HAL
open science

InREC: In-network REal Number Computation

Matthews Jose, Kahina Lazri, Jérôme François, Olivier Festor

To cite this version:

Matthews Jose, Kahina Lazri, Jérôme François, Olivier Festor. InREC: In-network REal Number Computation. IM 2021 - 17th IFIP/IEEE International Symposium on Integrated Network Management, May 2021, Bordeaux / Virtual, France. hal-03525052

HAL Id: hal-03525052

https://inria.hal.science/hal-03525052v1

Submitted on 13 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

InREC: In-network REal Number 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*—Current generation of Reconfigurable Match-Action Tables switches are highly programmable, able to support stateful operations and pipeline specifications using languages like P4. Nevertheless, these switches do not offer primitives to support real-valued operations on the data plane, thus requiring support from external servers or middleboxes to perform advanced operations. We introduce InREC, a system that extends the capabilities of programmable switches to support in-network real-valued operations using the IEEE half-precision floating point representation. It relies on decomposing real-valued functions into lookup tables taking into account the RMT model constraints to reach the right trade-off between accuracy and resource usage. InREC prototype on Barefoot Tofino switches demonstrates the efficiency of InREC for in-network computation of different types of operations and its application for in-network logistic regression models used for classification problems. Our evaluation of InREC shows that it is possible to implement complex in-network applications with high accuracy and low latency.**

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

I. Introduction

Software-Defined Networks introduced network programmability, enabling per-flow processing operations directly on the data plane within switches. Recent advances in hardware switch architectures allow flexible systems where data plane operations are enriched with the support of flexible per-packet processing capabilities through the execution of custom match-action table pipelines [1]. Proposals like domino [2] and open state [3] provide new programming languages and abstractions that give access to the persistent memory of a switch. This has resulted in mounting interest for deploying basic stateful applications like load-balancers [4], [5], DDoS detectors [6]–[8] and DNS servers [9], on the dataplane.

Operating networks involves the execution of complex operations such as data analytics, intrusion detection and encryption/decryption, currently available in dedicated appliances. These applications require computational capabilities of modern computers. Recent proposals have discussed methods for leveraging network elements to execute parts of such applications directly within the data plane. For instance, SwitchML [10] performs an in-network aggregation for a distributed machine learning cluster. IncBriks [11] is an in-network computation and caching system that lowers the request latency by 30%. Implementation of Rate Control Protocol (RCP) on a Reconfigurable Match-Action Tables (RMTs) switch [12] shows significant acceleration compared to server implementations. Despite these efforts, the lack of native primitives to support computations, equivalent to those provided by end servers, have resulted in modern day networks still being plagued with expensive middleboxes [12]–[14].

A programmable switch that supports real-valued operations would subsume many of the functionalities performed by these devices. Real numbers on computers can be encoded using predominantly fixed-point or floating-point representations. Computation using fixed point numbers requires less bit manipulation and can be implemented with simple arithmetic instructions such as those provided by P4 [15]. As a result, most work promoting in-network real-value computation relies on it [16]–[18]. Complex computational tasks including machine learning algorithms rely on floating-point numbers because they can represent a wide range of numbers. Even network applications, use several metrics such as average packet or byte counts that use large numbers. Besides, a distributed application that involves switches doing precomputation, and end-devices, applying complex algorithms, will benefit from a common representation to avoid re-encoding numbers.

Our objective is to empower the capabilities of RMT-based programmable switches with floating-point number operations. More complex operations could be offloaded to switches and so network function decomposition and orchestration can take benefit from InREC to distribute resource use and reduce latency thanks to in-network processing.

However, the limited resources on programmable switches [19] make it difficult to use traditional approaches to implement real-valued operations. There are also strong limitations on arithmetic and even restrictions on bit-wise operations. InREC promotes the use of LUTs (Lookup Tables) to support fast match between inputs and outputs of a real-valued function. A careful design and optimization of the LUTs is necessary to have a good trade-off between operation error and memory usage as a LUT cannot represent all possible inputs and is, by design, an approximation.

The main contributions of this paper are:

- a lookup table assisted implementation for a set of elementary operations,
- a procedure to automatically decompose a given real-valued function as a set of elementary operations including a set of optimizations to keep the LUTs minimal, in respect to the limited resources available on switches,
- an evaluation on a Barefoot Tofino switch to assess the accuracy and the overhead of InREC,
- the application of InREC for logistic regression, widely used in common classification problems.

The rest of the paper is structured as follows. Section II introduces the limitations of the RMT models. Starting from these limitations, the objective and the overview of InREC are given in Section III highlighting its two main blocks: elementary operations in Section IV and their automated composition in Section IV. The results of the evaluation are reported in Section VI. Related work is addressed in Section VII. Section VIII concludes the paper.

II. Real-numbers support limitations in switches

A. Re-configurable match-action tables or RMT

To identify the challenges for supporting real numbers on switches, we start by introducing the RMT architecture. RMT [19] is an architecture for programmable switches. It consists of 3 main blocks: (i) a parser that extracts fields from a packet, which, combined with metadata, forms a Packet Header Vector (PHV), (ii) a match unit that performs match operations (exact, ternary, longest prefix match ...) on a range of bits from the PHV and (iii) action units that can modify the PHV using a set of operations. The pipeline starts with the parser followed by a sequential series of tightly coupled match-action units. These units are present in resource clusters called stages. Each stage is allocated a fixed amount of memory (TCAM and SRAM) and CPU resources. Multiple stages can be combined to pool more resources. A separate metadata pipeline is used to carry information between stages including packet specific information, for example various timestamps associated with a packet.

