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

Recently several works proposed new approaches on how advancements in machine learning can be used to improve indexing strategies on sorted data. This led to the SOSD benchmark [10] which enabled having a baseline for evaluating different competitors in a standardized way. At the same time, with a wider adoption of network programmability through P4, a trend towards outsourcing computationally intensive procedures to programmable switches started. Our goal is to combine these two advancements by maintaining recent progress that learned indexing algorithms offer and evaluate possibilities of further leveraging their performance by using the power of network programmability.

After careful evaluation we come to the decision that we continue our work focusing on the learned indexing algorithm RMI [7], which suits our idea of implementing parts of it over the network best. Indeed we find that the P4 specification [14] allows an implementation of the lookup part of RMI on the switch and attains perfect accuracy when tested on virtually simulated network hardware. At this point we analyze how far away our theoretical implementation is, from actually running on real world hardware and we find that there is a long way to go. From this result we then generalize our solution to any dataset and arbitrary RMI configuration by adapting the code generation part of the RMI reference implementation [9] to output P4 source code files. We finally conclude our work by coming up with a strategy to, even though having a theoretical implementation, estimate how much our idea of outsourcing RMI calculations to the network could benefit a closed system. We find that we could save around 50-100ns per lookup per last mile search worker, which would result in a constant speed up that scales horizontally with the amount of last mile search workers available to a single switch.

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 by setting up the SOSD benchmark [6] on a local system, with the premise in mind that learned index structures potentially can outperform traditional index structures. Part of this process would be to verify the promised results on said local system. After comparable results to the original paper the idea of taking advantage of the benefits that learned index structures offer and fusing these with the now more and more established network programmability offered by P4 became the 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 it's 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. Higher prediction accuracy in this context means that the predicted bound returned from the learned index structure encloses the actual key more closely. 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, NICs, SSD drives, etc.) have 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 is given, as well as which ones and why some of them were 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 RMI reference implementation [9] to be able to generate P4 source code files is presented.

In chapter 5, some additional experiments are made and combined with what switches are capable of doing today to try to get an approximate idea 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 [6] on a local system. Further, as a second indicator, by looking at what different learned index structures 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 our work consisted of comparing our local results with what is given by the authors in table two of [6], which gave the result shown in figure 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 we select the optimal run with respect to the three metrics lookup time, build time and index size. At this point in time it was important for this work to take note of the fact that when tuned properly, learned index structures effectively *can* outperform traditional index structures with respect to the lookup time metric. When considering the remaining 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 figure 2.2 to then gain with faster lookups.

2.2 Learned index structures

In the SOSD 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 [7] and implemented for the benchmark in [9]. RS (Radix Spline) is proposed and implemented by the original authors in [5]. Finally the same holds for PGM (Piecewise Geometric Model), which is proposed and implemented in [2].

	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 above.

	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 above.

2.2.1 RMI: Recursive Model Indexes

RMI [7] 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, RMIs are fixed to two layers since this proved to be most efficient for most datasets and also reduces complexity for further chapters. In the reference implementation [9]

C++ source code files are generated that contain parameters as well as the adapted code depending on which models were chosen on which layer. Generally the input key is given as input to a first layer, which generates an index that then serves as a starting point for the next layer, and so on, until finally a last layer retrieves the key's estimated 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.

2.2.2 RadixSpline: A Single-Pass Learned Index

RS [5] is built on top of the idea of fitting a linear spline to the CDF function 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 not feasible in P4.

2.2.3 PGM: The Piecewise Geometric Model index

PGM [2] 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 ϵ distance to their actual position. By applying this approximation recursively onto itself multiple levels of PLAs are built such that efficient search becomes possible.

The given lookup code in [2], 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 section.

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 [12]. Another important source for a deeper understanding of what the P4 language offers and which limitations exist is via the official specification [14].
The idea behind learning P4 is to implement the best fitting of the previously presented learned index structures 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 state recursion)
- There are match-action tables that can optionally be filled with data from the control plane during runtime via P4Runtime [15]. Tables allow matches on keys and execution of some specific action depending on the match.
- There is no possibility of linking P4 source code 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 everything that is actually not possible. Namely this includes, especially with regard to what different learned index structure implementations require, 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 on the reference P4 software switch BMv2 [16]. 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 available. Therefore a major part of this chapter will be about dealing with floating point arithmetic in P4. 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 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 [16] stands for behavioral model version two and is the official reference P4 software switch implementation. It is written in C++ and takes a compiled P4 program, interprets it and simulates 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 [11]. All together though 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 shows our simple setup and where the BMv2 software switch comes into play together with more detail about the packet flow. Importantly though, this network structure is in no way bound to the learned structure implementation in P4, meaning that any arbitrary complex other network structure would work equally well.

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 network 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 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. Also in terms of what packet protocols are used for communication between the host and the switch, no restrictions apply. There are all sorts of possibilites to use the full power of P4 and network programmability to achieve combinations of packet forwarding and lookup operations. Another 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.

