computation. The main contributions of this paper are:

- an internal representation to encode a real number to ease computation for programmable dataplanes,
- a procedure to analyse functional and state dependencies between multiple functions.
- a mirroring-based mechanism to overcome restrictions regarding stateful/recursive functions,
- a linear program for division of computational tasks, state placement and routing within a network,
- an experimental evaluation of NetREC.

The rest of the paper has the following structure. Section II highlights the restrictions of programmable switch architectures to support real numbers. In section III, our objectives are refined. Our new real number encoding scheme is presented in section IV. Section V introduces elementary functions calculation and management operations. NetREC relies on the optimization of an elementary graph described in section VI. The computational nodes of the graph are allocated to the switches by resolving a integer linear problem in section VII which is further optimized at the deployment stage as highlighted in section VIII. The evaluation of NetREC is done in section IX. Related work is presented in section X followed by a conclusion in section XI.

II. Lack of support for real-value functions

A. Floating Point Numbers

IEEE-754 is a technical standard for floating-point representation [17], defined to solve compatibility and portability issues among different hardware architectures. NetREC supports a variant of this standard qualified as half-precision and composed of three parts: sign bit, a 5-bits exponent and 10-bits mantissa. Thus, half precision floating point numbers are represented using 16 bits, resulting in a range of ±65504. The

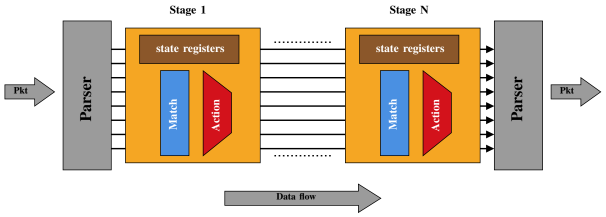


Fig. 1: RMT pipeline: a parser followed by a series of stages. The flow of data is uni-directional

density of these numbers is high around 0 and decreases at the extremes. Hence, operations performed using floating point numbers have to be rounded.

B. Programmable Hardware Switches

RMT [18] is a pipelined architecture for packet processing. A parser extracts fields from an incoming packet and stores them on a structure called the Packet Header Vector (PHV). It carries the parsed fields, along with packet metadata, through a series of stages. Each stage has match-action units used for matching and modifying values stored on the PHV. Stages have a limited amount of memory and RISC processor resources that are local to each stage. The processing time of a packet per stage is deterministic and constant.

Programming Protocol-independent Packet Processor (P4) [1] is the most widely used language to program different hardware switches. It is inspired by the RMT architecture. It supports various stateful processing structures like registers, meters and counters handled through vendor-specific extern functions. The language also supports the use of various base data types such as integers, bit-fields and boolean variables [19]. However, it does not natively support any real number representation scheme. NetREC uses P4.

C. Floating point numbers issues in RMT switches

As shown in figure 1 the flow of packets within a pipeline is uni-directional. To traverse the same stage multiple times, it has to be re-circulated through the whole pipeline, so doubling its processing time. In case of complex function with a number of stages higher than the hardware limit, this induces a significant delay. Unfortunately, real-value functions generally require several stages. For instance, a logistical regression function with only two input variables required six pipelines stages when implemented using a system like InREC [11], which is significant for current hardware. A simple addition operation between two floating point numbers requires four stages.

Moreover, simple floating point operations, particularly multiplications, heavily depend on the sign of each number. In previous implementations of floating point numbers on RMT switches, the sign value is checked using a control statement (e.g. an if statement) or is relegated to a lookup table [11], that uses a whole stage. This problem of inefficiency extends to implementation of certain frequently used basic operations. Therefore, the first challenge to be addressed by NetREC is **the lack of efficient real-number function computation (C1)**.

While the RMT pipeline has a stateful register to store a key-value pair at each stage, its access is limited to the current stage. Pipelines for recursive functions (eg. ARIMA model, EWMA) generally use several stages and require stateful registers to implement. Also, they require the following sequence of operations to be performed: 1) read a value from a register, perform computation across multiple stages and update the register read initially. **Support for recursive functions (C2)** is the second challenge. It requires a new approach as such a function might need to retrieve values computed by an earlier stage.

Moreover, the PHV is composed of fixed containers that are 8, 16 or 32 bits wide on which actions (for example bit-field operations) can be applied. Large containers can store different packet headers and several small containers can be glued together to store large headers. Thus, the size of the various components of a number (for example the mantissa of 5 bits) are not aligned with the structure of a PHV container. This results in the difficulties when the operation modifies different parts of a float simultaneously [11]. The **lack of an efficient real-number representation (C3)** is another challenge that directly contributes to **C1**.

D. Computing a function across multiple switches

As mentioned in the previous section implementing real-value functions can exhaust the stages available in a single switch, leveraging multiple switches would enable larger functions to be implemented.

However, functions are composed of elementary operations (addition, multiplication, log...) that are interdependent and hence have to be computed in a specific order. Moreover, Intermediate results have to be carried (and so routed) to each switch involved in computing the function.