To leverage the flexibility of programmable switches, languages like P4 [15] have been defined. P4 is a platform-agnostic language inspired from the RMT model to allow a programmer to define a packet processing logic including its parsing to extract header fields, match-action tables to have conditional actions based on field match and deparsing (packet reconstruction). However, P4 does not support native floating-point data type.

B. Background on floating-point real numbers

Floating-point [20] numbers are composed of three main parts: the sign, the exponent and the mantissa. Since we use half-precision numbers in this paper, their respective widths are : 1, 5 and 10 bits. Except for particular numbers (zero, infinite and subnormal numbers), the value of a floating point number is computed as follows:

$$(-1)^s \ * \ 2^{e-15} \ * M$$

with s the sign bit, e the exponent and M the mantissa

The IEEE 754 standard [20] requires that floating point numbers be *normalized*. This implies that the most significant bit of the mantissa is always set to 1 and hence can be omitted from the mantissa. Thus 1 is always prepended to the mantissa whenever a number is used in computation. Numbers between $[-1, 1]$ do not have this requirement and have their most significant bit set to 0. They are called sub-normal numbers.

C. Limitations of RMT based switches

The rigid structure of RMT and the implementation in silicon creates a number of physical restrictions. As introduced, the PHV includes metadata representing the switch states or variables. Since our objective is to support real numbers, they will need to be stored in the PHV. However, the latter consists of a large number of words or containers. These are usually 8, 16 or 32 bits wide. A single word can contain multiple small headers or a single large header that can span over multiple words. Hence, the different parts of a floating-point number (mantissa, exponent) are not necessarily aligned with the original word-length by nature and can span over multiple words. They therefore cannot be easily extracted individually to perform operations. In addition, switch restrictions limit to one per stage, the number of single arithmetic operations (such bit-field addition) that can be performed on a word.

In contrast, operations on floating-point numbers require several steps including extraction of the different parts and normalization. Adding two real numbers requires re-arranging the smallest number to have the two equal exponents, applying the binary addition over the two mantissas and finally normalizing the result by making the mantissa value to be between 1 and 2.

Implementing floating point number arithmetic as in general-purpose computers (defined in IEEE 754 [20]) on switches also requires many stages. Because memory resources cannot be shared between different stages of a pipeline, unused memory from a stage is wasted [21]. Therefore, supporting floating point arithmetic would be very costly, supposing enough stages are available.

Furthermore, Arithmetic–Logic Units (ALUs) of general-purpose computers include native pre-defined operations to simplify arithmetic like Floating-Point Multiply with Parallel Add [22] and Floating-Point Normalize [23], [24]. The extra design and cost to support make it harder for switch vendors to adopt.

III. Objective and approach overview

Our objective is to take a set of real-valued functions and create a switch specific processing pipeline using the P4 language and RMT-based hardware. The resulting pipeline should be minimal in terms of resource usage and latency. Due to challenges highlighted in the previous section, traditional hardware-based computational methods for floating point arithmetic are impossible.

InREC promotes the use of Lookup Tables (LUTs) as a workaround. A LUT [25] is a pre-computed table that maps a set of input values of a function to a set of output values. Using this method a computational operation can be translated into a match operation on the switch, hence circumventing most of the restrictions mentioned above. They also fit well with the common capabilities of a switch. Switches are designed to be efficient in two main tasks: classifying a network packet by means of match tables and then transmitting it. For instance, a switch with a throughput of 2.0 Tbps [1], [26] can classify 2 billion packets per second. On the other hand, a LUT performs

a computational task by means of classification and thus can leverage the high speed capabilities provided by switches.

For example, to compute $log(x/y)$, we can define a LUT for log and one for $/$ then use them sequentially. However, it is challenging to find the right trade-off between resource usage (both match-action tables and LUTs will be placed in SRAM) and accuracy depending of the number of entries in the LUTs. Indeed, a LUT represents a finite set of input values and approximates the other ones by rounding leading to loss of accuracy. Providing a manually-tailored specific pipeline to a particular function is the best way to optimize this trade-off. For instance, a single LUT can represent the full $log(x/y)$ operation but this also limits its applicability to other computational functions if not prohibiting them.

InREC consists of two main building blocks. The first one (section IV) defines a P4 processing pipeline for a set of elementary operations such as $log(x)$ or $x+y$. The second one relies on this set of elementary operations to automatically derive the processing pipeline of any compound function through multiple steps (detailed in section V):

1) combine elementary operations in a graph-representation of the function to compute;
2) identify the domain of variables and range of operations represented in the graph in order to better adjust the LUT entries, *i.e.* avoiding populating a LUT with useless out-of-domain input values;
3) aggregate several elementary operations in a single node in the graph if the resulting aggregated LUT to save some memory space;
4) transform the final graph into P4 instructions to implement the different operations represented as nodes based on the set of elementary operations (step 1) and optimizations done through steps 3 and 4;
5) do a final optimization by rearranging the composition of elementary operations to reduce output-input dependency between them because dependent operations cannot be executed in the same stage (output of a function cannot be reused as an input of another one in the same stage as explained in section II-C).