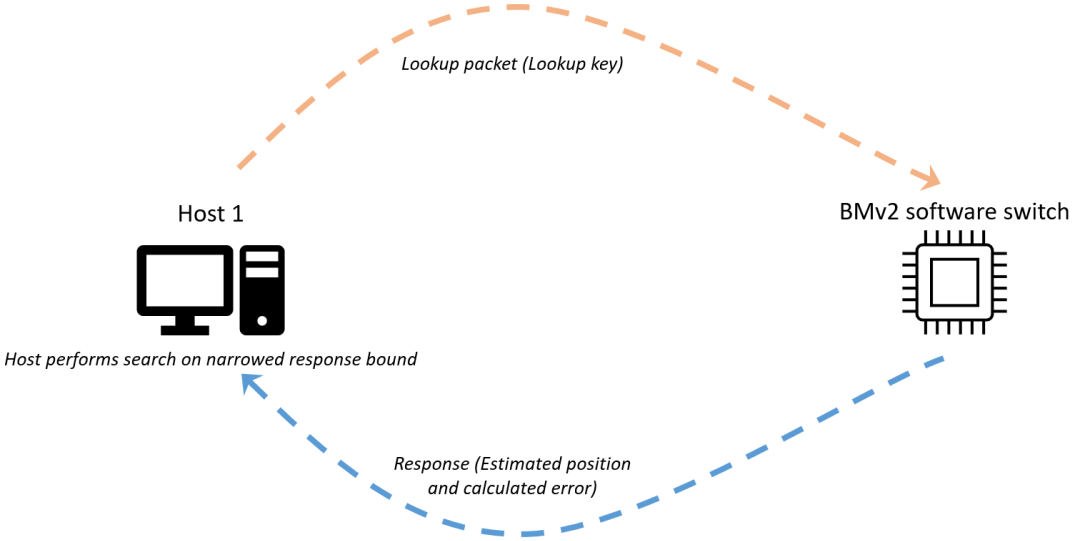


FIGURE 3.1: **Mininet network architecture.** Simplest network architecture and packet flow proposition to run RMI on a P4 capable switch.

3.3 Implementation

The reference RMI implementation [9] does generate C++ code depending on the specific dataset it is trained on. The code generating 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 code of the reference implementation into P4, our approach was to decide for a single dataset as well as fixed input parameters. The result of this is that whenever running the reference implementation the exact same

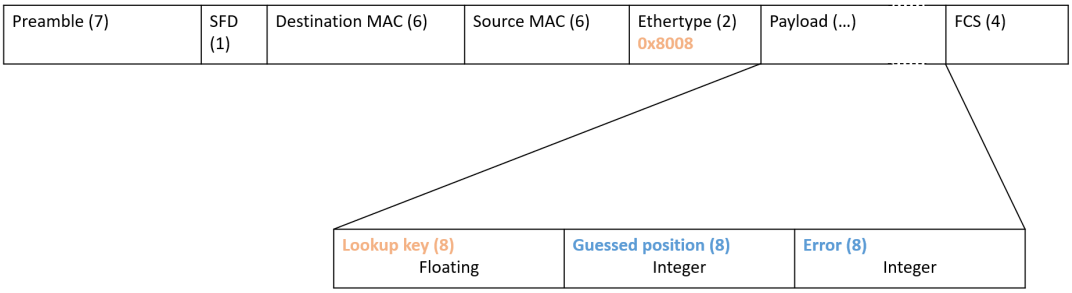


FIGURE 3.2: **Learned RMI packet structure.** Visualization of an RMI ethernet packet containing the custom ethertype and payload. Field sizes are shown in octets.

