UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

UNIVERSITY OF FRIBOURG

BACHELOR THESIS

# Two layer RMIs on P4 capable network switches

*Author:*
Lucas Bürgi

*Supervisor:*
Prof. Dr. Philippe Cudré-Mauroux

*Co-Supervisor:*
Dr. Alberto Lerner

January 01, 2022

eXascale Infolab
Department of Informatics

Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland
phone +41 (26) 300 84 65 • diuf-secr@unifr.ch • www3.unifr.ch/inf

# Abstract

Lucas Bürgi

*Two layer RMIs on P4 capable network switches*

Write the thesis abstract here. Should be between half-a-page and one page of text, no newlines.

**Keywords:** SOSD, Learned index structures, RMI, P4, Network programmability, BMv2, Mininet, IEEE754, Floating point arithmetic

# Contents

# List of Figures

# Chapter 1

# Introduction

This work started with a first task of setting up the SOSD benchmark by Kipf et al., 2019 with the premise in mind that learned index structures potentially can outperform traditional index structures. Part of this process would also be to recheck and verify the promised results on a local system. After comparable results to the original Kipf et al., 2019 paper the idea of taking advantage of the benefits that learned index structures offer and fusing these with the now more and more establised network programmability offered by P4 became an essential goal of this work.

## 1.1 Motivation

A network device is usually very good and very fast at specific simple tasks. In other words it can treat an enourmous amount of packets in a very small amount of time. On the down side of things though it is limited in what operations it can offer and how complex a composition of them can get as well as the amount of memory that is at disposal. This becomes interesting when looking the fact that learned index structures tend to work in a way that they have a rather complex learning phase, where quite some time is spent on examining the data and its nature before then storing the gathered information in some form. The assumption is that the actual lookup then, due to the previous processing, should now be relatively simple and especially computationally cheap. Further learned index structures are often capable of adapting the amount of memory they consume depending on the requested prediction accuracy defined in the learning phase. This results in a memory to prediction accuracy tradeoff that could potentially be interesting for devices with limited memory capabilities. Another important aspect is that lately hardware acceleration through the efficient usage of secondary devices with some sort of computational power (like switches, NIC's, SSD drives, etc) has become a key aspect for making (distributed) systems faster. In terms of networks this was mainly allowed through a wider adoption of the P4 language.

In general the motivation for this work is to potentially speed up all sorts of operations that require lookups on sorted data by allowing them to satisfy their requests directly through the network. With that in mind this work tries to explore the actual feasibility and possibilities that a suitable learned index structure could offer when implemented on a P4 capable network device.

## 1.2 Thesis structure

In chapter 2, an overview of what techniques and resources were used and tested is given, as well as which ones and why some of them where finally further pursued.

In chapter 3, a potential implementation of a two-layer RMI on the synthetic BMv2 switch is proposed as well as an evaluation on how far away current physical switches are from what would be needed.

In chapter 4, an adaptation of the reference RMI implementation by Marcus, Zhang, and Kraska, 2020 to be able to generate P4 source code files is presented.

In chapter 5, some additional measurements are made and combined with hypothetical calculations of what switches are theoretically capable of doing today to try to get an approximate idae of how fruitful this work could be in the future.

Finally in chapter 6 this thesis is closed by stating our conclusions and looking at potential future work.

# Chapter 2

# Background

This chapter is about what steps were involved to determine which learned index structure was worth the effort of further exploration. As a starting point, this was achieved by running and evaluating the SOSD benchmark from Kipf et al., 2019 on a local system. Further, as a second indicator, by looking at what different learned index strucures make use of in terms of complexity during their learning phases or more importantly what level of complexity and what programming concepts are needed during a lookup.

Finally this chapter looks at the basic capabilities and limitations of the P4 language, especially in regard to what the language offers that could be used as an advantage for learned index structures or on the other hand which important concepts are potentially missing.

## 2.1 SOSD

In this first section the goal is to analyse the results of the SOSD benchmark. For doing so my work consisted of understanding the code by myself as well as comparing our local results with what is given by the authors. To do so the initial goal was to reproduce Table 2 from Kipf et al., 2019, which gave the result shown in 2.1. Important to mention for these figures is that to obtain these results, the benchmark runs different pareto runs for each algorithm on each dataset from which the optimal run with respect to the three metrics lookup time, build time and index size is selected. At this given point in time it was important for this work to take note of the fact that learned index structures effectively can outperform traditional index structures when tuned properly. This means that for now the assumption that the ability to train an algorithm on a given dataset can translate to faster lookup times is true. With respect to the other metrics, this results in a tradeoff between either slower lookup times but no build time or instead spending time upon build as pointed out in 2.2 to then gain with faster lookups.

## 2.2 Learned index structures

In the benchmark there are currently three main competitors that belong to the category of learned index structures. Namely these are RMI, RS and PGM. RMI (Recursive Model Indexes) is proposed in Kraska et al., 2018 and specificly implemented for the benchmark in Marcus, Zhang, and Kraska, 2020. RS (Radix Spline) is proposed and implemented by the original authors in Kipf et al., 2020. Finally PGM (Piecewise Geometric Model) is proposed in Ferragina and Vinciguerra, 2020.