This process is executed on a controller and the resulting fully specified pipeline is installed on the switch.

IV. Elementary operations

A. LUT-assisted elementary operations

Elementary operations are the basic building blocks of any mathematical function (addition, multiplication,...). Elementary operations are classified as either *native* or *lookup table based*. Actually, native operations are limited by the hardware to the bit-wise operations (shift, concatenation,...) and the bit-field arithmetic operators. Furthermore, limitations described in section II-C still apply in particular on the number of operations that can be applied per stage on each word.

As an example, assuming $f(x,y) = x + y$, $x > 0$ and $y > 0$. To compute $f(x,y)$, x and y must have the same exponent values by shifting the bits of the smallest number. Once done, bit-field addition is applied on the mantissas followed by the normalization of the result. The number of shift operations to perform depends on the difference between the exponents e_x and e_y of x and y respectively. If $y < x$, the mantissa of M_y is shifted by $s = e_x - e_y$ bits, *i.e* $M_y >> s$. However, this is not allowed by the hardware we use (Tofino) because s cannot be a variable in a bit-shift operation. Alternatively, multiple hard-coded conditional statements based on the value of s can represent each possible shift. This is not efficient because the restrictions on operations described in section II-C forces each individual conditional statement to be in a different stage as they modify the same variable M_y. The only viable option is to use a LUT, $l(d,m)$, to implement the shift operation in regards of a compound input key composed of the exponent differences, d, and the mantissa m, so having $l(s, M_y)$ equivalent to $M_y >> s$. Followed by the shift operation, the mantissas are added, using the native addition operator provided by P4. The native addition operator is designed to function optimally on fields of 8, 16 or 32 bits which is different from the mantissa length (10 bits). So, the mantissa of the larger number M_x should be extracted and casted beforehand but only full words can be extracted (trying to get 10 bits leads to an alignment error). Therefore, the whole real number x is directly added with M_y to compute R. Unfortunately, x includes both exponent and sign bits which are removed from R using again a LUT to find the correct result.

As illustrated above, even a simple operation relying on the native addition operator must be LUT-assisted. Actually, all other elementary operations do not have a equivalent native operator. They must be pre-computed with a LUT but also rely on bit-wise operations for side manipulation (*e.g.* bit shift). Hence, the processing pipelines of elementary operations are designed beforehand and comprise a combination of lookup tables and native operations.

B. Constraints on LUT

The size of a LUT is fixed and the function inputs (keys in the LUT) can have a large combination of values. Its accuracy increases with smaller intervals between values. So, if a bounded representation of the function to compute exists, it should be used instead of the original one in order to group the number of possible inputs (constrained by the memory size) in the restricted intervals.

For example, $f(x) = log(x)$ is partially bounded because x must be positive. However, an equivalent and bounded form is $f(x) = n + log(\frac{x}{2^n})$ where n is the integer that makes $1 < \frac{x}{2^n} < 2$. With this transformation, values for $[log(\frac{x}{2^n})]$ are stored in the lookup table. The process is further simplified when using floating-point numbers, since they are stored in normalized form, or are expressed in the form $2^e * \sum_{n=0}^{10} 2^{-n} * man[n] + 2^e$ where e is the exponent and $man[i]$ the ith bit in the mantissa. Hence the mantissa is always bounded between 1 and 2, removing the cost of finding an equivalent bounded form. This is true for the example mentioned above and several other functions like e^x, $\sqrt{x}$ and $\frac{x}{2^n}$.

Therefore, when manually designing processing pipelines for elementary operations, looking for a bounded form is a good approach. Dependency between operations is also important to take into consideration. If operations are badly ordered, a single one can force another depending on its output to be put in the next stag for example, due to RMT restrictions on variable modification, and so increases the latency to compute the result. A dependent operation has to wait a full 12 CPU cycles before starting [27]. Carefully constructing pipelines to minimize these dependencies can help reduce the number of stages. Also, reducing the range of LUT keys would allow to represent with a finer granularity the value of the smaller range. For example, all $sin(x)$ values can be easily computed from $[0, \frac{\pi}{2}]$ rather than its period $[0, 2\pi]$. Pruning a LUT from low-varying intervals, *e.g.* an asymptotic functions, can save a lot of memory.

C. Summary

The restrictions on the use of native operators in current generation RMT switches makes a table-less implementation of elementary operations impossible. Our approach relies on a fusion of lookup operations, careful bit manipulation and native operations. Similarly to other works focused on fixed-point numbers [16]–[18], the design of the processing pipeline to compute these elementary operations is heavily manual even if the general idea is to reduce the operations to a bounded form and find a good trade-off between memory (LUT size) and the number of native operations and stages.