Parallel deployment can be leveraged but raises several issues. First, registers used in stateful computation must be synchronized in all nodes. Even for stateless functions, the placement of different elementary functions is very important. For example, a placement scheme of the function $f(x) = g(x) + h(x)$ such that $h(x)$ and $g(x)$ are placed on two different branches would lead to error since the value of $h(x)$ and $g(x)$ will not arrive at the same time in the merge node assuming caching is prohibited to avoid latency. However, an alternative placement scheme in figure 2(b) does not result in an inaccurate result while still doing some parallel computation. This is particularly important to consider when multi-path routing is used, which would be an additional advantage to distributed the computational load over the network. Our last challenge is so to automatically **define correct and efficient placement scheme on the programmable switches (C4)**.

III. Objectives and overview of NetREC

Given a network topology and a set of real-value functions (including recursive real-value functions) to be applied on

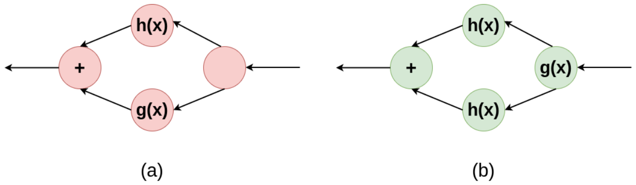


Fig. 2: Alternative placement schemes: (a) will result in inconsistent computational results, whereas (b) will not.

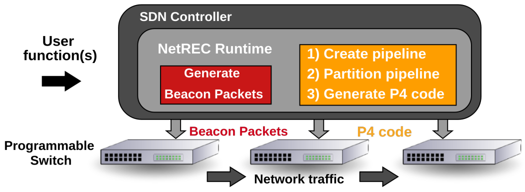


Fig. 3: A global overview of NetREC

specific flows, NetREC aims at building a single unified computational pipeline to be distributed amongst the switches. The proposed system also handles traffic routing, *i.e.* consider the path of network flows in order to decide how to distribute the pipeline, and minimize the overall latency. To reach this ultimate objective, NetREC must address **C1**, **C2**, **C3** and **C4**.

As shown in figure 3, NetREC takes a user defined function, partitions it across multiple switches and generates P4 code for each switch. NetREC builds on the strength of InREC [11], using mathematical lookup table for computing elementary operations and functional decomposition to analyse dependencies among them. NetREC extends the latter by considering multiple functions to be computed across a full set of switches and also adds the support of stateful recursive functions. The design of NetREC is compliant with the programmable switch architecture by breaking down real-value computation into its basic processing units and then mapping them to structures available on the switch. Currently it is designed to work with continuous non-recursive real-value and linearly recursive functions. The whole process of building the pipeline is NP-complete because of the integer linear problem.

NetREC is composed of two main parts. The first defines three features: 1) a representation scheme optimized for storing real-numbers on RMT switches, 2) a lookup table-based implementation for a set of elementary operations and 3) a mechanism for state update that is required when implementing recursive functions. The second part uses these features to build a unified pipeline across multiple switches for a set of user functions.

IV. Optimized Intermediate Representation

NetREC introduces **Optimized Intermediate Representation** (OIR). An OIR-encoded number is 40 bits wide with:

- a 32 bit wide fixed-decimal representation. However, the exact number of bits representing the whole number part and the decimal part of the fixed-decimal is ascertained during compilation by NetREC depending on the domain, the range and the accuracy of the function to be computed. For example, a normalized function whose output is between 0 and 2 needs only a single bit-wide whole number part with the remaining bits for the decimal part. Also, two's complement system is used to represent positive and negative numbers, bringing the total number of positive and negative numbers to $2^{31} - 1$ and 2^{31} respectively. In the worst case, 16 bits have to be used for the whole number part to cover the entire range of half precision floating point numbers.
- the second part (8 bits) of an OIR number are the management bits. They are used to accelerate (or simplify) operations to be applied on the value of the number (stored as a fixed point number). For example, computing the logarithm of a number x can be performed by the following procedure $log(x) = n + log(x >> n)$ with $n \in Nand(x/n) \ni 1 < x/2^n < 2$. Bit shift $(log(x >> n))$ can be computed using a simple lookup table with x as the key and the value n pre-computed and stored in the management bits of the OIR. Computing n normally would require us to count the number of leading 0s in the fixed decimal and so would consume a whole stage. The management field can be used for storing a range of values to help computing, varies with elementary operation and is modified based on the next operation in the pipeline (see section VI-B).

NetREC uses OIR to represent real values within the network including all stateful registers. To properly address **C3**, this representation is aligned with PHV container sized (32 and 8 bit-long containers). The transformation to and from OIR is performed at the start and end of the pipeline. The advantages of OIR can be summarized as: **(1)** fixed-decimal representation can use the arithmetic operations provided by P4 (hence the RMT instruction set) directly unlike InREC [11], **(2)** negative numbers are managed by the representation without the need for control statements and **(3)** the impact due to the lack of rounding is greatly reduced, since fixed-decimal numbers are more uniformly distributed (knowing than floating point number density is higher around 0 by specification).