| | ALEX | ART | BTree | BinarySearch | FAST | IBTree | RBS | PGM | RMI | RS | Wormhole |
|---|---|---|---|---|---|---|---|---|---|---|---|
| books_200M_uint32 | 316.16 | NaN | 591.732 | 893.826 | 559.118 | 528.544 | ***139.448*** | 314.942 | 190.161 | 198.699 | NaN |
| lognormal_200M_uint32 | 333.41 | NaN | 584.522 | 934.58 | NaN | 514.678 | 223.478 | 258.571 | 138.051 | ***126.64*** | NaN |
| normal_200M_uint32 | 290.344 | NaN | 601.172 | 955.37 | 556.347 | 439.418 | 124.874 | 186.973 | ***91.8032*** | 99.0789 | 1051.77 |
| uniform_dense_200M_uint32 | 294.567 | NaN | 616.39 | 895.077 | 558.462 | 370.233 | 110.617 | 98.1507 | 83.8801 | ***80.8129*** | 1024.61 |
| uniform_sparse_200M_uint32 | 328.36 | NaN | 591.211 | 898.189 | NaN | 428.702 | ***128.604*** | 299.459 | 160 | 198.77 | NaN |
| books_200M_uint64 | 312.221 | 449.365 | 649.398 | 890.887 | 672.928 | 526.815 | ***137.643*** | 364.815 | 186.805 | 204.429 | 1052.57 |
| books_400M_uint64 | 360.351 | 470.969 | 702.37 | 1017.28 | 753.39 | 570.739 | ***165.432*** | 435.238 | 228.205 | 223.122 | 1160 |
| books_600M_uint64 | 390.46 | 495.147 | 738.915 | NaN | 817.406 | 595.732 | ***187.076*** | 465.427 | 245.369 | 242.549 | 1223.37 |
| books_800M_uint64 | NaN | 508.648 | 765.111 | NaN | NaN | 615.326 | NaN | 474.086 | 265.94 | ***252.299*** | NaN |
| osm_cellids_200M_uint64 | 527.39 | 527.905 | 648.883 | 891.171 | 676.749 | 696.41 | ***265.75*** | 490.408 | 321.225 | 285.03 | 1057.17 |
| osm_cellids_400M_uint64 | 575.163 | 562.773 | 700.965 | 1014.34 | 759.154 | 737.453 | ***301.552*** | 529.348 | 359.27 | 312.589 | 1153.61 |
| osm_cellids_600M_uint64 | 602.256 | 573.829 | 739.257 | NaN | 820.099 | 781.126 | ***326.518*** | 568.718 | 383.993 | 327.995 | 1214.57 |
| osm_cellids_800M_uint64 | NaN | 584.587 | 762.652 | NaN | NaN | 810.072 | NaN | 591.456 | 407.657 | ***352.552*** | NaN |
| fb_200M_uint64 | 513.211 | 541.836 | 651.078 | 893.551 | 674.479 | 554.279 | 923.133 | 447.051 | ***262.245*** | 630.806 | 1039.77 |
| wiki_ts_200M_uint64 | 367.394 | NaN | 665.586 | 920.712 | NaN | 611.645 | ***160.364*** | 384.602 | 205.337 | 259.339 | NaN |
| normal_200M_uint64 | 299.804 | 520.138 | 654.013 | 947.358 | 685.591 | 525.957 | 563.156 | 231.13 | 109.144 | ***84.6746*** | 1184.15 |
| lognormal_200M_uint64 | 285.402 | 436.82 | 653.134 | 891.473 | 673.484 | 444.378 | 131.5 | 184.301 | ***88.6153*** | 92.9393 | 1047.86 |
| uniform_dense_200M_uint64 | 299.243 | 385.001 | 646.252 | 894.231 | 685.664 | 362.695 | 109.699 | 96.8831 | 82.639 | ***77.1423*** | 1033.54 |
| uniform_sparse_200M_uint64 | 359.365 | 375.967 | 651.722 | 886.51 | 679.105 | 420.886 | ***124.665*** | 297.306 | 161.805 | 198.195 | 1052.56 |

FIGURE 2.1: **SOSD lookup times (ns).** Lookup times produced by the benchmark when installed on our local test machine, selecting the best performing run with respect to the three metrics lookup time, build time and index size among all pareto runs.

| | ALEX | ART | BTree | BinarySearch | FAST | IBTree | RBS | PGM | RMI | RS | Wormhole |
|---|---|---|---|---|---|---|---|---|---|---|---|
| books_200M_uint32 | 5613541631 | NaN | 35491027 | ***0*** | 510288530 | 1626830967 | 997934052 | 11717402216 | 35644110755 | 4452743109 | NaN |
| lognormal_200M_uint32 | 70224644028 | NaN | 79718462 | ***0*** | NaN | 431489072 | 447882540 | 2613987801 | 22948005419 | 842604354 | NaN |
| normal_200M_uint32 | 34408112431 | NaN | 79529861 | ***0*** | 494151502 | 1609719961 | 568111845 | 6254877325 | 20326382313 | 1682689337 | 1129 |
| uniform_dense_200M_uint32 | 15968586795 | NaN | 84369704 | ***0*** | 507362112 | 1608830540 | 1103820491 | 13267855213 | 19937531313 | 1810956646 | 931 |
| uniform_sparse_200M_uint32 | 51328704987 | NaN | 41722287 | ***0*** | NaN | 435032689 | 2202372551 | 10641256426 | 33052197928 | 5448896265 | NaN |
| books_200M_uint64 | 5362716211 | 668951282 | 20722703 | ***0*** | 7945521 | 216257132 | 1236351861 | 10787330586 | 39078266240 | 4344139374 | 1449 |
| books_400M_uint64 | 12783280618 | 3255052314 | 21205570 | ***0*** | 8362200 | 307440615 | 1606847039 | 19402312806 | 69812309062 | 7773094039 | 1200 |
| books_600M_uint64 | 20957946782 | 4830253613 | 7894538 | NaN | 71821410 | 1243889626 | 1972353587 | 41252476632 | 1.00052E+11 | 11868144698 | ***1696*** |
| books_800M_uint64 | NaN | 5976081912 | ***44093720*** | NaN | NaN | 1529892294 | NaN | 50816509294 | 1.29437E+11 | 12769004123 | NaN |
| osm_cellids_200M_uint64 | 14363124428 | 1559089625 | 21889086 | ***0*** | 7973213 | 155961592 | 627624363 | 13537160951 | 34199826191 | 4532354195 | 1159 |
| osm_cellids_400M_uint64 | 28349410911 | 3347032645 | 20946210 | ***0*** | 8235786 | 246038304 | 938409968 | 27056319845 | 63307640082 | 7219472260 | 1468 |
| osm_cellids_600M_uint64 | 41100065731 | 5630119627 | 8019769 | NaN | 6588704 | 369011763 | 1265070334 | 39949853443 | 93167288241 | 10547667514 | ***1600*** |
| osm_cellids_800M_uint64 | NaN | 2759561032 | ***10817514*** | NaN | NaN | 632150814 | NaN | 49530400993 | 1.20586E+11 | 14131136703 | NaN |
| fb_200M_uint64 | 9596065 | 615466791 | 20580318 | ***0*** | 8382451 | 155495889 | 302993747 | 12379131817 | 34612057333 | 2156308076 | 1116 |
| wiki_ts_200M_uint64 | 13848598291 | NaN | 69862145 | ***0*** | NaN | 216274205 | 1469597746 | 6795575268 | 41867240037 | 2629094469 | NaN |
| lognormal_200M_uint64 | 19810041014 | 585827372 | 2425054 | ***0*** | 7944107 | 234811903 | 469658051 | 12220160016 | 25866654986 | 2032042953 | 1270 |
| normal_200M_uint64 | 4109215022 | 640488227 | 2304952 | ***0*** | 8023211 | 1930349469 | 500091969 | 12477923781 | 31093744311 | 1822729292 | 1176 |
| uniform_dense_200M_uint64 | 16617866257 | 5072178119 | 20650827 | ***0*** | 8445758 | 1921241901 | 986645116 | 13298443626 | 16189861984 | 1840515221 | 1173 |
| uniform_sparse_200M_uint64 | 9690558684 | 520701367 | 20065459 | ***0*** | 8303068 | 382554533 | 2443645115 | 11345964171 | 33143358893 | 6001268760 | 1154 |

FIGURE 2.2: **SOSD build times (ns).** Build times produced by the benchmark selected among pareto runs the same way as described above.