C++ code is generated. An example can be found in the appendix in section 7.1. The next step was to examine the generated code and implement the same behaviour in P4 by hand. For this multiple challenges had to be tackled. The first one of them being the load function, where quite a large chunk of binary layer one parameter data is loaded into memory. Another 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 shown in figure 3.4 that follows the official IEEE754 floating point standard [3] published by the IEEE. 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.

```
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.

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 input. With this the mathematical addition result is calulcated but the representation is not yet sound, meaning 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 normalization 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 calculated mantissa against a static normalization 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.

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 once again the representation is not yet sound. To correct this, the exact same normalization 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.

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. The trivial implementation can be found in the appendix in section 7.2.4.

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, representing the so called model parameters. These are accessed depending on the resulting calculations of the previous layer to determine the inputs for the next layer. To solve this problem a combination of control plane and data plane is needed. On one side the data plane does define an empty table declaration which can be filled later, while the switch is runnning, from the control plane via the P4Runtime API [15]. An example of such a table definition is given in the appendix in section 7.3.1. The control plane on the other side now simply reads the existing model parameters file and sends these informations over to the switch. This code is written in Python since the P4Runtime environment is implemented in Python. To work correctly the script takes 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 simulated network environment but besides that the implementation shown in the appendix in section 7.3.2 is trivial. On the data plane side of things again, whenever a lookup packet arrives and a layer needs access to model parameters, the P4 program performs an exact table match with the resulting 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 figure 7.4. One additional but relatively simple action that had to be implemented was a function that casts a floating point value to an integer in P4. In order to briefly illustrate the basic concept, which consists of each layer taking its layer parameters as arguments and returning a prediction index for the next layer, we look at the example case of the books_200M dataset with 32-bit keys a little more in depth. As previously stated, for this dataset and the fixed parameters the reference implementation generates a two-layer RMI, where the first layer follows a cubic and the second layer a linear model. The first layer (cubic model) takes the statically present L0 parameters as input. 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 second layer. Finally the FMA calulations for the second layer (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 on 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 reaches a 100%

prediction accuracy. Meaning for all lookup packets sent, the desired 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 implementations. A next limitation is discussed in the next section where current switches are very limited in terms of what bit width ALUs can handle for basic bit operations. Finally the most important limitation or more so a large performance issue comes from the fact that current real world switches are limited to a certain amount of ALU stages, meaning that for the switch to run at optimal operation speed, a packet can maximally perform a certain amount of chained operations, before drastically slowing down operation speed of the switch. For now, since not having any ALU hardware support for floating point arithmetic, all mathematical operations are implemented in software and therefore the limit of ALU stages for optimal operation speed is more than exceeded. This not only leads to overfull stage usage but also to bad performance in comparision to hardware implemented FMA instructions on a server's CPU, where not only the fact that the operations are implemented in hardware itself means a significant speed up, but also the fact that FMA circuits 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 another 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 implementation following from [9] available at [13] 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 hyperparameters 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 model types the most frequently used ones being linear and cubic. The functionality of the implementation though does not stop at this point, instead there is a possibility to pass an optimize option 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 suggestions for which combination of models can be used together with a branching factor as well as information about the layer parameter size and approximately how many binary search steps will be needed in the last mile search. This table can further be used as input to the reference implementation to directly generate code for each table entry.

4.2 Adaptation for P4

Until now the in chapter 3 discussed RMI implementation for BMv2 was extremely unflexible and solely focussed on a single dataset where every configuration or change 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 targeted 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 possible, but instead lots of different configurations for all provided SOSD datasets hopefully become executable 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 only the code generation part of the reference implementation will be adjusted.

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 which stays the same for any 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 mathematical helper functions depending on which models are used. An important property here is that a function should only be printed into the result file if it is actually needed. This is covered in more detail in sections 4.3.1 and 4.3.2. 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 the centerpiece and most complex part of the code generation and discussed more in detail in section 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 remains. 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 table declaration printed into the P4 source file. This is looked at in more detail in section 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 7.2.4. The only thing left for the code generation in the proposed implementation apart from printing said code into the P4 source code 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 from a more general point of view in section 4.3.4.

4.3.2 For radix models