V. Elementary and Management Operations

A. Elementary operations

Elementary operations are the building blocks for any function. NetREC uses a set of predefined procedures to implement them after manual optimization to overcome the limitations mentioned in section II-B as highlighted in Table I. These procedures leverage mathematical lookup tables in combination with the operations provided by P4 [1](eg. addition, multiplication, bitwise shift). As shown in the table, OIR highly simplifies the implementation of these operations and so reduces the number of stages needed to compute them in a RMT switch as induced by the challenge **C1**. For example,

[1] RMT switches have native instruction sets to support these operations

Elementary operation	# of stage in INREC	# of stage in NetREC
+	4	1
*	3	3
* (by a constant)	1	1
$log_2(x)$ (generic)	5	1
$log_2(x)$ (bounded interval form)	1	1
2^x	3	2
$sin(x)$	1	1
$\frac{x}{n}$	1	1

TABLE I: A comparison between InREC and NetREC on the number of stages needed to implement various elementary operations.

addition in InREC [11] using floating point numbers uses 4 stages but only a single action unit process with NetREC.

B. Management operations

NetREC also defines procedures for management operations. A time slot window is an operation that cannot be directly implemented on the pipeline and is a critical component for implementing time-series functions. To enable deployment of functions that make use of this, NetREC, uses the controller to send out a beacon packet, that signals the start of a new time slot window.

As mentioned in section II-C, updating a register from a subsequent stage in the pipeline is impossible. To support recursive functions **(C2)**, NetREC relies on egress packet mirroring mechanism. Egress packet mirror duplicates the packet resulting from egress processing and sends it to a pre-configured output port. The duplicate packet is placed at the start of the egress pipeline and is processed like any normal packet. Therefore, a newly computed state value can be transferred back to the beginning of the pipeline and used in the proper stage (for being stored in a register which can be accessible when processing the next packets). This mechanism has an advantage of being non-blocking in nature (as opposed to recirculating the packet). However, with high bandwidth traffic and depending on the number of switches used, there can be large delays and so inconsistencies when updating the initial register. To address the former NetREC partitions traffic using flow identifiers (section VI-A) reducing the number of packets used for computation in each flow. We will demonstrate in section IX that the error can be tolerable.

VI. Inter-function dependency analysis

A. Elementary graph

From a set of user-given functions, NetREC starts by expressing them as a graph called elementary graph. The elementary graph is a directed acyclic graph that represents a set of decomposed real-value functions where nodes represent an elementary operation or a variable and a directed edge between nodes indicating a dependency, i.e. the output of the node is used as input for the other.

Hence, it is equivalent to the order in which the mathematical and management operations have to be applied. Figure 4 shows an example of an elementary graph. For sake of clarity, only a single function, the Exponential Weighted Moving Average (EWMA) of the size of packets, is presented $f_t = (1 - \alpha) f_{t-1} + \alpha \times pkt\_size$ **(1)**. It is computed by time windows, which implies to do the cumulative sum of the size of a packet denoted as pkt within a single window. Depending on available switch feature, the implementation of time windows can vary. In the worst case (when no time function is available), we consider here a controller sending beacon packets.

In this example, we assume the topology in figure 5 with 6 virtual input and output ports delimiting the boundaries of the network, capable of performing real-number computation.

Each color represents a different type of node:

- **Port numbers (green)**: these ports indicate the network boundaries of the network within which the function has to be calculated. Assuming the topology of figure 5, only flows traversing the network from ports 1 and 2 toward port 4 are considered.
- **Flow identifiers (light-grey)**: explicit definition of the flows based on packet headers used for computation. Such a node is placed at the output of the port nodes in order to only select packets belonging to the defined flow. Without loss of generality, this example illustrates the whole process when a beacon is received, so the flow identifier should match the header of the beacon packet.
- **Variables and constants (blue)** are elements that can be directly accessible in PHV (*e.g.* $size$ is the current size of the packet) or in registers like $\alpha_1 = \alpha$ and $\alpha_2 = 1 - \alpha$. Using α_2 in our example as constant defined by the user would avoid recomputing it at each time window.
- **Computation nodes (red)** are arithmetic operations such as the addition or the multiplication in figure 4
- **State-storage nodes (yellow)** represent the access to stafeful values as read (S nodes) or write (M nodes). So $S_{f_{t-1}}$ retrieves the value of the previous times window f_{t-1} while S_{pkt} reads pkt. As shown in the figure, pkt is incremented by the size of the current packet ($size$) before being saved. As highlighted here, the elementary graph does not need to specify how the states are managed. Because pkt must be read and write within the same graph, NetRECwill leverage the capability to mirror packet as explained in section V-B. The packet is thus mirrored in M_{pkt} node with the state value pkt to be saved in the register used in S_pkt when receiving the next packet of the processed flow. At each time window f_{t-1} is updated similarly in node $M_{f_{t-1}}$
- **Action nodes (dark-grey)** specifies the actions to apply on the packet such as drop or forwarded. Obviously, such action can be conditional or use Match-action tables based on the computed values as lookup keys, for example f_{t-1}.