### 2.2.1 RMI: Recursive Model Indexes

RMI Kraska et al., 2018 is a learned index structure that is based on the idea that different models fit certain data better. By having a number of different models to choose from during the learning phase and by allowing to stack different models on top of each other in different layers, RMI should adapt well to mostly any given shape of sorted data. In the context of the benchmark as well as in the context of this work, RMI's are fixed to two layers since this proved to be most efficient for most datasets and also reduces complexity for further chapters. In the concrete implementation in Marcus, Zhang, and Kraska, 2020 CDFShop generates C++ source files that contain parameters as well as the adapted code depending on which models where chosen on which layer. Generally the input key is fed into a first layer, which generates an index that then serves as a starting point for feeding the next layer, and so on, until finally a last layer retrieves the estimated key's position together with a stored error margin.

Notable for a potential P4 implementation here is that RMI is using floating point arithmetic, solely focussed on using the floating point FMA instruction. This immediately makes it a big challenge to think about implementing RMI in P4 but leaves some hope in the sense that if an FMA operation together with some simple form of floating point arithmetic could be implemented in P4 then RMI quite quickly would become realistic on a P4 device.

| | ALEX | ART | BTree | BinarySearch | FAST | IBTree | RBS | PGM | RMI | RS | Wormhole |
|---|---|---|---|---|---|---|---|---|---|---|---|
| books_200M_uint32 | 433219884 | NaN | 87075880 | 0 | 106666880 | 2421989256 | 1073741828 | 70640624 | 402653216 | 1397279468 | NaN |
| lognormal_200M_uint32 | 3464837240 | NaN | 174150608 | 0 | NaN | 75693848 | 1073741828 | 73184 | 17563648 | 17220924 | NaN |
| normal_200M_uint32 | 3464521080 | NaN | 174150608 | 0 | 106666880 | 2421989256 | 1073741828 | 62784 | 3145744 | 443772 | 1367184 |
| uniform_dense_200M_uint32 | 3464409164 | NaN | 174150608 | 0 | 106666880 | 2421989256 | 1073741828 | 304 | 24592 | 172 | 1367184 |
| uniform_sparse_200M_uint32 | 3464426636 | NaN | 87075880 | 0 | NaN | 75693848 | 1073741828 | 41592384 | 402653200 | 1092850956 | NaN |
| books_200M_uint64 | 576049120 | 156030056 | 58008568 | 0 | 3571712 | 25202664 | 1073741828 | 45175940 | 402653216 | 974927880 | 1562496 |
| books_400M_uint64 | 1152069800 | 261696176 | 58008568 | 0 | 3571712 | 25202664 | 1073741828 | 49306400 | 402653216 | 1453207928 | 2312128 |
| books_600M_uint64 | 1727965816 | 498050232 | 21753608 | NaN | 21428992 | 302370032 | 1073741828 | 497978400 | 402653216 | 1854804488 | **4687488** |
| books_800M_uint64 | NaN | 597290360 | **116016232** | NaN | NaN | 403151800 | NaN | 638247040 | 402653216 | 1739250184 | NaN |
| osm_cellids_200M_uint64 | 612210912 | 155154112 | 58008568 | 0 | 3571712 | 12603400 | 1073741828 | 118835100 | 402653216 | 1220776568 | 1562496 |
| osm_cellids_400M_uint64 | 1194629024 | 310601720 | 58008568 | 0 | 3571712 | 12603400 | 1073741828 | 262357400 | 402653216 | 1087997836 | 3124992 |
| osm_cellids_600M_uint64 | 1785032456 | 466463216 | 21753608 | NaN | **2678912** | 18903032 | 1073741828 | 425168960 | 402653216 | 1196538236 | 4687488 |
| osm_cellids_800M_uint64 | NaN | 618537904 | **29004872** | NaN | NaN | 50401192 | NaN | 274750980 | 402653216 | 1358377432 | NaN |
| fb_200M_uint64 | 8998528 | 69335392 | 58008568 | 0 | 3571712 | 12603400 | 16777220 | 43774460 | 402653200 | 7364228 | 1562496 |
| wiki_ts_200M_uint64 | 1155476352 | NaN | 116016232 | 0 | NaN | 25202664 | 1073741828 | 14269460 | 402653216 | 81779860 | NaN |
| lognormal_200M_uint64 | 1168820672 | 129446800 | 7251896 | 0 | 3571712 | 50401192 | 1073741828 | 109280 | 100663312 | 537116104 | 1562496 |
| normal_200M_uint64 | 576112536 | 136087424 | 7251896 | 0 | 3571712 | 3225127816 | 1073741828 | 78080 | 20512 | 306344 | 1562496 |
| uniform_dense_200M_uint64 | 4607268944 | 1618824944 | 58008568 | 0 | 3571712 | 3225127816 | 1073741828 | 380 | 24592 | 180 | 1562496 |
| uniform_sparse_200M_uint64 | 1151819896 | 135796784 | 58008568 | 0 | 3571712 | 100794136 | 1073741828 | 52020340 | 402653200 | 1895162596 | 1562496 |

FIGURE 2.3: **SOSD index sizes (bytes).** Index sizes produced by the benchmark selected among pareto runs the same way as described above.

### 2.2.2 RadixSpline: A Single-Pass Learned Index

RS Kipf et al., 2020 is built on top of the idea of fitting a linear spline to the CDF of some sorted data. Different spline segments are indexed and to each segment two spline points are stored. Upon lookup the learned index tries to locate the responsible spline segment and performs a linear interpolation between the two spline points to find the estimated key position.

For P4 programmability important to note here is that RS stores the spline points in floating point format and upon lookup performs mathematical operations on these floating point numbers. The most notable operations in this context are the calculation of the slope of a spline segment which involves a floating point division and finally the interpolation itself which involves a floating point FMA instruction. This again makes it quite a big challenge to even start thinking about an RS implementation in P4. In comparision to RMI a big downside of RS is the part where the lookup code tries to locate the responsible spline segment. To make sure the correct segment is chosen either a linear search on small ranges or a binary search on bigger ranges is used. Both search concepts involve conditional iteration over dynamic data and are with that to my current knowledge far from easily feasible in P4.

### 2.2.3 PGM: The Piecewise Geometric Model index

PGM Ferragina and Vinciguerra, 2020 is a pure learned index structure that tries to create a piecewise linear approximation (PLA) that maps keys to their approximate positions in the data with at most $\epsilon$ distance to their actual position. By applying this approximation recursively onto itself multiple levels of PLA's are built such that efficient search becomes possible.