The model function for radix models is the only one 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 this work, as previously stated, focus on only having a single following layer from now on designated as the second layer. Therefore, 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 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 into the model parameters table loaded with the mechanism described in the next section happens. 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. Next, when generating code for a following second layer, 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 resulting index from the previous layer as input. 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. With that the generation of the lookup function is complete.

4.3.4 For loading model parameters

The RMI reference implementation does already dump larger chunks of model parameters into a binary file. The idea of saving a binary file containing all layer parameters for later use is kept by our implementation. In order to load these parameters properly for each layer, multiple source code files are concerned. The first one being the table declaration itself in the P4 source code file. A table declaration and a corresponding table lookup function are added to the P4 source file whenever a layer needs to load model parameters. The generation of this function takes into account how many parameters need to be loaded and how large the table is going to be based on the learned RMI layer properties. Finally the generated table lookup function can be used at the appropriate position in the lookup function in order to retrieve layer parameters for a specific model index. The second file has to be generated in Python and uses P4Runtime [15]. It has to load the previously stored layer parameters from said binary file and then fill the previously declared table. It does so by creating a table entry for each model parameter loaded from the binary file and sending these in batches to the switch. The generation of this source code file is in the same way dynamic as the table declaration, in the sense that the generated source code will adapt depending on how many model parameters each generated RMI layer needs.

4.4 How to run the generated code

All together this creates a process where based on a learned RMI configuration the proposed implementation in this chapter can output a P4 and a Python source code file. The idea is to first setup the virtual network via Mininet that contains at least one switch which is executing the freshly generated P4 source code file. When everything is operating correctly the generated Python source code file can be run in order to load model parameters from the saved binary file and send them to the switch using P4Runtime. Finally after this process is complete the switch is able to successfully respond to incoming learned RMI packets formatted as described in the previous chapter in section 3.2.

4.5 Supporting other models

Currently the proposed adaptation does only support the three model types described in sections 4.3.1 and 4.3.2. The reference implementation on the other hand does support additional model types like normal, logarithmic or histogram. There are multiple reasons for why these models are currently not supported.

A first reason and probably the most important one being that these models are hard to implement in P4 because they all rely on some additional mathematical functions which in turn rely on some currently not implemented floating point arithmetic operations. An example being the exponential functions which are used for the normal or logarithmic model types. In my opinion there is no reason why a similar approach than the one taken for the FMA instruction would not work. In other words a software implementation of said mathematical floating point arithmetic functions would probably be doable in P4. Currently though, since time is short and since this project does not necessarily have the goal of being a full copy of the reference implementation, these model types are left aside for future work. Lastly, also quite a major reason coming from a more practically oriented point of view, when looking at all pareto optimal configurations generated by the reference RMI implementation, most of them rely on some combination of linear and cubic model types.

Chapter 5

Experiments

The goal of this chapter is to try to get closer to understand how much the ideas presented in the previous chapters could potentially benefit a concrete setup and implementation in the real world. As already quite extensively discussed in section 3.4 the P4 implementation as is cannot be tested on real world hardware. With this in mind the first section of this chapter will explain the chosen approach to still try to measure something useful in order to evaluate the potential impact. Further, section 5.2 will present the measured results on the test university machine and finally the last section of this chapter will once again try to put the measured results into perspective and give an evaluation.

5.1 Method

As previously stated the following method was chosen since not being able to actually run the generated P4 source code on concrete real world hardware due to multiple reasons also already discussed previously. In order to come up with some alternative we started with the idea that instead of actually measuring lookup time on concrete network hardware we could instead measure the pure lookup time in an existing implementation and go on from there. We would use what we measured as maximally possible speed up. This immediately leads to the already established separation between pure lookup operation time and time spent for the last mile search. For this having the SOSD benchmark that can precisely measure the time it takes for a specific amount of lookups including last mile search on a dataset comes in handy. In order to measure hypothetical maximal time gain it is enough to separate the last mile search and measure only the pure lookup time. The final piece to the puzzle now is that we have to assume that our network switches are able to process packets at a higher or at least a similar rate than a processor can handle last mile searches. This initially seemed like a bold claim to me but when taking into account that switches usually operate at extremely high speed and that the proposed RMI implementation in P4 can easily scale horizontally with the amount of network hardware available, this quickly becomes realistic. Finally if we wanted to look at a closed system and effectively evaluate which method is faster, we would have to consider round trip time of packets. Since not even being able to reliably test any of the implementation on actual hardware, this major concern for a real world setup is neglected in the scope of this work. In that sense we assume an application where packets need to travel over the network and with that over a P4 capable network switch anyways. This leads to travel time spent not really being lost, but instead being used more efficiently.

5.2 Results

5.2.1 Individually measuring pure lookup and last mile search time

In order to perform these measures we adapted the existing SOSD benchmark to measure not only lookup time and last mile search times at the same time but instead differenciate between the two steps and measuring only one at a time. The initial result though, when measuring only pure lookup operation time, is very disappointing. As shown in figure 5.1 only a very small amount of time is spent on performing the pure RMI lookup operations.