In addition to their type, each node contains other attributes (in figure 4, these attributes partially appear for sake of clarity):

- **Pipeline resources** provide details about the number of stages, SRAM, TCAM and PHV memory necessary for elementary operations based on table I. This applies to the computational nodes in figure 4 (addition and

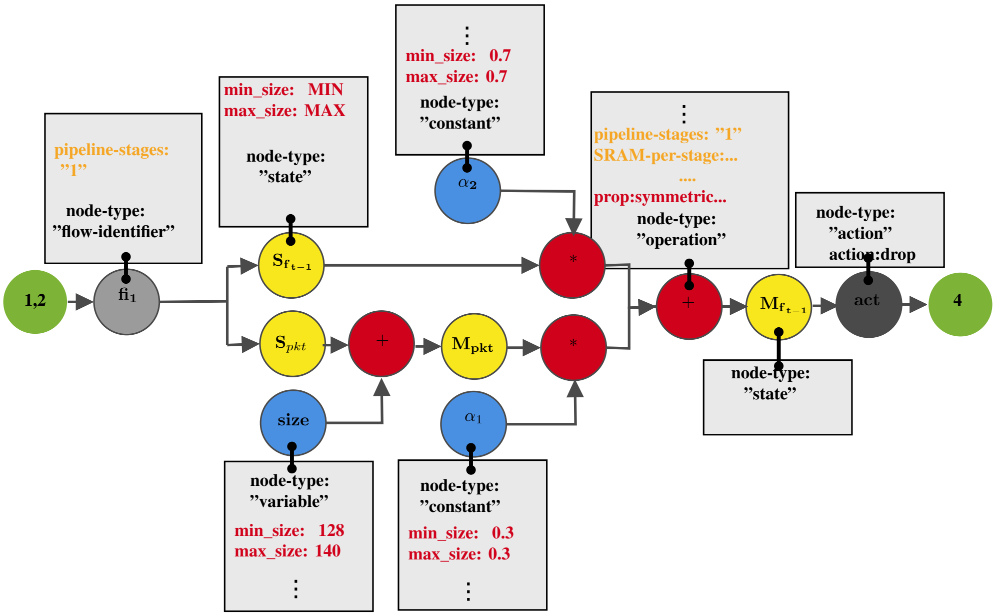


Fig. 4: Elementary graph for EWMA function

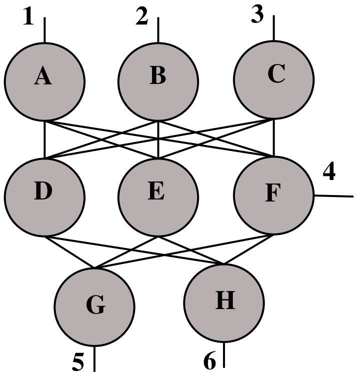


Fig. 5: A network topology consisting of 8 programmable switch indicated by the letters with 6 virtual ports serving as entry/exit points to the network. NetREC obfuscates the internal details of the network and presents only the edge ports as virtual ports.

multiplication).

- The **minimal and maximal values** of a constant or a variable if they are known. This is used in the next stage to perform aggregation and predict what will be the domain of computation functions and so estimate the overall memory (table size) needed to deploy that node. By default, the min and max values of OIR used. The interval can be restricted depending on the properties of the node. In figure 4, α_1 and α_2 are constant and so equivalent to their bounds. Packet size or other packet attributes can be also bounded by nature or protocol definition.
- The **computational properties** of an operation (*e.g.* symmetry) or **extrinsic properties** of values of constants, *e.g.,* TCP port numbers are discrete.

B. Reduction

The next phase consists in optimizing the elementary graph by reducing its overall size and simplifying computational nodes **(C1)** using two steps:

1) From initial value ranges in the elementary graph, the mathematical domains can be derived. The domain of a function actually depends on its intrinsic definition and the possible values as input. For example, the domain of $log(x)$ assumes x to be positive which can be further restricted if x is the output value of sin function. In that case, the domain is limited to $[-1, 1]$. Hence, constraints can be easily propagated in the graph like our previous approach [11] but considering the domain and ranges as OIR which requires to also populate their management bits. The management bit of each OIR are populated with values that support the real number computation. An example has been presented in section IV about log computation. A few other examples include the value of x divide by 2π for $sin(x)$ and the result of "if value of x is greater than n or not" for $\frac{x}{n}$. These highlight other implemented operations where management bits are helpful, mostly by pre-computing to reduce the number of operations to be performed (like bit shift, comparison...). As clearly highlighted, the management bit of the operation op depends on the value of its parameters. Therefore, constraint propagation also impact the management bits.
2) Graph aggregation or vertical reduction: this process combines the nodes of a subgraph that has a finite number of possible input combinations into a single node, *i.e.* a single lookup table. This process is carefully done, ensuring that the number of stages required to store the single table is less than the total number of stages needed to implement the subgraph. We applied the method previsouly defined in [11].

C. Inter-node dependencies

Following reduction, NetREC identifies groups of nodes of the graph that must met certain conditions when deployed on the switch. The most notable example is with state variables nodes that require both nodes (read and mirror, mentioned in section V-B) to be ideally placed on the same device to minimize error. This stage is essential to enable the distribution of computational tasks **(C4)**. It also identifies nodes that cannot be computed in parallel. For all atomic operations, for example the binary operation $+$, it can potentially produce inconsistent results if both the input branches are placed on separate parallel devices in the network since there is no guarantee that both branches will finish computing their respective results at the same time. In figure 4, the two branches leading to the

N	nodes of the reduced elementary graph
f	the flow identifier
s_f	source of the flow f
t_f	sink of the flow f
$p_f(u,v)$	proportion of flow f between the nodes u and v
$W_f(n,s)$	1 if $n \in N$ is deployed on s for flow f, else 0
$W_f(n,u,v)$	equals $p_f(u,v)$ if node $n \in N$ is already allocated, else 0
r_n	number of stages to implement a node $n \in N$
c_s	total number of stages on a switch s
$S \subset N$	state nodes (read and write)
$S_d : S \times S$	set of dependent read/write nodes (s_r, s_u)

TABLE II: Term definition

addition node cannot be placed on parallel branches(as shown in figure 2). This information is used in the next stage to formulate the constraints for the linear program.