The given lookup code in Ferragina and Vinciguerra, 2020, due to it's recursive nature at creation, needs to iterate over all levels of PLA for each lookup, which already marks a first challenge. To make things even harder in terms of P4 programmability on each level of PLA either linear search for small ranges or binary search for bigger ranges is used to determine which PLA is responsible for a given key on the next level. This then results in a very similar situation as for RS, where core concepts used for looking up keys are far from easily feasible in P4 as described in the previous paragraph.

## 2.3 P4 and network programmability

P4 is a programming language that allows standardized programmability of network devices, especially targeting their packet forwarding planes. The language first appeared in 2013 and is since maintained by the P4 language consortium. Personally I was introduced to this language through my supervisor and further learned some of the basic concepts through the publicly available records of the P4 Developer Day held by Robert Soule, 2017. Annother important source for a deeper understanding of what the P4 language offers and which limitations exist is via the official specification by The P4 Language Consortium, 2021.
The idea behind learning P4 is to implement the best fitting of the previously presented learned index structures with the help of this language on a network device to satisfy our goal of resolving lookup requests directly through the network. The following preliminary facts need to be taken into consideration.

- The P4 packet flow consists of different pipelines, inlcuding a packet parser, a checksum verification, an ingress pipeline, an egress pipeline, a checksum computation and finally a packet deparser.
- Parsers can have different states and allow transitions from state to state. States can loop back to other states but their logic must be reducable to a final state machine at compilation time, meaning there is no dynamic form of recursion or iteration allowed. (The only exception to this are header stacks where a packet can contain a dynamic but only up to a fixed amount of stack items that can be extracted through some form of state recursion)
- There are match-action tables that can optionally be filled with data from the control plane during runtime via P4Runtime. Tables allow matches on keys and execution of some specific action depending on the match.
- There is no possibility of linking P4 source files to each other similar as for example in C. This creates an environement where no major libraries exist, at most code snippets could be used.
- There is no notion of floating point numbers or floating point arithmetic. The concepts do simply not exist in P4 land.

All together this leads to a very interesting language by itself but also to a rather cumbersome discovery of all that is actually not possible. Namely this means, especially in regard to what the different learned index structures implementations need, that in P4 there is no floating point arithmetic and there is no sort of dynamic loop or recursion capability.

# Chapter 3

# RMI on BMv2

The next chapter of this work is dedicated to the idea of implementing RMI using the reference P4 software switch BMv2. As briefly described in section 2.2.1, a RMI lookup heavily relies on floating point arithmetic but besides that does not need a lot of other operations to work properly. Therefore a major part of this chapter will be about dealing with these floating point operations. For a simpler start as well as for learning a lot of the basics of the P4 language the BMv2 software switch is used. This does bring quite a lot of advantages to begin with, but as decribed later in section 3.4 does also lead to a large differences between what is doable in theory in software and what is actually possible in a real world scenario.

## 3.1 BMv2 and Mininet

BMv2 stands for behavioral model version two and is the official reference P4 software switch implementation found at The P4 Language Consortium, 2020b. It is written in C++ and can take in and interpret a compiled P4 program and simulate the packet-processing behaviour specified in said P4 program. The implementation runs out of the box on traditional linux distributions and can be combined with virtual network simulation softwares such as Mininet Project Contributors, 2021. Altogether though with regard to this work, neither BMv2 nor Mininet are meant to be production-grade implementations in their area. This means that there are on one side in the case of BMv2 a lot less restrictions imposed than a real world switch would and on the other side for both tools there is a lot less processing speed and throughput potential available than real world equipment would provide.

With that out of the way the network architecture simulated by Mininet is extremely simple and barely even worth mentioning. Figure 3.1, in the next section, shows our simple setup and where the BMv2 software switch comes into play together with more detail about the packet flow.

## 3.2 Network setup and packet structure

When looking at how the SOSD benchmark implementation handles many different algorithms at the same time it becomes apparent that separating a phase where an algorithm can do its processing to return a result bound and on the other hand performing the so called last mile search on this remaining bound is key for most learned competitors. This separation comes in handy since performing an efficient search on a given range of data on a P4 switch is not easily doable as of today and therefore this task remains on the host by choice. This leads to the setup shown in figure 3.1. In principle a host sends a regular ethernet packet containing all the

usual fields such as destination and source MAC address to the switch. Importantly though the packet sets the ether type field to a custom value of $0x8008$ and appends a payload containing the desired lookup key and space for the response in form of a field for the guessed position and a field for the expected error calculated during the learning phase as visualized in figure 3.2. As a next step the switch performs the actual RMI calculations, completes the guessed position and expected error field in the packet and forwards the packet back to the source MAC address. The last step that was previously abstracted away is now performed upon receiving the forwarded response packet on the host, meaning that some sort of last mile search is performed in the interval between $[estimated - error, estimated + error]$ to finally find at which position the initially desired lookup key is to be found.

In terms of flexibility there is absolutely no restriction of where to forward the finalized RMI packet containing the estimated position and caluclated error to. There are all sorts of possibilites here to use the full power of P4 and network programmability to achieve combinations of packet forwarding and lookup operations. Annother flexibility is the sort of search that is used on the host on the narrowed down search bound. For this work as well as in the SOSD benchmark a regular form of binary search is used, but for different concrete use cases other algorithms may be preferable.

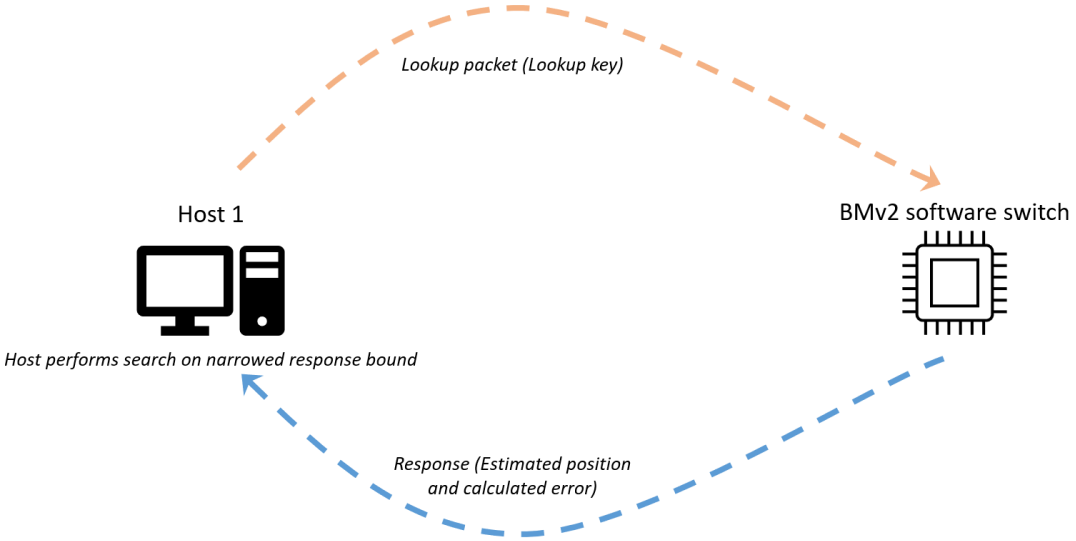


FIGURE 3.1: **Mininet network architecture.** Simple network architecture and packet flow proposition to run RMI on a P4 capable switch.