Besides, several variations of match/action units exist for each elementary operation. The most generic form is when input variables are not bounded. However, if an input variable is bounded (the domain is subset of $\mathbb{R}$), a more efficient variant exists. For example, the $log(x)$ elementary operation, consists of a single lookup followed by an addition operation. If x is bounded to an interval, the alternative implementation uses a single lookup table for all values in the interval.

Table I lists the implemented elementary operations and the resources they use (in their most generic form with unbounded variables). Except for $x+y$, no more than 3 stages are required [1], therefore limiting the latency to perform the operations. Besides, many of them are derived from others and so the same logic can be reused when defining the pipelines. For example, x/y is equivalent to $2^{log(x)-log(y)}$, so the building blocks of the log operation can be reused and adapted.

V. Automatic building of function processing pipeline

Although the design of processing pipelines for elementary operations is manual, the definition of the processing pipeline of an arbitrary compound function f is fully automated.

[1] $log(x)$ requires more than 3 stages if and only if the variables are not bounded. We can fix a larger negative number i_l as $log(x)$ is zero if $x < i_l$ and i_u as $log(x)$ is infinity if $x > i_u$

Elementary Function	#stages	#LUTs	#native op.
$log_2(x)$	4	3	1
$log_2(x)$ assuming no change after x=N	1	1	0
2^x	2	1	1
x/y	1	1	0
$sin(x)$	1	1	0
$\sqrt{x}$	3	2	0
$x+y$	3	2	1
$x*C$	1	1	0

TABLE I: Resources used by elementary operations

A. Step 1: A theoretical pipeline

The theoretical pipeline, if it exists, provides the composition of f in terms of elementary operations and captures the dependencies between these elementary operations. It is represented using a Directed Acyclic Graph (DAG) as illustrated in figure 1. The nodes in this graph represent an elementary operation or a function variable and the edges represent the dependencies between variables or operations that compose a function. If the DAG cannot be constructed (iterative and recursive functions), so the function cannot be calculated.

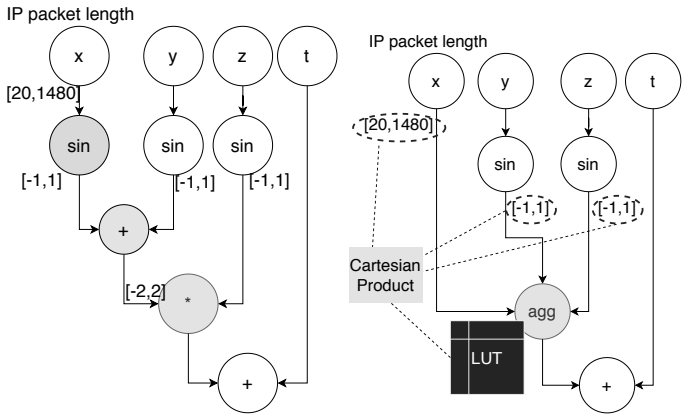
Variables can have contextual properties about their meaning. For example, a variable representing the network throughput cannot be higher than the bandwidth. Elementary operation nodes may also have mathematical properties as symmetry, a limited range (*e.g.* $sin(x) \in [-1, 1]$) or domain. Those properties are critical in determining if a node is bounded and when performing several optimizations in next steps.

B. Step 2: Check for bounds on domain and range

The goal of this step is to infer the domain and range of each node in the theoretical pipeline when possible to identify bounded nodes. Although all types of nodes can have a bounded range, we qualify as bounded nodes only the functions and so exclude the variables. Bounded ranges (of variables and elementary operations) identified in step 1 are propagated recursively. More precisely, a node is said to be bounded only if it is a function whose its domain is a subset of the real set, $\mathbb{R}$. This is only the case if the ranges of all its parent nodes (including variables) are bounded. Bounded nodes result also in a bounded range and, consequently, their child nodes are bounded.

As a result, all adjacent bounded nodes form a directed acyclic subgraph, called a bounded chain, to be used for performing aggregation and other optimizations afterwards. Several bounded chains can exist. In figure 1(a), the sin operation is used multiple times and has a bounded range $[-1, 1]$ by definition. Also, x is a variable equivalent to the IP packet length bounded by the Ethernet MTU. So it is between 20 and 1480 bytes. As a result, $sin(x) + sin(y)$ is a bounded node. The multiplication is also bounded because the inputs are a sin function and the previous addition that is bounded. So the subgraph composed of sin, $+$ and $*$ in grey is a bounded chain. This graph represents the bounded range of $sin(x) + sin(y)$ but it is not used. Only identified initial

bounded ranges (of variables and elementary operations) are necessary for the next step (aggregation).



(a) Identification of bounded ranges and nodes (in grey)

(b) Aggregation

Fig. 1: Aggregation of bounded chains in $f(x, y, z, t) = (sin(x) + sin(y)) * sin(z) + t$ where x is the IP packet length

C. Step 3: High level aggregation

If a bounded chain is identified in the previous step, both input and output values of their aggregated equivalent function are limited to a certain domain and range respectively. So, an aggregated LUT, agg, representing the whole chain is preferred because the inputs keys can be carefully selected. This is similar to finding an equivalent bounded form of an elementary operation. In addition, aggregating multiple operations into a single LUT avoids all data manipulation which are necessary between each operations to manipulate real numbers (extraction from the PHV, normalization...).