VII. Distributed dataplane deployment

Based on the elementary graph which has been reduced, NetREC solves an Integer Linear Problem (ILP) to find the optimal placement of all the elementary operations on the underlying network, *i.e.* on multiple switches as expressed by the challenge **C4**. Based on terms in table II, the ILP is formulated as an extension of the multicommodity flow problem [20] to maximize the link utilization while routing the packets to their destinations.

$$\forall n \in N, W_f(n,s) + \sum_k W_f(n,s,k) \geq W_f(n+1,s) ...\mathbf{(1)}$$

$$\forall n \in N, W_f(n,t_f) + \sum_k W_f(n,k,t_f) = 1 ...\mathbf{(2)}$$

$$\forall_f \forall_s \sum_s r_n * W_f(n,s) \leq c_s ...\mathbf{(3)}$$

$$\forall n_{state} \in S, \sum_s W_f(n_{state},s) = 1 ...\mathbf{(4)}$$

$$\forall n_{state} \in S, \sum_k p_f(s,k) \geq W_f(n_{state},s) ...\mathbf{(5)}$$

$$\forall n_{state} \in S, \sum_k W_f(n_{state},k,s) + W_f(n_{state},s) = \sum_l W_f(n_{state},l,s) ...\mathbf{(6)}$$

$$W_f(n,u,v) \leq p_f(u,v) ...\mathbf{(7)}$$

$$\forall (n_{stateread}, n_{stateupdate}) \in S_d . W_f(n_{stateread},s) = W_f(n_{stateupdate},s) ...\mathbf{(8)}$$

To ensure that traffic is routed through a specific sequence of nodes as indicated by the dependencies in the elementary graph, we introduce the variables $W(n,s)$ and $W_f(n,u,v)$ in the inequalities **(1)** and **(2)**. The constraint **(3)** guarantees the allocation of multiple computational nodes in a switch does not exceed its number of stages reminding sequential operations cannot be in parallel in the same stage. SRAM resources are considered in the final stage described in section VIII.

State nodes in the elementary graph have two more additional requirements. Indeed, the consistency of the state must be maintained. Only a single copy of the state can exist because synchronizing different switches while preforming operations at line rate is impossible. Hence, Inequalities **(4)**,**(5)**,**(6)** and **(7)** ensure that a node representing a state can only be deployed on a single switch in the network. Secondly, the register and the mirroring primitive needed to implement a state variable must be deployed on the same device as forced by equation (**8**) (in section IX-E, we experimentally verify that using a different device would lead to an unacceptable computational accuracy). Furthermore, since the read and updated operations have to be atomic in nature, state variables are placed linearly thanks to equations (**1**) and (**2**). Table 11 shows the number of switches and stages with respect to network topology and real-value function. The grey cells show the case where state is global for all traffic.

VIII. Switch specific deployment

In the final stage, NetREC substitutes switch routines for each elementary operation and generates the equivalent P4 code. Once a switch is assigned a certain section to compute, a local optimization is performed. Even if the stage requirements have already been checked, the memory being split among the various parallel tables have to be ascertained within each stage. The first step is to aggregate the parallel pipelines according to common elementary operations they can share.

NetREC identifies all the common elementary computational nodes pairs between any two branches (from each function or pipeline to compute) in the subgraph deployed on a given switch. A naive approach would have merged all these pairs, but this would raise several issues. Assuming a pair of nodes, one from each pipeline, n_1 and n_2. Both represents the same operations op but possibly with two alternatives versions, op_1 and op_2, depending on constraints propagated individually through each graph (for each pipeline). op_1 and op_2 can thus have a different accuracy respectively acc_op_1 and acc_op_2 and could use a different number of stages $\#stages\_op_1$ and $\#stages\_op_2$ (as highlighted in section IX-A). To avoid lowering the accuracy, only the most accurate version should be kept (op_i such that $acc\_op_i = max(acc\_op_1, acc\_op_2)$). Also, the range of values represented in the lookup table is extended to cover the full ranges of values for both alternatives. However, if this results in increasing the number of stages of the other alternative version $\#stages\_op_i$ $>$ min($\#stages\_op_1$,$\#stages\_op_1$), the computational nodes are not merged. In that case, there would be a risk of stage overflow as the subsequent operations (computational nodes) with a lack of enough free stages.

If all these checks are passed, the nodes are candidates to be merged in a last step. NetREC performs this task in an iterative manner by considering the pairs in the order of memory (SRAM) which can be saved. This process forbids a new merged node n_2 to be always computed after a merge node n_1 in all pipelines due to the unidirectional nature of flows through the stages. Otherwise, n_2 will not be created, and each pipeline holds its own version.