## 3.3 Implementation

The reference RMI implementation does generate C++ code depending on the specific dataset it is trained on. This part of the algorithm though is not yet part of this chapter and will be explored more in depth in chapter 4. As a first step to try to translate existing RMI code in C++ into P4 our approach was to decide for a single dataset as well as some constant input parameters. The result of this is that whenever running the existing RMI implementation the exact same C++ code is generated. An example can be found in the appendix in section 7.1. The next step would be to examine the generated code and finally implement the exact same behaviour

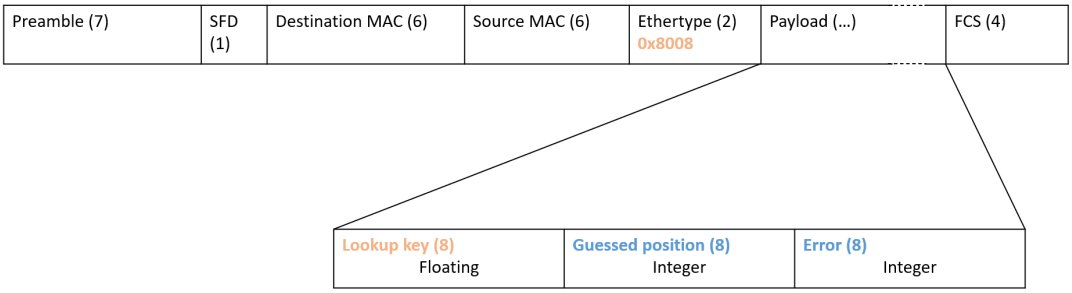


FIGURE 3.2: **Learned RMI packet structure.** Visualization of an RMI ethernet packet containing the custom ethertype and payload. Field sizes are shown in octets.

in P4 by hand. For this multiple challenges must be tackled. One of them being the load function where quite a large chunk of binary layer one parameter data is loaded into memory. Annother one becoming apparent when observing that in this case the learning phase decided for layer zero to use a cubic function and for layer one to use a linear function. The implementations of both of these functions are short as shown in figure 3.3, but since they solely rely on using the FMA instruction it will turn out that they are going to be quite tricky to translate to P4. In other words a second challenge will be to implement an operation that behaves similarly to the commonly in hardware implemented fused multiply-add CPU instruction in P4.

```
inline double linear(double alpha, double beta, double inp) {
  return std::fma(beta, inp, alpha);
}

inline double cubic(double a, double b, double c, double d, double x) {
  auto v1 = std::fma(a, x, b);
  auto v2 = std::fma(v1, x, c);
  auto v3 = std::fma(v2, x, d);
  return v3;
}
```

FIGURE 3.3: C++ lookup implementation for the linear and cubic model.

### 3.3.1 FMA in P4

This section focusses on a software implementation of the fused multiply-add instruction in P4. The goal is to potentially provide a proof of concept and put performance or optimization considerations aside for now. In section 3.4 a more top down look on things will be given together with some thoughts in form of an evaluation in what way this was a good idea or not.

The FMA instruction takes in three parameters and calculates the value resulting from $(x * y) + z$ rounded only once. This means that in order to implement an FMA instruction in P4 one needs to be able to multiply two floating point values as well as adding two floating point values together. This consequently means that there has to be some sort of representation in P4 for a floating point value. This is pretty easily doable by defining a custom header type that follows the official IEEE754 floating

point standard published by the IEEE, 2019 shown in figure 3.4. For 64-bit double values the standard describes a 1-bit field representing the sign, an 11-bit field representing the exponent stored as non-negative biased binary number and finally a 52-bit field representing the mantissa stored as a regular binary number excluding the so called "hidden bit". With this definition set floating point addition as well as floating point multiplication can be addressed.

Floating point addition works by first setting the hidden bit on both mantissa fields. Next at its core, by looking at the exponent difference between the two floating point values and shifting the mantissa of the smaller value to the right by that amount. With both mantissas now in the same exponent base, they can be added together with a regular bit addition operation. Lastly both the resulting sign as well as the exponent are determined by the larger floating point value. With this the mathematical addition result is calulcated but the representation is not yet sound, meaning that that due to the calculation the first significant mantissa bit might not be at position 53 where the so called hidden bit is supposed to be. To correct this a normalzation procedure is run where the mantissa and exponent are shifted and adapted such that the first significant mantissa bit moves to position 53 and will finally be omitted to increase the representable value range. In P4 this operation is implemented by ternary matching the calulcated mantissa against a static normalzation table which is shown in a narrowed down form in the appendix in section 7.2.3. The implementation of the addition operation can be found in the appendix in section 7.2.1.

On the other hand floating point multiplication works similarly by first setting the hidden bit on both mantissa fields. Next, to determine the resulting sign, both input sign bits are XOR-ed together. Further, to determine the resulting exponent, the two unbiased input exponents are added together with a regular bit addition operation. Finally the two mantissa fields are mutliplied together this time with a regular bit multiplication operation and shifted back into their initial exponent space. At this point the mathematical multiplication result is caluclated but again the representation is not yet sound. To correct this the exact same normalzation procedure described in the previous paragraph is run. The implementation of the multiplication operation can be found in the appendix in section 7.2.2.

To finish this section now with both mathematical base operations in place, the final FMA control in P4 does simply execute both of the just described operations one after the other.

```
typedef bit<1> sign_t;
typedef bit<11> exponent_t;
typedef bit<52> mantissa_t;

struct double_t {
  sign_t sign;
  exponent_t exponent;
  mantissa_t mantissa;
}
```

FIGURE 3.4: **IEEE754.** P4 header definition following the IEEE754-2019 standard for 64-bit double values.

### 3.3.2 Loading model parameters in P4

With the calculation heavier operations out of the way a next challenge is to have access to the data from the binary file normally directly loaded into memory in C++ representing the so called model parameters that are accessed depending on the resulting calculations of the previous layer. To solve this problem a combination of control plane and data plane is needed. On one side the data plane does predefine an empty table description that later can be filled, while the switch is runnning, from the control plane via the P4Runtime API specified and maintained by The P4 Language Consortium, 2020a. An example of such a table definition is given in the appendix in section 7.3.1. The data plane on the other side now simply reads the existing model parameters file and sends these informations over to the switch. This is done with Python since the P4Runtime environement is implemented in Python. To work correctly the script takes in the location of the binary model parameters file as well as the necessary connection parameters to establish a connection to the switch. The table entries are sent in batches to the switch for acceptable performance in the simlauted network and hardware environement but besides that the implementation is trivial. On the data plane side of things again, whenever a lookup packet arrives and a future layer needs access to model parameters the P4 program performs an exact table match with the reulting index from the previous layer to determine the parameters for the next layer.