Obviously, the number of entries in agg depends on the bounds of incoming edges. The incoming edges of the agg node are the ones from functions being aggregated. To avoid a loss in accruacy, the agg input keys are derived from the cartesian product of initial LUT keys. Assuming agg to have E in-going edge, each in-going edge e is bounded $[l_e, u_e]$ and originally associated with $\#[l_e, u_e]$ LUT entries, the input keys of the newly created LUT for agg is all possible combinations of all original keys so:

$$\#[l_{agg}, u_{agg}] = \Pi_{e \in E}[l_e, u_e]$$

where Π is the n-fold Cartesian product.

In Figure 1(b), the identified bounded chain of figure1(b) is replaced by a single node *agg*. This corresponds to the $agg(x, y, z) = (sin(x) + sin(y)) * sin(z)$ function. The input keys of the newly built LUT are derived from the bounded ranges of functions or variables used as input. In this example, the inputs are *x*, *sin(y)* and *sin(z)* which are bounded to $[20, 1480]$, $[-1, 1]$ and $[-1, 1]$ respectively. The number of entries in the new LUT is so $\#[20, 1480]$*$\#[-1, 1] * \#[-1, 1]$.

Taking into considerations all the possible combinations from the keys of the original LUTs avoids a loss in accuracy. However, it can lead to a very large LUT which might need to be split into multiple stages. Doing such an aggregation is only viable if the number of stages to compute the aggregated function is not higher than the number of stages without aggregation. Otherwise, InREC does not aggregate the considered bounded chain. In that case, a smaller aggregation might occur. To do so, the sink (ending) node of the bounded chain is removed and the aggregation process is performed on the remaining part. This process is then applied recursively. In Figure 1(b), if the most global aggregation fails, the multiplication node is removed and the process tries to aggregate $sin(x) + sin(y)$ only.

D. Step 4: Substitution of elementary operations and simplification

Remaining elementary operation nodes in the graph (nodes not aggregated) are substituted for their actual instructions in the form of lookup tables and action unit primitives that have been pre-established [2] in regards of their inputs because several variants can be implemented depending on them (see section IV). However, further simplifications can be still applied.

First, chained operations involving log and division are checked to see if they cannot be arithmetically simplified. For instance, a/b equals $2^{log(a)-log(b)}$. If the graph contains a subgraph corresponding to $log(\frac{a}{b})$, this is equivalent to $log(2^{log(a)-log(b)}) = log(a) - log(b)$.

Second, some elementary operation nodes can have incoming edges with bounds when they have not been aggregated. These bounds are the limited range of input values; so all other values in the LUT can be discarded to save memory space.

When computing the entries of a lookup table for a node (representing a function), there are 4 factors to consider: the intervals from the domain of the function being used, the frequency of sampling in this intervals, restrictions on the range of the function and properties due to the nature of the intervals being used. The steps are as follows:

- Perform partial differentiation with respect to each independent variable for the function to detect slow or fast varying subranges (output values) using a threshold.
- Fix a sampling frequency for each interval and obtain a set of samples with respect to each independent variable. A higher sampling frequency is used for fast varying ranges. The sampling frequency can be tuned based on accuracy targets.
- Create the keys of the newly created LUT from the Cartesian product of the sets obtained before. Then, compute the value of the function at each key and create the set of lookup table entries. It is similar to the aggregation optimization but on a single function here.
- Prune the set of lookup table entries by removing keys that map to values considered as out of domain of its child node (given in step 1, when constructing the theoretical pipeline, in contextual properties of nodes).

[2]elementary operations have several pipeline representations that are manually designed and exist to leverage optimizations that arise due to the nature of the input variables, mathematical property or structure of the graph.

- Substitute all numbers in the pruned set with its equivalent floating-point representation and output the entries for the lookup table.

This procedure is performed using a high precision floating-point representation and is rounded to the nearest half-precision float number in the last step. The resulting table entries are encoded using the table entries paradigm provided by P4 for static elements.

E. Step 5: parallel processing

Parallel execution of independent operations reduces the number of stages needed and ensures maximum resource utilization in each stage while reducing the latency (directly depending of the number of stages to go through). Elementary operations that are commutative and occur consecutively can be executed in any order. This can be exploited to re-adjust the graph in such a way that consecutive commutative operations are independent, and merged later in a child node. For example, for $f(w, x, y, z) = w + x + y + z$, three stages are needed if an iterative add of x, y and z to w is performed to strictly follow the order of $+$ operators. Actually, this corresponds to apply a single addition per stage. However, computing $w + x$ and $y + z$ in parallel in the first stage before the final addition leads to two stages only.

VI. Evaluation

A. Implementation

InREC is implemented as a python program producing a P4 program compiled then onto a Tofino switch using the BareFoot SDE. In that case, other optimizations can be done but are hardware specific. The elementary operations can use more compact and minimal action units. Several of the primitives used by elementary operations had table units to assist certain key operations, like variable bit right shift. To address alignment issues (mantissas are smaller than a 16 bit word), we use identity hash functions provided by Tofino.

B. Setup and metrics

Our setup consists of a Stordis BF2556X-1T-A1F switch[3] connected to IBM BladeCenter HS22 7870 servers with an Intel Xeon X5660 2.80 GHz and 1Gbps Broadcom BCM5709S NIC. For the evaluation, 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.