IX. Evaluation

A. P4 code generation

A pipeline substitution step replaces the abstract node representation with a table scheme as follows. First, the properties of the node are parsed to choose one variant of the elementary operation. For example, the general implementation of the log operation is a series of two lookup tables followed by an action operation whereas the same operation with its domain variable being finite can be implemented in a single lookup table. Second, entries for each lookup table are generated based on the domain of the function represented by the node. Finally, P4 equivalent code is generated to create the lookup table and its corresponding action units and metadata variables needed to carry information through the switch pipeline are defined for each temporary variable. Also, headers are defined for parsing and emitting incoming and outgoing OIR structures.

B. Setup

Our setup consists of a Tofino-based Stordis BF2556X-1T-A1F switch connected to IBM Blade Center HS22 7870 servers with an Intel Xeon X5660 2.80 GHz and 100Mbps NIC. For the evaluation, we use the 4 parallel pipes, to simulate 4 different routers interconnected in a full mesh topology. This setup is relatively small but, due to the nature of our research, we prefer an evaluation with real hardware than simulation or emulation. The function to be computed, and its inputs are sent in a packet to the switch. The result is then sent back to the controller, which performs the same computation on a commodity computer and compares the results.

C. Error in a single elementary operation using OIR

In figure 6, we evaluate the relative error when using the OIR to perform a single elementary operation $op(x)$ while varying x between 0 and 20000. This error is relative to the same operation performed using floating point numbers with python. Similarly, the figure reports the relative error using our previous approach, InREC, based on floating point numbers (with a 10 bits mantissa). NetREC has a higher relative error than implementations using floating point numbers in InREC but are still less than 5%. Operations that do not use lookup tables like $+$ and $*$ have a higher relative error due to the lack rounding. Errors are mainly due to the impossibility to represent accurately the decimal part for smaller numbers with OIR. However, the error is still lower than 1%. This can be considering as enough accurate for many applications while the needed resources are drastically lower as shown in next section.

D. Resource overhead

Figure 7 shows the number of pipeline stages used, the use of SRAM and Very Long Instruction Words (VLIWs). For SRAM and VLIWs, an average over all stages is computed.

$sqrt(x)$ uses the most resources (25% or 3 out of 12 stages and 13% of the total SRAM available) because it requires a log operation followed by a bit-wise operation to compute. With floating point representation, a simple addition requires 4 stages as explained in section II-C which led to one of the most computational operation with InREC [11]. NetREC can use the native $+$ operator with OIR and so a single stage and a single action unit and stage. Most other operations are fully lookup table based and use a maximum of 2 pipelines stages. OIR reduces the overall resource usage across the board and consumes less SRAM and VLIW instructions per stage (close to 0 in figure 7) leaving room for parallel computing (other packet processing program). Furthermore, the amount of SRAM used is further reduced in Stage 2 because of variable constraining that results in fewer entries.

E. Error due to state update over multiple network hop(s)

Figure 8 shows the relative error when a state update packet is mirrored at variable distances (measured in hops) from the state register and at various bandwidths. Indeed, we claim that the mirroring packet has to be used within a single switch (as mentioned in section V), i.e. both the read and update actions are co-located but one could argue for transmitting the state values over a set of switches to have higher flexibility. For our evaluation, we consider the EWMA function across 1, 2 and 3 switches and compared the results obtained against a Python-based implementation. To eliminate the error due to the time-slot delay, we set the time-slot interval as 1 minute and performed the test within this interval.

At bandwidths above 147 Kbps, an error higher than 95% is observed for 1 and 2 router hops. This is because the mirrored packet takes a significant time to reach the initial state register as other traffic flows are interleaved. Packets of a flow are ignored until the update packet is received at the register resulting in the difference in computed value. However, at speeds of 147 Kbps or if the register is on the same switch as the mirrored packet(0 hops) the error is negligible. Hence, it is not recommended to have several network hops between where a state register is read and where an update packet is generated. NetREC performs optimizations(to minimize the number of hops. Multiple hops can be used in the case of low bandwidth flows like DNS traffic (usually between 1Kbps to 147Kbps per flow).

F. Error due to state update mechanism using mirror

The mirroring mechanism we use to update the state register is such that mirrored packets have to be interleaved between subsequent packets in a flow. To simulate the probability that packets are interleaved in the right order we make the following model the switch pipeline as a series of n packets that have to pass through the switch pipeline within a second in order to sustain a bandwidth B. Assuming packets arrive in a uniform stream and that each packet arrives sometime within a fixed time interval that can have a maximum value t_{max} that is $\frac{1}{n}$. Hence, the sooner mirrored packets reach within the interval t_{max} the more probable it is to obtain correct ordering. Before the mirrored packets can be processed the primary packet has to be processed the mirrored packet has to be generated which are represented by the variables $P_{processing}$ and $P_{mirrorgenerate}$. Assuming a uniform distribution of when