### 3.3.3 The actual lookup function

Finally with the described functions implemented the actual lookup control simply becomes a matter of putting it all together as shown in the appendix in section 7.4. One additional but relatively simple function that had to be implemented was a function that casts a floating point value to an integer in P4. As a basic concept each layer takes in its layer parameters as arguments and returns a prediction index for the next layer. In the example case of the books_200M dataset with 32-bit keys a two-layer RMI is generated where the first layer follows a cubic model and takes in the statically in the C++ header file (shown in 7.1) present L0 parameters. After doing the FMA calculations for said cubic layer, the result is cast back to an integer which is then used as an index to perform an exact table match with the table described in the previous section to retrieve the L1 parameters used as input for the linear model. Finally the FMA calulations for the linear model are computed and after casting the floating point result back to an integer value, the guessed index is clamped to a value between zero and the dataset size and finally returned.

## 3.4 Evaluation

When running and testing the described setup including the virtual network and the software emulated network switch, the possibilites are obviously very limited. Still though, when looking at accuracy when sending one million test lookups generated by the SOSD benchmark, the P4 implementation on the switch achieves a 100% prediction accuracy. Meaning for all lookup packets sent, the precomputed lookup key is actually to be found in the range given by the guess and the error returned in the response packet.

In any case accuracy is fine and for the scope of this work rather pleasing but the implementation as is does ignore quite a lot of real world limitations. A first one of

them being for example that currently available switches do mostly not support multiplication on their ALUs. This does break the floating point multiplication function, which is needed for the FMA implementation which in turn is needed for different model lookup implemenations. Annother one of these limitations being that current real world switches are limited to a certain amount of ALU stages. Meaning that for the switch to reach optimal operation speed, a packet can maximally perform a certain amount of computation. For now since not having any ALU hardware support for floating point arithmetic all mathematical operations are implemented in software and therefore in the current implementation this limit of ALU stages is more than exceeded. A next limitation is discussed in the following section where current switches are very limited in terms of what bit width ALUs can handle for basic operations. A final limitation or moreso a very large negative point comes from the fact that the FMA operation is fully implemented in software. This not only leads to the previously described overfull stage usage but also to bad performace in comparision to hardware implemented FMA instructions on a server's CPU where not only the hardware itself means a significant speed up but also the fact that FMA circuits can often benefit from smarter design choices instead of just performing one mathematical operation after the other.

All in all with so much of these limitations on the table, the goal and purpose of this work and of this implemenation definetely and at best becomes a theoretical proof of concept, showing what could potentially be possible. While reaching expected prediction accuracy a lot of progress in terms of extending ALU capability on real world switches needs to be done in order to enable RMI the way it was proposed in this chapter.

### 3.4.1 32-bit width attempt

As already mentioned currently existing real world switches often have ALUs that can maximally treat and compute values up to 32-bit width. With this in mind the first attempt made for all the steps described in this chapter was also limited to floating point values following the IEEE754 single floating point standard. While initially working quite well and being a bit simpler to deal with in terms of readability, when testing lookup accuracy it pretty quickly became clear that the amount of accuracy that single floating point values offer was simply not precise enough for RMI to work properly. This especially holds true due to the fact that all the generated model parameters are designed to use double floating point values. Changing the inner workings of the RMI learning phase to use larger error bounds or cope with the smaller accuracy in annother way would have very quickly overshot the scope of this work, especially when looking at how much complexity and work was already put into this part of the algorithm by other people.

# Chapter 4

# RMI for P4

## 4.1 RMI reference implementation

The RMI reference implmentation following from Marcus, Zhang, and Kraska, 2020 available at Ryan Marcus, 2020 is implemented in Rust and primarily serves as a compiler that takes in a dataset as input and outputs C++ source code files. In this constellation one can play with multiple tuning parameters in order to influence the generated code and with that the potential performance of the generated implementation. Concretely there is the possibility to choose which model type is used on which level as well as a parameter called the branching factor that determines the number of leaf models between two layers. The reference implementation currently supports nine different types of model types the most frequently used ones being linear and cubic. The functionality of the implemenntation though does not stop at this point, instead there is a possibility to pass an optimize optio to the executable to let RMI perform automatic tuning that outputs a table that covers heuristically selected possible RMI configurations that cover the Pareto front. This table then contains different suggestion for which combination of models can be used together with a suggested branching factor as well as information about the layer parameters size and approximately how many binary search steps will be needed in the last mile search. This table can fruther be used as input to the reference implementation to directly generate code for each table entry.

## 4.2 Adaptation for P4

Until now the the in chapter 3 discussed RMI implementation for BMv2 was extremely unflexible and solely focussed on a single dataset where every configuration or adaptation step was done by hand in a quite uncomfortable way. This chapter is about going a step further, where an adaptation of the reference RMI implementation in Rust should potentially be able to automatically generate P4 source code files depending on which input dataset was used and what models were selected. With this not only an RMI implementation in P4 for the books_200M dataset with 32-bit keys should be doable but instead lots of different configurations for all provided SOSD datasets hopefully become runnable on the BMv2 switch.

An important thing to mention though is that the entire mathematical or learned part of the reference RMI implementation will stay completely untouched and mainly the code generation part of the project will be adjusted and extended.

## 4.3 Code generation

Generally when looking at the fully implemented final result from chapter 3 a lot of static code is to be found in the P4 file that can be used for any sort of dataset or model combination. Coyping all these header definitions or the normalization function and other helper functions into a P4 source file is straight forward. A first thing that remains is to treat code generation for different models lookup functions depending on which models are used. An important property here is that a function should really only be printed into the result file if it is actually needed. This is covered in more detail in section 4.3.1 and 4.3.2. Until here still not much complexity is involved. A next thing to treat is code generation of the actual lookup function which has to adapt with respect to different model combinations. This is more or less the centerpiece and most complex part of the code generation and discussed more in detail in 4.3.3. Finally the code generation that makes sure that model parameters can be sent over to the switch via P4Runtime in Python or when small enough statically printed into the result source file is left. This part of the implementation proved to be more complex than initially thought since the generated Pyhton source file for P4Runtime has to seamlessly work with the P4 table definition printed into the P4 source file. This is looked at in more detail in 4.3.4.

### 4.3.1 For linear and cubic models

The inner workings of the lookup functions for both of these models were already quite extensively covered in section 3.3 and 3.3.1 together with the concrete implementation shown in the appendix in section 7.2. The only thing left for the code generation in the rust implementation apart from printing said code into the P4 source file is to make sure that either static model parameters are correctly printed into the source file or that larger amounts of model parameters are correctly loaded during the switches runtime. As already stated this is looked at in a more general approach in section 4.3.4.