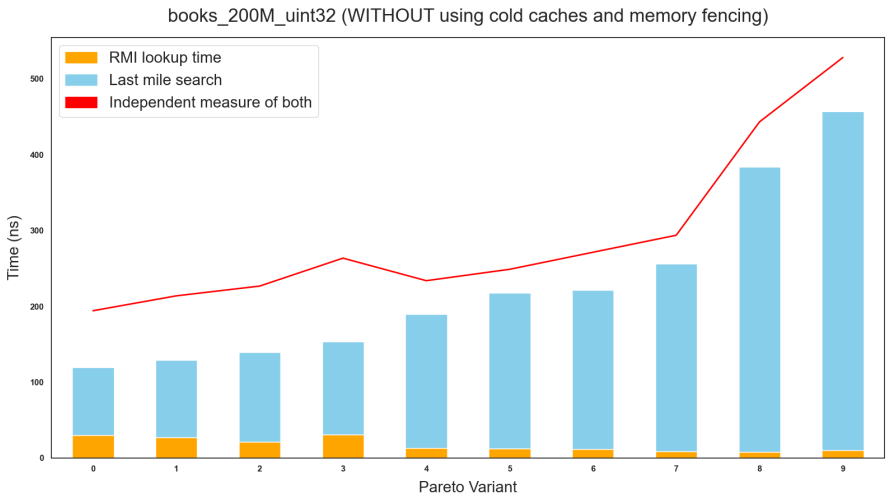


FIGURE 5.1: Running the SOSD benchmark on the *books_200M_uint32* dataset *without using* cold caches or memory fencing, differentiating between pure lookup time and last mile search time.

There are reasons for the way these measurements turned out which were already pointed out in section 4.4 of [10]. Namely these reasons are that lookups in a tight loop can greatly benefit from low level CPU optimization techniques like operator reordering or caching. The same applies for the measurements in the figure above in an even more intense way, since only measuring performance of essentially lots of tightly repeated FMA instructions which the processor will optimize into a more optimal instruction order and therefore exaggerate the measured performance. The same holds true for caching, in the sense that some of the requested data will already be loaded in some cache level and therefore access time is greatly reduced.

5.2.2 Using cold caches and memory fencing

Luckily the authors of [6] suggest and also implemented a way to mitigate these usually very desired CPU optimizations in the SOSD benchmark. The two proposed techniques aim at starting from a fresh CPU state before every lookup calculation. The first technique called cold caching mitigates cache side effects by filling the L3 CPU cache with a constant randomly generated set of numbers before each lookup. The second technique called memory fencing aims at mitigating instruction reordering by introducing memory fences before each lookup using the appropriate CPU instruction. With these techniques in place running SOSD takes a lot longer and RMI performance drastically decreases but with the advantage that more reliable

meaures can be taken. The results from running the SOSD benchmark on the same dataset as previously but now using cold caches and memory fencing are shown in figure 5.2. Very similar observations can be made for all remaining 64-bit datasets provided by the SOSD benchmark shown in the appendix in section 7.5.

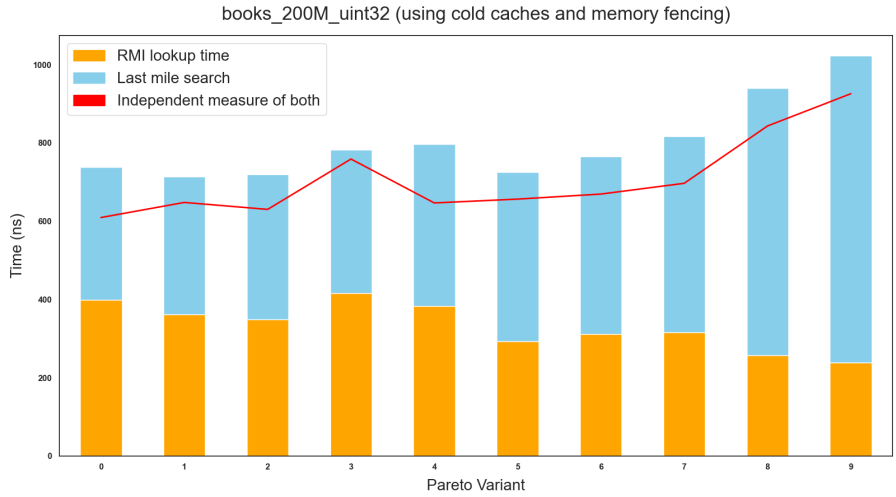


FIGURE 5.2: Running the SOSD benchmark on the *books_200M_uint32* dataset *using* cold caches and memory fencing, differentiating between pure lookup time and last mile search time.

5.2.3 Roughly approximating pure lookup time by the difference of total and last mile search time

Finally the observation which probably leads to the best approximation of how long the pure RMI lookup operations actually take without cold caches and memory fencing, is when taking a normal measure (the red line in figure 5.1) and subtracting the last mile search time from this measure in order to guess the pure lookup operation time. This makes sense under the assumption that the last mile search code is less affected by operation reordering and caching. This is then visualized in figure 5.3. As for the previous experiment, similar graphs can be plotted for the remaining 64-bit datasets provided by the SOSD benchmark shown in the appendix in section 7.6.

5.2.4 Experiment observations

When looking at the experiment figures, one can observe that the height of the blue bar (representing the last mile search time) increases with increasing pareto variant. This is expected since the SOSD benchmark trains its RMIs with decreasing index size limit relative to increasing pareto variant, meaning that with decreasing index size the last mile search bound becomes larger and therefore time spent to find a key in said larger bound increases. A next observation which is pointed out in [10] is that the intensity at which RMI benefits from low level CPU optimizations heavily depends on many factors such as the shape of the data itself and the actual optimization capability of the processor doing the calculations. This forcibly means that any measure becomes very heavily application dependant and therefore the best advice is still to test different (learned) indexing algorithms individually in concrete applications. Still, a regular CPU can highly improve RMI performance in practice, while

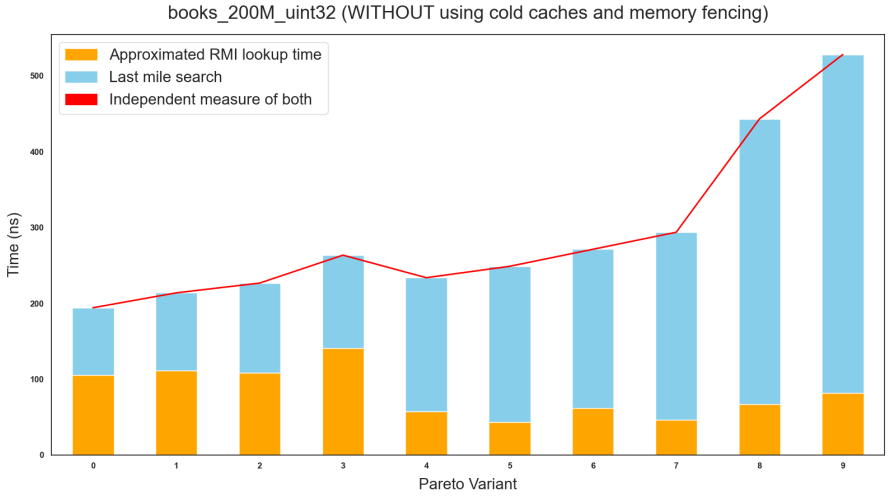


FIGURE 5.3: Running the SOSD benchmark on the *books_200M_uint32* dataset *without using* cold caches and memory fencing, trying to approximate pure lookup time by subtracting the last mile search time from the independently measured total time.

switches do not currently support any sort of CPU optimization techniques at this level. Concretely this means that it makes sense to compare measures from figure 5.2 without any CPU optimizations at play, when directly trying to compare a server's RMI lookup performance against a switch's RMI lookup performance. Due to reasons already pointed out multiple times, this comparision is unfortunately currently not possible in practice. On the other hand, when opting for a comparision under ideal conditions, we can refer to figure 5.3, especially if we want to know how fast our university machine can handle last mile searches only.

5.3 Evaluation

After having described our measurement method in the first section and showing visualized results in the previous section, this section focuses on giving potential take-aways from what we measured and showing that outsourcing a learned index structure to the network can lead to a speed up. The major drawback that this work makes apparant is that there are lots of requirements currently missing in real world hardware. Namely these consist of what has been described in section 3.4.

Our university machine's CPU takes around 80 - 400ns for a full RMI lookup, including pure lookup operations as well as last mile search, without cold caches and memory fences (Refering to figure 2.1). This means that our machine can handle between 2.5 - 12.5 million full RMI lookups per second. When now looking at the graphs in figure 5.3 and section 7.6 we find that the pure RMI lookup calculations take pretty constantly around 50 - 100ns depending on the RMI layer configuration. Further, our university machine which would only have to deal with last mile searches and could outsource the pure lookup calculations to the switch could handle between 3 - 20 million last mile searches per second and takes around 50 - 300ns for a single last mile search.

All in all we find that by outsourcing the RMI calculations to the switch we can constantly gain around 50 - 100ns per lookup depending on the RMI configuration but independently of the RMI's index size and with that it's last mile search bound size, which means that without performing the RMI lookup calculations a server can constantly handle more lookups per second (since the server only has to perform the last mile searches). Further, if only a single Intel® Tofino™ 3 switch can provide pure lookup operations at a rate of around 10 billion packets per second [4], then this switch would easily outperform multiple magnitudes of servers working on last mile searches. This means that in a closed system the amount of speed gained, when outsourcing RMI to the network, is determined by the amount of last mile search workers. Each of them can treat more last mile searches per second and adds to the overall speed up. Additionally the constantly less time not spent on the server for calculating pure lookups would be mostly free, especially when having an application in mind where a lookup has to be sent over the network anyways. Finally, when reaching a point where a single switch cannot outperform all of the available last mile search workers anymore, horizontal scaling in terms of network switches is perfectly compatible with our RMI implementation in P4.