Hence, the first metric to assess the accuracy of InREC is the (relative) error between the computed value by the switch and by one computed by the controller. Secondly, to assess its efficiency, the overhead implied by the computation is derived from the resource usage in the switch and the additional latency required for packet transmission. The evaluations are done with elementary operations tunes so that their maximum relative error is within 5%. This number is arbitrary and must be set according to a particular application in practice.

[3]https://shop.stordis.com/switches/bare-metalwhite-box-switches/stordis-48x-25gbe-sfp28--8x-100gbe-qsfp28-switch-bf2556x-1t-a1f

C. Error in a single elementary operation

In figure 2, we evaluate the error with respect to the memory used by the elementary operations. The exact amount of memory used depends on lengths of chosen keys and action values. For comparison, we took w (4, 6, 8 or 10 bits for match) bits representing the w most significant bits of mantissa for the lookup operations. The mantissa is thus truncated when $w < 10$ leading to a loss of accuracy but a lower LUT size. Thus a table contains 2^w entries.

For each elementary operation, the average relative error is computed while varying x between 0 and 10000 and w from 4 to 10 bits. The error decreases when w increases. This is especially true for operations with a high rate of changes like 2^x. To maintain an error below 5% w must be at least 8 for sin and 6 in all other cases. An ideal lookup table would contain all possible values of the mantissa and is represented by the case where $w = 10$. In that case, the error is due to other operations, like the addition operation, used after the lookup.

As described in Section IV, the addition is a lookup-assisted. It is done using the native bit-field addition operator bu LUTs are used for performing bit-wise shift and computing the exponent values of the final result. Because these tables replace operations the switch does not support, w must be equal to 10 bits. For example, if shifting the mantissa by n bits (10 at most) is not possible, the addition cannot be processed further. For $w = 10$, the relative error is around 10^{-4} and is omitted in this figure. The value w directly impacts the size of LUT in SRAM that is discussed later in Section VI-F.

D. Error propagation

In Figure 3, we performed an elementary operation multiple times consecutively. The LUT size have been chosen such that the average relative error is less than 1% each time a single operation is performed and x varies between 10 and 2000.

As expected, the error increases with the number of times we perform the elementary operation except for the $sin(x)$ operation because its range is bounded and so the input values are restricted to $[-1, 1]$ after the first iteration. Obviously, the average relative error when performing an operation in a fixed interval is the same. This is different from the other operations where the result increases or decreases in magnitude, changing the average error observed at each iteration.

We can conclude that the error of an operation, when performing several iterations, depends on the properties of the underlying operations and has a maximum bound of 10% at 4 iterations. On average it is less than 1%, which is enough for most applications.

E. Latency overhead

Figure 4 shows the average processing time of the elementary operations. We compare 2 scenarios:

- without routing: once the operation is performed, the packet is simply sent back to its source port;
- with routing: once the operation is performed, the destination port is set using 3 lookup tables that use a combination of IP header fields as keys.

Fig. 2: Impact of the LUT key size on the relative error of elementary operations with C a constant.

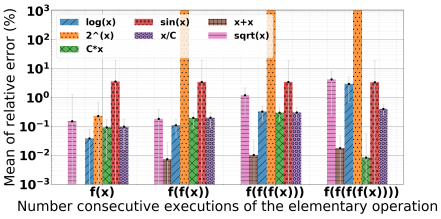


Fig. 3: The relative error when computing an elementary operation successively .

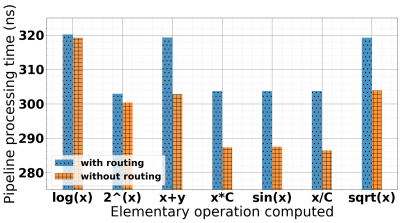


Fig. 4: Processing time by the switch for elementary operations .

Assuming no routing, log requires the most processing time followed by the $sqrt$ and $+$. This is because the processing time is mostly impacted by the number of stages used in the pipeline. The log operation requires 4 stages in the pipeline while a single one is necessary for $*$, $/$ and sin.

The difference in processing time between both scenarios are smaller for log than for $*$,$/$ and sin because routing tables share the same pipeline stage as the LUTs for log. Moreover, routing tables use the TCAM memory available in the stages that is considerably faster and hence do not add much more overhead even in other cases. Especially, in the case of $*$, $/$ and sin, the operation only uses a single stage compared to the routing tables that take 3 stages, leading to a three-stages pipeline. An exception is the $+$ operation. It requires a significant amount of action units competing with those needed to perform actions of the routing tables pushing the tables to other stages with more available memory.

Nonetheless, the additional latency involved by InREC for a single operation is below 15ns (nanoseconds). Even if multiple operations are applied, this overhead is well below the end-to-end latency of packets whose order of magnitude is in milliseconds. Also, the worst case in Figure 4 is 320ns for $log(x)$ which is equivalent to 3,125,000 packets per second.

F. Ressource overhead

Figure 5 shows the average usage of SRAM[4], pipeline stages and Very Long Instruction Word (VLIW). The average value is computed across the total available quantity of a specified resource. The sizes of LUTs are adjusted in size such that the induced relative error is lower than 1%.