### 4.3.2 For radix models

The lookup function for radix models fortunately is the only lookup function that does not rely on floating point arithmetic and is therefore predestined to work in P4. The reference RMI implementation contains two radix models. The first of them uses a certain prefix length to bit shift on the input to generate a radix value which is directly the resulting output of the model whereas the second model does calculate a radix value based on the input the same way but instead uses it then to index a radix table which then serves as the resulting output of the model. For these models the adaptation into P4 and code generation is even simpler since all necessary primitive operations used are also available in P4. The second radix model involving a radix table though needs a bit more consideration which involves loading the radix table using the mechanisms described in 4.3.4 and adapting the lookup function generation accordingly.

### 4.3.3 For the lookup function

The lookup function is probably the most important but with that also the trickiest part of the code generation implementation. In this part all sorts of combinations of models as well as other properties of the learned RMI must be considered. Generally

due to the nature of RMI in the sense that each model layer's output provides the input index for the next layer, the generation code works in the same way by looping over each generated layer. For each layer based on model properties it is decided if floating point or integer input and output is needed. Based on this information, conversions between layers are added if necessary. Further there is a difference between code generation for the first layer and all following layers. Theoretically several following model layers are possible but the reference implementation and especially in the scope of this work, as previously stated, focus lies on only one single following layer from now on designated as second layer. With that said when generating code for the first layer, the model parameters originally printed into a header file are now converted to the customly defined floating point header format in P4 and statically written into the resulting source code file as input for the model function. One exception to this being the model involving a radix table where additionally to the call of the model function a table lookup happens into the model parameters table loaded with the mechanism described in the next section to make the sure the correct output index is retrieved as input for the next layer. Even though this is already implied, in both cases the code generation does insert a call to the respective model function at the appropriate location. Finally when generating code for a following second layer model, first a call to the function that looks into the model parameters table to retrieve the corresponding model parameters gets written to the source code file with the index from the previous layer as input index. Finally a call to the second layer model function is appended with the just retrieved model parameters as input arguments. At the end a function to calculate the final result by clamping it to a value between 0 and the dataset size is appended and with that the generation of the lookup function is complete.

### 4.3.4 For loading model parameters

## 4.4 Supporting other models

# Chapter 5

# Measurements

## 5.1 Method

describe the idealistic and theoretical measurement approach (measuring search time and hypothetically calculating the rest)

## 5.2 Results

## 5.3 Evaluation

# Chapter 6

# Conclusion

## 6.1 Conclusion

## 6.2 Future Work

# Bibliography


Ferragina, Paolo and Giorgio Vinciguerra (2020). "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds". In: *PVLDB* 13.8, pp. 1162–1175. ISSN: 2150-8097. DOI: 10.14778/3389133.3389135. URL: https://pgm.di.unipi.it.

IEEE (2019). *IEEE Standard for Floating-Point Arithmetic*. https://standards.ieee.org/ieee/754/6210/.

Kipf, Andreas et al. (2019). "SOSD: A Benchmark for Learned Indexes". In: *NeurIPS Workshop on Machine Learning for Systems*.

— (2020). "RadixSpline: a single-pass learned index". In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, 5:1–5:5. DOI: 10.1145/3401071.3401659. URL: https://doi.org/10.1145/3401071.3401659.

Kraska, Tim et al. (2018). "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 489–504. ISBN: 9781450347037. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

Marcus, Ryan, Emily Zhang, and Tim Kraska (2020). "CDFShop: Exploring and Optimizing Learned Index Structures". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2789–2792. ISBN: 9781450367356. DOI: 10.1145/3318464.3384706. URL: https://doi.org/10.1145/3318464.3384706.

Mininet Project Contributors (2021). *Mininet*. http://mininet.org/.

Robert Soule Stephen Ibanez, Carmelo Cascone Brian O'Connor Mina Tahmasbi Samar Abdi (2017). *P4 Developer Day*. https://www.youtube.com/watch?v=3DJeqS_dl_o&list=PLf7HGRMAlJBzGC58GcYpimyIs7D0nuSoo.

Ryan Marcus (2020). *RMI*. https://github.com/learnedsystems/RMI.

The P4 Language Consortium (Dec. 2020a). *P4Runtime Specification*. https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html.

— (2020b). *The reference P4 software switch*. https://github.com/p4lang/behavioral-model.

— (May 2021). *P4 Specification*. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

# Chapter 7

# Appendix

## 7.1 Generated C++ code for books_200M with 32-bit keys

### 7.1.1 Header file (L0 parameters)

```
namespace books_200M_uint32_0 {
const double L0_PARAMETER0 = 0.0;
const double L0_PARAMETER1 = 0.0;
const double L0_PARAMETER2 = 0.003906249768078851;
const double L0_PARAMETER3 = 0.0;
char* L1_PARAMETERS;
}
```

### 7.1.2 Code file (Lookup code)

```
#include "books_200M_uint32_0_data.h"
#include <math.h>
#include <cmath>
#include <fstream>
#include <filesystem>
#include <iostream>
namespace books_200M_uint32_0 {
bool load(char const* dataPath) {
  {
    std::ifstream infile(std::filesystem::path(dataPath) / "
    books_200M_uint32_0_L1_PARAMETERS", std::ios::in | std::ios::binary
    );
    if (!infile.good()) return false;
    L1_PARAMETERS = (char*) malloc(402653184);
    if (L1_PARAMETERS == NULL) return false;
    infile.read((char*)L1_PARAMETERS, 402653184);
    if (!infile.good()) return false;
  }
  return true;
}
void cleanup() {
  free(L1_PARAMETERS);
}
```

```
inline double linear(double alpha, double beta, double inp) {
  return std::fma(beta, inp, alpha);
}

inline double cubic(double a, double b, double c, double d, double x) {
  auto v1 = std::fma(a, x, b);
  auto v2 = std::fma(v1, x, c);
  auto v3 = std::fma(v2, x, d);
  return v3;
}

inline size_t FCLAMP(double inp, double bound) {
  if (inp < 0.0) return 0;
  return (inp > bound ? bound : (size_t)inp);
}

uint64_t lookup(uint64_t key, size_t* err) {
  double fpred;
  size_t modelIndex;
  fpred = cubic(L0_PARAMETER0, L0_PARAMETER1, L0_PARAMETER2,
    L0_PARAMETER3, (double)key);
  modelIndex = (uint64_t) fpred;
  fpred = linear(*((double*) (L1_PARAMETERS + (modelIndex * 24) + 0)),
    *((double*) (L1_PARAMETERS + (modelIndex * 24) + 8)), (double)key);
  *err = *((uint64_t*) (L1_PARAMETERS + (modelIndex * 24) + 16));

  return FCLAMP(fpred, 200000000.0 - 1.0);
}
}
```