Chapter 6

Conclusion

In this work we started by evaluating different learned index structures by their potential programmability in P4. We compared performances of different (learned) index structures using the SOSD benchmark [6] and looked at what is possible as well as what the limits of network programmability are. We came to the conclusion that the RMI learned index structure initially proposed in [7] is a good fit for further pursuing our idea of implementing a learned index structure in P4 in order to run lookup calculations on the fly over the network.

In a next step we implemented a proof of concept by hand in P4 testing it's operability on virtually simulated network hardware and found that an RMI implementation with perfect accuracy in P4 is indeed possible. At the same time though, we learned what limitations apply on real world switch hardware that do not exist in simulated hardware which proved to be crucial for our implementation. This leads to probably the most pertinent conclusion of this work, namely that in order for learned indices to become viable solutions in real world applications in the future, switches need to be able to support floating point arithmetic ideally on their ALUs or even FPUs or in some other computationally cheap form. In the same sense another limitation which we are facing, especially strengthened due to the fact that we naïvly implemented floating point arithmetic in software, is that real world switches are limited by the amount of ALU stages, meaning that chained operation complexity for a single packet is pretty quickly exploited. After having implemented a proof of concept that worked for a specific dataset and a specific RMI configuration, we further opted for a more generalizable solution, by adapting the RMI reference implementation [9] in a way that P4 source code files can be generated for any input dataset and mostly any RMI configuration.

Finally in a last step, we tried to circumvent the fact that our implementation is theoretical and tried to estimate, how much an RMI implementation over the network could save in a closed system. We come to the conclusion that when assuming ideal conditions, where an RMI implementation in P4 exists under which a switch can operate at state-of-the-art packet processing rates in an application where a lookup has to be sent over the network anyways, our idea could potentially leverage each last mile search worker by a constant magnitude of around 50-100ns per lookup depending on the RMI layer configuration.

6.1 Future Work

The continuation of this work would initially consist of finding solutions for the different limitations and assumptions that were taken in the scope of this work in order to arrive at a real world learned index structure implementation over the network. One solution for floating point arithmetic could come for instance, from an immensely interesting paper [1] about enabling accurate floating point arithmetic on P4 capable switches in a smarter way than we did for this work. Unfortunately, the results are currently limited to 16-bit floating point arithmetic and everything beyond is left as future work. Further there is another paper [8] which proposes an alternative implementation for finding RMI configurations that yield similar lookup times and often better build times. The result of this alternative implementation could be used as input to the P4 code generation part of our adaptation of the reference implementation. This does not immediately lead to a solution of one of the problems shown in this work, but could further leverage the performance of RMI. Finally when looking at all limitations and assumptions that we discovered during this work, most of them could either be solved or at least greatly weakened when having hardware supported floating point arithmetic. In that sense a very systems related but extremly interesting start for our future work could be to try to extend the capabilities of network switches or potentially NICs to support floating point arithmetic. The topic is currently already quite highly frequented and there are already papers [17] which pursue this exact idea of applying hardware enhancements in order to allow efficient floating point arithmetic.

Generally evaluating other or in the future also new learned index structures regarding their network programmability will continuously be a good starting point for our future work. In that sense pursuing the idea of having a learned index structure which does not rely on floating point arithmetic could be worth exploring.

Bibliography

[1] Penglai Cui et al. *NetFC: enabling accurate floating-point arithmetic on programmable switches*. 2021. DOI: 10.48550/ARXIV.2106.05467. URL: https://arxiv.org/abs/2106.05467.

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

[3] IEEE. *IEEE Standard for Floating-Point Arithmetic*. https://standards.ieee.org/ieee/754/6210/. 2019.

[4] Intel. *Intel® Tofino™ 3 Intelligent Fabric Processor Brief*. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/product-brief-final-version-pdf.pdf.

[5] Andreas Kipf et al. "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*. 2020, 5:1–5:5. DOI: 10.1145/3401071.3401659. URL: https://doi.org/10.1145/3401071.3401659.

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

[7] Tim Kraska et al. "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, 2018, 489–504. ISBN: 9781450347037. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

[8] Marcel Maltry and Jens Dittrich. *A Critical Analysis of Recursive Model Indexes*. 2021. DOI: 10.48550/ARXIV.2106.16166. arXiv: 2106.16166 [cs.DB]. URL: https://arxiv.org/abs/2106.16166.

[9] Ryan Marcus, Emily Zhang, and Tim Kraska. "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, 2020, 2789–2792. ISBN: 9781450367356. DOI: 10.1145/3318464.3384706. URL: https://doi.org/10.1145/3318464.3384706.

[10] Ryan Marcus et al. "Benchmarking Learned Indexes". In: *Proc. VLDB Endow.* 14.1 (2020), pp. 1–13.

[11] Mininet Project Contributors. *Mininet*. http://mininet.org/. 2021.

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

[13] Ryan Marcus. *RMI*. https://github.com/learnedsystems/RMI. 2020.

[14] The P4 Language Consortium. *P4 Specification*. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html. May 2021.

[15] The P4 Language Consortium. *P4Runtime Specification*. https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html. Dec. 2020.

[16] The P4 Language Consortium. *The reference P4 software switch*. https://github.com/p4lang/behavioral-model. 2020.

[17] Yifan Yuan et al. *Unlocking the Power of Inline Floating-Point Operations on Programmable Switches*. 2021. DOI: 10.48550/ARXIV.2112.06095. URL: https://arxiv.org/abs/2112.06095.

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 Normalization