The VLIW instructions is the memory used to define action units and the instructions to be executed by them. $log(x)$ uses the most resources (33% of the total number of stages and 8% of the total SRAM available) in its most generic form. The bulk of the SRAM usage is similar of $x + y$, around 8%.

Elementary operations relying on native operators or bit manipulation in their implementation, like $x + y$ and $log(x)$ (for performing addition), require more VLIW instructions. Most of elementary operations are fully lookup table based and have simple action units that are mostly assignment operations. Nonetheless, the capabilities of InREC (number of real-valued operations) are constrained by the number of stages and the available SRAM because the number of VLIW instructions remain low ($< 1\%$) in all cases. Thanks to its design, InREC reduces simultaneously the requirements on these resources as multiple operations initially put on different stages can be combined in a single LUT (high-level aggregation in step 3) and so in a single stage. Furthermore, the number of input keys is also optimized in step 4 to reduce the LUT size and so the SRAM used. Finally, parallel processing in step 5 reduces the number of stages. As a result, reaching a good precision is possible with a low overhead on resource usage.

G. Use Case: logistic regression for networks

To show the viability of our approach, we implemented a logistic regression model that is mainly used for classification, for example to detect phishing [28] or malicious URLs [29]. Several of these applications are deployed on the edge of a network and require a low latency response time. A logistic regression model can be computed in its equivalent form as:

$$f(x_1, x_2, x_3, .., x_n) = \frac{1}{1 + 2^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}} \quad (1)$$

The training of the model, *i.e.* learning the β parameters, is pre-computed and only its application is run on this switch as this would be the case for a real deployment, *e.g.* to detect malicious traffic flows at the edge of networks. Equation 1 is derived from usual logistic regression but uses bases 2 rather than base e and uses 4 variables with constants as $\beta_0 = 0.1$, $\beta_1 = 0.1$, $\beta_2 = 0.1$, $\beta_3 = 0.1$ and $\beta_5 = 0.1$.

As highlighted, the complexity of the computation depends on the number of variables but increasing their number from 2 to 4 does not impact the accuracy as noticed in figure 6 (error lower than 2.5×10^{-4}). For testing purpose, we classified 50 random data points in four dimensions either with InREC or with Scikit-Learn[5] in python with a common computer and verified that the output is the same. So, the very low error induced by InREC does not impact the final classification.

The latency increases with the number of variables: 304.1, 314.64 and 320.3 nanoseconds in average for 2, 3 and 4 variables respectively. The higher difference between 2 and 3 variables is due to one more stage whereas the 4th variable

[4]The total SRAM size of the switch cannot be divulged due to a non disclosure agreement

[5]https://scikit-learn.org/

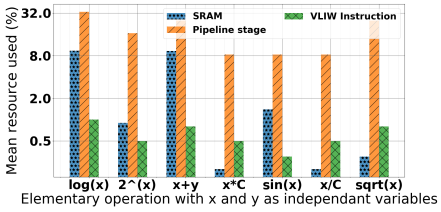


Fig. 5: Pipeline resources used by each elementary operation.

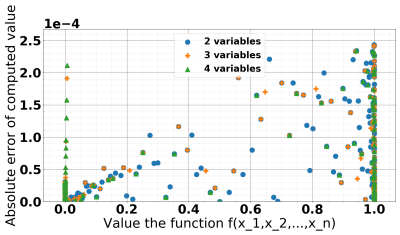


Fig. 6: Logistic regression error with 2, 3 or 4 independent variables.

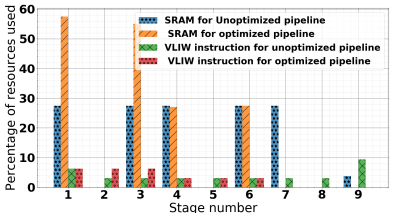


Fig. 7: Resource usage for logistic regression with 4 independent variables.

calculation can be done in parallel. This confirms that our design choices to optimize the pipeline, including aggregation and simplification as described in section V, are valuable. For comparison, the naive pipeline as introduced in step 1 in section V-A is also applied. Figure 7 shows that the optimized pipeline uses three less stages than the naive one considering four variables. Actually, the three last stages performed in the naive pipeline have been combined in a single LUT thanks to our aggregation scheme. In practice, the number of stages in the switch we used limits the maximum number of variables for logistic regression to four for the naive pipeline whereas our technique allows to use up to six variables.

Assuming the computation done in an external server connected to the switch, the overall latency is 4.733 milliseconds while it is reduced to 2.367 milliseconds in our case. So, the increased computation time due to the switch architecture in comparison to a modern computer is largely compensated by the link latency even with a single hop. Therefore, InREC can be used in practice to support advanced monitoring functions like logistic regression with a high accuracy and low latency.

VII. Related Work

A. In-network computing

Multiple works propose to leverage in-network computing capabilities. Monitoring applications that use measurement constructs like counters and sketches are one of the most popular use cases [30], [31] but state machine representations have also been proposed [32], [33]. Other examples target in-network key-value store applications [11].