## 7.2 FMA operation in P4

### 7.2.1 Addition

```
action floating_add(in double_t first, in double_t second, out
    overflow128_t result) {
  bool first_bigger = first.exponent == second.exponent ? first.
    mantissa > second.mantissa : first.exponent > second.exponent;
  uint64_t first_mantissa = ((uint64_t) first.mantissa) | HIDDEN_BIT;
  uint64_t second_mantissa = ((uint64_t) second.mantissa) | HIDDEN_BIT;

  if ((first.exponent == 0 && first.mantissa == 0) || (second.exponent
    == 0 && second.mantissa == 0)) {
    if (first.exponent == 0 && first.mantissa == 0) {
      result = { second.sign, second.exponent, (bit<128>)
    second_mantissa }; // first zero, return second
    } else {
```

```
      result = { first.sign, first.exponent, (bit<128>) first_mantissa
    }; // second zero, return first
    }
    return;
  }

  exponent_t exponent_difference = first_bigger ? (first.exponent -
    second.exponent) : (second.exponent - first.exponent);
  uint64_t bigger_mantissa = first_bigger ? first_mantissa :
    second_mantissa;
  uint64_t smaller_mantissa = first_bigger ? second_mantissa :
    first_mantissa;
  smaller_mantissa = smaller_mantissa >> ((bit<8>) exponent_difference)
    ;

  result.sign = first_bigger ? first.sign : second.sign;
  result.exponent = first_bigger ? first.exponent : second.exponent;
  if (first.sign != second.sign) { // inputs have different sign, this
    is a subtraction
    result.mantissa = (bit<128>) (bigger_mantissa - smaller_mantissa);
  } else { // both numbers have the same sign, regular addition
    result.mantissa = (bit<128>) (bigger_mantissa + smaller_mantissa);
  }
}

control FloatingAdder(in double_t first, in double_t second, out
    double_t result) {
  FloatingNormalizer() normalizer;

  overflow128_t temp;
  apply {
    floating_add(first, second, temp);
    normalizer.apply(temp);
    result = { temp.sign, temp.exponent, (mantissa_t) temp.mantissa };
  }
}
```

### 7.2.2 Multiplication

```
action floating_multiply(in double_t first, in double_t second, out
    overflow128_t result) {
  if ((first.exponent == 0 && first.mantissa == 0) || (second.exponent
    == 0 && second.mantissa == 0)) {
    result = { first.sign ^ second.sign, 0, 0 }; return;
  }

  result.sign = first.sign ^ second.sign; // ^ = xor
  result.exponent = (first.exponent - EXPONENT_BIAS) + (second.exponent
     - EXPONENT_BIAS) + EXPONENT_BIAS;
```

```
  bit<128> first_mantissa = ((bit<128>) first.mantissa) | (bit<128>)
    HIDDEN_BIT;
  bit<128> second_mantissa = ((bit<128>) second.mantissa) | (bit<128>)
    HIDDEN_BIT;

  result.mantissa = (first_mantissa * second_mantissa) >> 52;
}

control FloatingMultiplier(in double_t first, in double_t second, out
    double_t result) {
  FloatingNormalizer() normalizer;

  overflow128_t temp;
  apply {
    floating_multiply(first, second, temp);
    normalizer.apply(temp);
    result = { temp.sign, temp.exponent, (mantissa_t) temp.mantissa };
  }
}
```

### 7.2.3 P4 floating point normalzation

```
control FloatingNormalizer(inout overflow128_t overflow) {
  action floating_shift_left(inout overflow128_t result, bit<8> amount)
     {
    result.mantissa = result.mantissa << amount;
    result.exponent = result.exponent - (exponent_t) amount;
  }

  action floating_shift_right(inout overflow128_t result, bit<8> amount
    ) {
    result.mantissa = result.mantissa >> amount;
    result.exponent = result.exponent + (exponent_t) amount;
  }

  table floating_normalize {
    key = {
      overflow.mantissa: ternary;
    }
    actions = {
      floating_shift_left(overflow);
      floating_shift_right(overflow);
      NoAction;
    }
    const default_action = NoAction();
    const entries =  { // value to match against && bit mask
      0b1000...0 &&& 0b100...0: floating_shift_right(overflow, 75);
      0b0100...0 &&& 0b110...0: floating_shift_right(overflow, 74);
```

```
      0b0010...0 &&& 0b111...0: floating_shift_right(overflow, 73);
      // ...
      // the mask where the first significant bit is at pos 53 needs no
    action
      // ...
      0b0...0100 &&& 0b1...100: floating_shift_left(overflow, 50);
      0b0...0010 &&& 0b1...110: floating_shift_left(overflow, 51);
      0b0...0001 &&& 0b1...111: floating_shift_left(overflow, 52);
    }
  }

  apply {
      floating_normalize.apply();
  }
}
```

## 7.3 Loading model parameters in P4

### 7.3.1 Dataplane RMI table definition

```
control ModelLookup(in uint64_t model_index, out double_t first_l1, out
    double_t second_l1, out uint64_t err) {

  action assign_variables(sign_t first_sign, exponent_t first_exponent,
    mantissa_t first_mantissa,
                          sign_t second_sign, exponent_t
   second_exponent, mantissa_t second_mantissa,
                          uint64_t err_val) {
      first_l1 = { first_sign, first_exponent, first_mantissa };
      second_l1 = { second_sign, second_exponent, second_mantissa };
      err = err_val;
  }

  table model_lookup {
      key = {
          model_index: exact;
      }
      actions = {
          assign_variables;
          NoAction;
      }
      const default_action = assign_variables(0, 0, 0, 0, 0, 0, 0);
      const size = 550000;
  }
  apply {
      assign_variables(0, 0, 0, 0, 0, 0, 0);
      model_lookup.apply();
  }
```

```
}
```

## 7.4 RMI lookup code in P4

```
control LearnedLookup(in double_t input_key, out uint64_t guess, out
    uint64_t guess_err) {
  ModelLookup() lookup_instance;
  LearnedCubic() cubic_instance;
  LearnedLinear() linear_instance;

  double_t fpred;
  uint64_t model_index;
  double_t first_l1; double_t second_l1;

  apply {
    cubic_instance.apply({ 0, 0, 0 }, { 0, 0, 0 }, { 0, 1014,
    4503599092596736 }, { 0, 0, 0 }, input_key, fpred);
    double_to_int(fpred, model_index);

    // retrieving L1 parameters
    lookup_instance.apply(model_index, first_l1, second_l1, guess_err);

    linear_instance.apply(first_l1, second_l1, input_key, fpred);
    double_to_int(fpred, guess);

    f_clamp(fpred, 0xbebc1ff, guess); // 0xbebc1ff = 199999999
  }
}
```