```
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();
  }
}
```

7.2.4 FMA

```
control FloatingFusedMultiplyAdd(in double_t x, in double_t y, in double_t z, out double_t result) {
  FloatingAdder() adder_instance;
  FloatingMultiplier() multiplier_instance;

  apply {
    multiplier_instance.apply(x, y, result);
    adder_instance.apply(result, z, result);
  }
}
```

7.3 Loading model parameters

7.3.1 Data plane RMI table declaration in P4

```
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; // depends on model parameter size
  }
  apply {
```

```
      assign_variables(0, 0, 0, 0, 0, 0, 0);
      model_lookup.apply();
  }
}
```

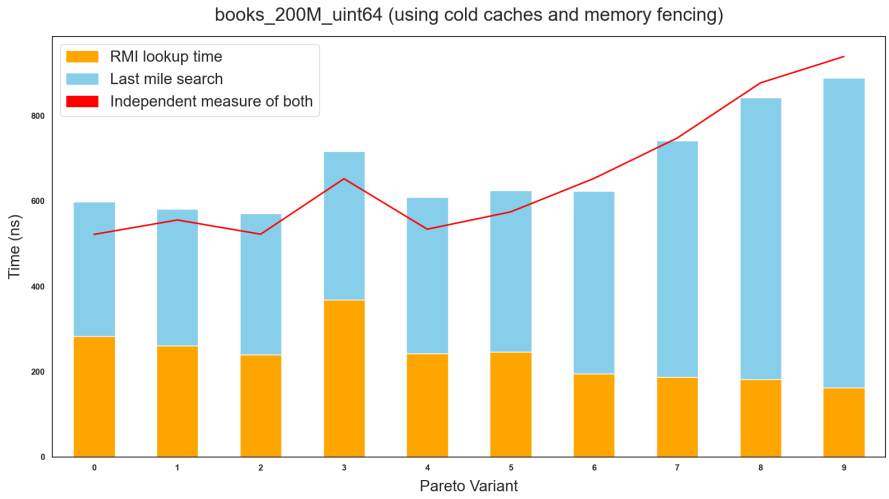
7.3.2 Control plane batch sending of model parameters in Python

```
BATCH_SIZE = 2048

SIGN_MASK = 0x8000000000000000
EXPONENT_MASK = 0x7FF0000000000000
MANTISSA_MASK = 0x000FFFFFFFFFFFFF

def writeL1Parameters(p4info_helper, switch):
  if not os.path.exists('path/to/layer_parameters'): print('Parameters file not found!'); return
  model_index = 0
  with open('path/to/layer_parameters', 'rb') as file:
    bytes = file.read(24 * BATCH_SIZE)
    while bytes:
      entries_batch = []
      for index in range(0, BATCH_SIZE):
        model_bytes = bytes[(24 * index):(24 * (index + 1))]
        first_l1 = int.from_bytes(model_bytes[0:8], byteorder='little')
        second_l1 = int.from_bytes(model_bytes[8:16], byteorder='little')
        third_l1 = int.from_bytes(model_bytes[16:24], byteorder='little')

        table_entry = p4info_helper.buildTableEntry(
          table_name='LearnedIngress.lookup_instance.l1_lookup.l1_model_lookup',
          match_fields={ 'model_index': model_index },
          action_name='LearnedIngress.lookup_instance.l1_lookup.assign_variables',
          action_params={
            'first_l1_sign': (first_l1 & SIGN_MASK) >> 63,
            'first_l1_exponent': (first_l1 & EXPONENT_MASK) >> 52,
            'first_l1_mantissa': first_l1 & MANTISSA_MASK,
            'second_l1_sign': (second_l1 & SIGN_MASK) >> 63,
            'second_l1_exponent': (second_l1 & EXPONENT_MASK) >> 52,
            'second_l1_mantissa': second_l1 & MANTISSA_MASK,
            'third_l1_input': third_l1
          })
        entries_batch.append(table_entry)
        model_index += 1