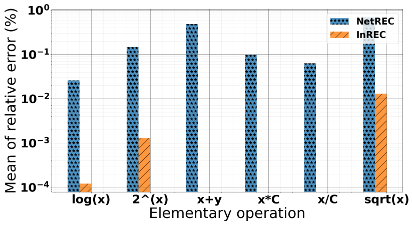


Fig. 6: Error in a single elementary operation

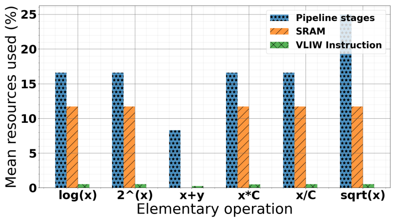


Fig. 7: Resource overhead for various elementary operations using NetREC

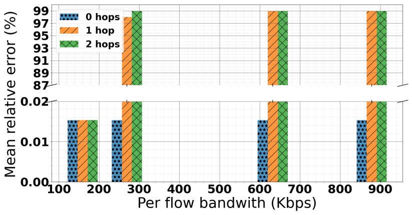
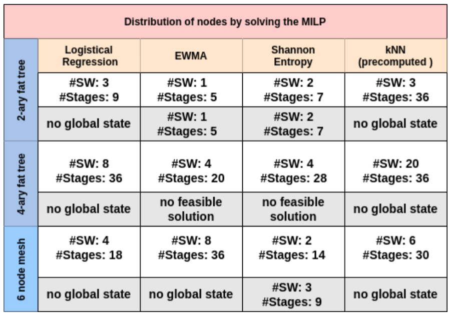


Fig. 8: Error due to state update over multiple hops



Distribution of nodes by solving the MILP				
	Logistical Regression	EWMA	Shannon Entropy	kNN (precomputed)
2-ary fat tree	#SW: 3 #Stages: 9	#SW: 1 #Stages: 5	#SW: 2 #Stages: 7	#SW: 3 #Stages: 36
	no global state	#SW: 1 #Stages: 5	#SW: 2 #Stages: 7	no global state
4-ary fat tree	#SW: 8 #Stages: 36	#SW: 4 #Stages: 20	#SW: 4 #Stages: 28	#SW: 20 #Stages: 36
	no global state	no feasible solution	no feasible solution	no global state
6 node mesh	#SW: 4 #Stages: 18	#SW: 8 #Stages: 36	#SW: 2 #Stages: 14	#SW: 6 #Stages: 30
	no global state	no global state	#SW: 3 #Stages: 9	no global state

Fig. 11: Task distribution with respect network topology and real-value function

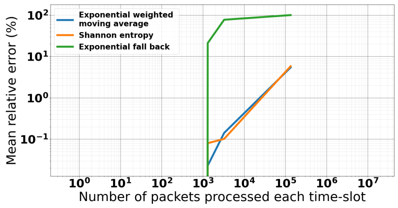


Fig. 9: The relative error when calculating various functions for different quantities of packets in a time-slot

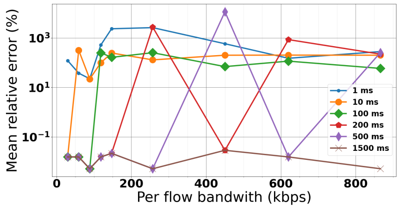


Fig. 10: The mean error when computing a simple EWMA with different time-slot values over various bandwidths

the packet with arrive in this time-interval t_{max}, the probability that the mirrored packet before the packet is P_{order} that is $1-\frac{P_{processing}+P_{mirrorgenerate}}{t_{max}}$. The probability that all packets are interleaved in the right order is $(p_{order})^n$, this computes to 0.0000001 for 1Gbps, 0.0136 for 25Mbps, 0.50327 for 10Mbps, 0.993 for 1Mbps and 0.998 for 500Kbps. Figure 9, extends on this and shows the relative error when calculating various functions for different quantities of packets in a time-slot using the probability p_{order}.

G. *Error due to time-slot vs bandwidth*

Figure 10 shows the relative error at different bandwidths when computing the EWMA function with different time-slot values. As introduced in V-B, the end of the time window is triggered by a beacon packet sent from the controller. At the switch level, the time window is so not perfectly aligned with the precise time windows of the controller. It is due to Round-Trip Time (RTT) between the controller and the switch (maintained to 3ms) and because of the underlying traffic which competes with beacon packets in the switch queues.

The error is higher when a larger bandwidth of traffic is sent to the switch. At lower bandwidths the error is very small from 200, 500 and 1500ms. However, in the case of 1ms interval, even low bandwidth traffic has a significant error due to the RTT between the router and the controller being three times more the time slot. In contrast, with high time slot intervals (500ms and 1500ms), the error remains low even for medium bandwidths(around 300 Kbps) because a lower bandwidth leads to a larger gap between packets, where the beacon can be inter-leaved. Actually, the error rate is intrinsically linked to the probability the beacon packet will reach and reset both registers at the right time. If the beacon packet does not arrive in time to signal a new window, all packets in the flow that are received till this event will contribute to an error in the result being computed.