In the scope of this paper, various methods have been proposed to perform real valued operations. Naveen Kumar et al [12] implemented RCP and calculate fair-rate using in-network division. In [17], the authors offload an entropy model for DDoS detection to the dataplane, using a combination of sketches and LUTs to perform real valued operations. Recently, [18] focuses on entropy calculation.
While all these proposals provide methodologies to optimize specific functions, they do not consider the general case of any real-valued function as InREC promotes. Furthermore, none of the work considers floating point real numbers.

In [34], the authors explore the possibility of deploying machine learning classification algorithms in-network, hoping to one day have switches perform some of the task being done on servers. They highlight the lack of floating point operations in current hardware. Adding hardware accelerator for machine learning has been also proposed [35].

B. Pipeline Design and Resource Optimization

One major challenge of InREC was to build efficient pipelines for real valued operations. Magellan [36] takes a high level program and builds a compact pipeline that is optimized for a given hardware. [37] looks at the advantages of a greedy approach compared to a linear programming approach for optimizing a pipeline. InREC uses a combination of techniques to create pipelines but, most of all, is specific to floating point numbers. Thus, specific optimizations like aggregation of operations or reducing the LUT size with bounded operations can be performed unlike the general case. While our work is focused on optimizing the processing on a single switch, the literature is vast about the decomposition and optimal placement of a program. SNAP [38] uses xFDD for decomposing and analyzing dependencies of state variables in a program and further optimally places them on the dataplane by solving a mixed integer linear program. Merlin [39] uses service chains and optimizes traffic using an MILP (Mixed-integer linear programming). InREC is complementary to these approaches as it brings the possibility to perform real valued operations in a switch, so the constraints of those operations (resource needed and achieved accuracy) could be integrated into a global optimization problem. Due to resource struggle, there are several papers that propose adding external memory [40] and, in some cases [41], with custom functions.

VIII. Conclusion

InREC is a new approach to support real-valued functions on RMT based switches. It cleverly combines native operations and LUTs to construct a minimal switch specific pipeline guaranteeing an acceptable loss of accuracy in the computing. This pipeline is expressed as P4 logic and sent 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 InREC a viable approach to support complex functions and algorithms. In future work, we plan to release InREC as an open-source software. We will also leverage it to support functions with an extreme complexity, requiring the decomposition of the latter in multiple switches. We will also investigate how to minimize the error based on the available resources.

References

[1] Arista, "Arista 7170 multi-function programmable networking." [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf

[2] 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.

[3] 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.

[4] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM Symposium on SDN Research (SOSR)*, 2016.

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

[7] 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.

[8] 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.

[9] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4dns: In-network dns," *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[10] 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*. KAUST, 2019.

[11] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," *ACM SIGOPS Operating Systems Review*, vol. 51, april 2017.

[12] 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.

[13] 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.

[14] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment," *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] IEEE, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, 2008.

[21] 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.

[22] C. M. University, "Floating-point parallel multiply add instruction." [Online]. Available: http://qcd.phys.cmu.edu/QCDcluster/intel/vtune/reference/6400234.htm

[23] Princeton, "Floating point, branching, and assembler directives." [Online]. Available: https://www.cs.princeton.edu/courses/archive/spring04/cos217/lectures/IA32-II.pdf

[24] A. K.ManolopoulosaD, ReisisaV., "An efficient multiple precision floating-point multiply-add fused unit," *Microelectronics Journal*, vol. 49, march 2016.

[25] C. d. B. S.D. Conte, *Elementary Numerical Analysis: An Algorithmic Approach Updated with MATLAB*, ser. Classics in Applied Mathematics. SIAM-Society for Industrial and Applied Mathematics, 2018.

[26] Stordis, "Stordis bf6064x-t." [Online]. Available: https://shop.stordis.com/media/files_public/e14aae34be8cf7cb19aa4f849e735d2b/stordis\%20bf6064x-t-a2f\%20datasheet.pdf

[27] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *NSDI*. USENIX Association, 2015.

[28] S. Garera, N. Provos, M. Chew, and A. D. Rubin, "A framework for detection and measurement of phishing attacks," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode*. Association for Computing Machinery, 2007.

[29] V. Anandkumar, "Malicious-url detection using logistic regression technique," *International Journal of Engineering Business Management*, vol. 9, Dec 2019.

[30] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[31] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 576–590.

[32] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, "Defeating protocol abuse with p4: Application to explicit congestion notification," in *IFIP Networking 2020*, 2020.

[33] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli, "Xtra: Towards portable transport layer functions," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1507–1521, 2019.

[34] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, 2019.

[35] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2019.

[36] A. Voellmy, S. Chen, X. Wang, and Y. R. Yang, "Magellan: Generating multi-table datapath from datapath oblivious algorithmic sdn policies," in *ACM SIGCOMM Conference*, ser. SIGCOMM '16. ACM, 2016.

[37] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, May 2015.

[38] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," *ACM SIGCOMM Conference*, 2016.

[39] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for managing network resources," *IEEE/ACM Transactions on Networking*, vol. 26, 2018.

[40] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation," *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[41] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Symposium on Software Defined Networking (SDN) Research*. Association for Computing Machinery, Inc, Jun. 2015.