X. Related Work

A. *Stateful dataplanes*

FAST [21] and OpenState [22] provided flow level stateful elements on a single switching device. SNAP [12] extends this to a network-wide implementation, that performs state placement based on a dependency analysis. Domino [23] is a programming language introduced proposes the use of structures called atoms at each stage to store stage on a single switch. Kinetic [24], uses a controller assisted method to store state arrays using a non-blocking paradigm on a flow level. Several switches [25] that are P4 complaint have stateful registers at each stage as an extern function. Flowradar [26] uses reverse blooms filters to store flow level counters in-network. Compared to these other proposed solutions, NetREC allows for the operator to store multi-stage state variables, these variables are read and updated at different stages.

B. *In-network real value computation*

Several methods have been proposed to perform real valued operations in-network. Naveen Kumar *et al.* [27] implemented RCP, and calculate fair-rate which uses division in-network. They used fixed point decimals to represent real numbers. Sonate [28] uses programmable switches to perform in-band telemetry but are restricted to a certain class of functions. In [29], the authors implemented an entropy function for DDoS detection directly on the dataplane thanks to a combination of sketches and look up tables to calculate packet frequencies. Recently, new methods for computing elementary operations log and exp have been proposed in [30]. We can also mention [31] that implements floating point arithmetic in-network with 99.94% accuracy in the worst case. N2Net [32] and BaNANA Split [33] have shown implementations of binary neural networks on the dataplane. Recently several aggregation use-cases have also emerged [7], [34] However, these proposals provide methodologies to optimize specific functions or elementary operations and are generally meant to be deployed on a single switch, they do not address a network-wide function deployment that NetREC does.

XI. Conclusion

NetREC is a new approach to support real-valued functions on distributed RMT based switches. From an elementary graph that is compacted and refined along different processing steps, it then combines native operations and LUTs to construct a minimal switch specific pipeline. This pipeline is expressed as P4 logic and distributed to operational switches. The evaluation on a Tofino programmable switch shows that reaching a relative error below 5% or even 1% is possible with a low amount of resources making NetREC a viable approach to support complex functions. Currently, our tests have been preformed only with non-recursive continuous functions and single variable recursive functions. Other types of functions will be covered in our future work along with an in-depth investigation of the scalability. Since NetREC uses an ILP, possible heuristics can be considered. Also, in future work, we plan to release NetREC as an open-source software.

References

[1] 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.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[2] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *IFIP/IEEE Symposium on Integrated Network and Service Management*, 2019.

[3] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of NDSS*, 2020.

[4] M. Dimolianis, A. Pavlidis, and V. Maglaris, "A multi-feature ddos detection schema on p4 network hardware," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2020, pp. 1–6.

[5] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[6] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, Aug. 2011.

[7] A. Sapio, M. Canini, C.-y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *KAUST*, 2019.

[8] E. D. Changhoon Kim, Parag Bhide. In-band network telemetry (int). [Online]. Available: https://p4.org/assets/INT-current-spec.pdf

[9] switch.p4. [Online]. Available: https://github.com/p4lang/switch/blob/master/p4src/switch.p4

[10] M. Jose, K. Lazri, J. François, and O. Festor, "Leveraging in-network real-value computation for home network device recognition," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 734–735.

[11] M. Jose, J. François, and K. Lazri, "InREC: In-network REal Number Computation," in *IFIP/IEEE Symposium on Integrated Network and Service Management* . IEEE, 2021.

[12] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *ACM Workshop on Hot Topics in Networks*, 2019.

[13] J. Zhang, W. Bai, and K. Chen, "Enabling ecn for datacenter networks with rtt variations," in *International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, 2019.

[14] I. Kunze, M. Gunz, D. Saam, K. Wehrle, and J. Rüth, "Tofino+p4: A strong compound for AQM on high-speed networks?" in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021.

[15] J. Zhang, W. Bai, and K. Chen, "Enabling ecn for datacenter networks with rtt variations," in *CoNEXT '19: Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 12 2019, pp. 233–245.

[16] S. Wang, J. Bi, C. Sun, and Y. Zhou, "Prophet: Real-time queue length inference in programmable switches," in *ACM Symposium on SDN Research (SOSR)*, 2019.

[17] Floating point numbers. [Online]. Available: https://floating-point-gui.de/formats/fp/

[18] 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 Conference*. ACM, 2013.

[19] M. Budiu and C. Dodd, "The p416 programming language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, p. 5–14, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3139645.3139648

[20] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. USA: Prentice-Hall, Inc., 1993.

[21] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation," *IEEE International Conference on Network Protocols (ICNP)*, 2018.

[22] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, april 2014.

[23] 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," *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[24] J. Santiago da Silva, F.-R. Boyer, L. Chiquette, and J. P. Langlois, "Extern objects in p4: an rohc header compression scheme case study," in *IEEE Conference on Network Softwarization and Workshops (NetSoft)*, jun 2018.

[25] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "drmt: Disaggregated programmable switching," in *SIGCOMM Conference*. ACM, 2017.

[26] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *NSDI*, 2016.

[27] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *14th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Mar. 2017.

[28] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, 2018.

[29] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *IEEEInternational Symposium on Integrated Network Management*. IEEE, 2019.

[30] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in p4," in *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[31] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie, "Netfc: enabling accurate floating-point arithmetic on programmable switches," 2021.

[32] R. B. Giuseppe Siracusano, "In-network neural networks," *SysML Conference 2018*, 2018.

[33] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*. Association for Computing Machinery, 2018.

[34] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.