      switch.WriteTableEntries(entries_batch)
      bytes = file.read(24 * BATCH_SIZE)
```

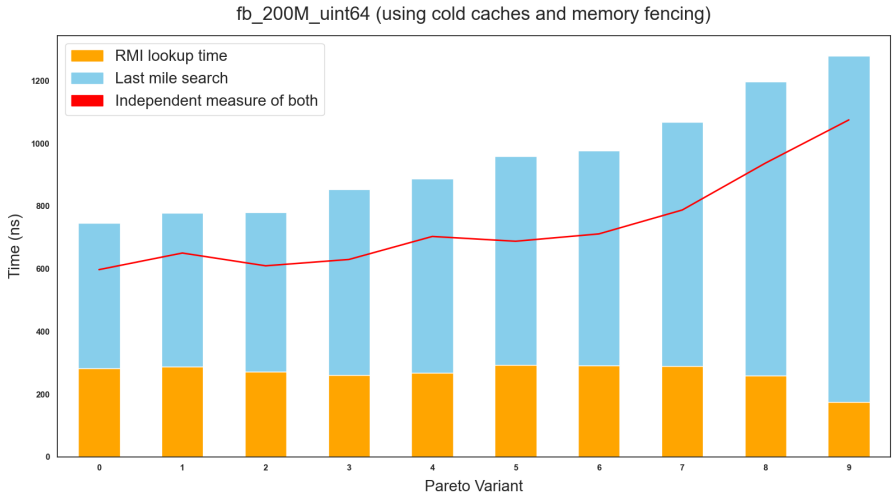
7.4 RMI lookup fuction in P4

```
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 {
    // 1. using static L0 parameters as input for the first cubic model layer
    cubic_instance.apply({ 0, 0, 0 }, { 0, 0, 0 }, { 0, 1014, 4503599092596736 }, { 0, 0, 0 }, input_key, fpred);
    double_to_int(fpred, model_index);

    lookup_instance.apply(model_index, first_l1, second_l1, guess_err); // 2. retrieving L1 parameters

    // 3. using retrieved L1 parameters as input for second linear model layer
    linear_instance.apply(first_l1, second_l1, input_key, fpred);
    double_to_int(fpred, guess);

    f_clamp(fpred, 0xbebc1ff, guess);
  }
}
```

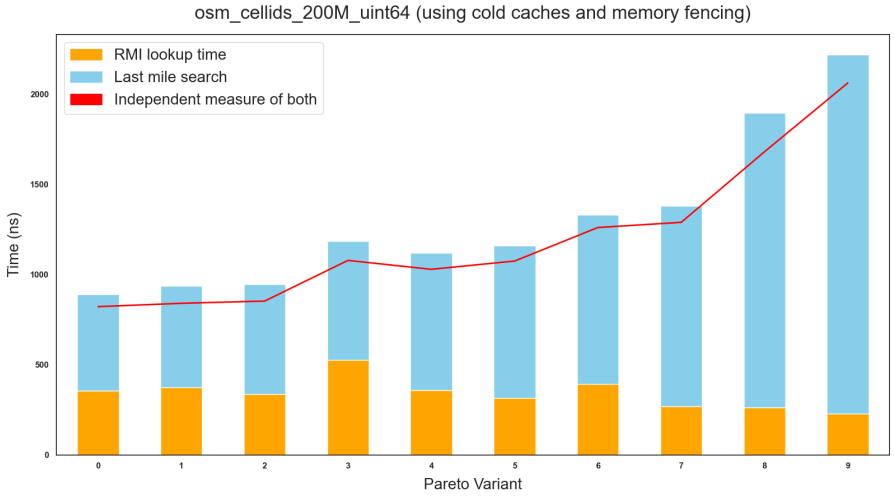
7.5 Experiment results using cold caches and memory fencing



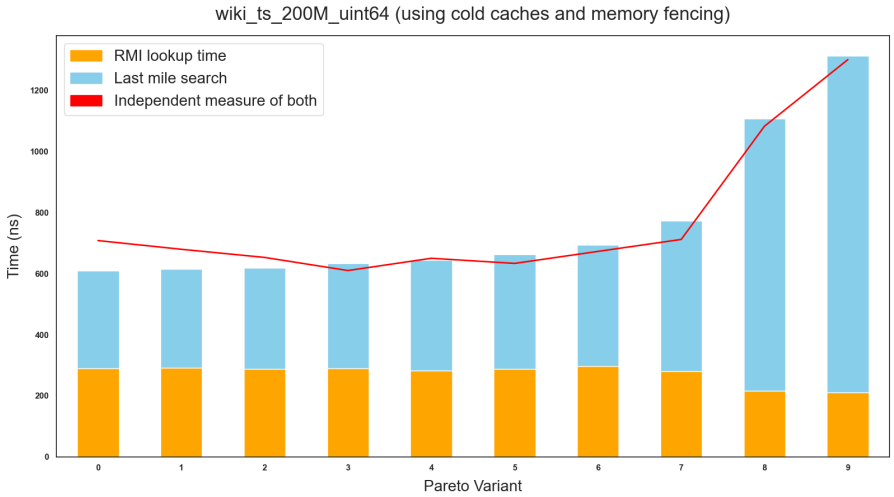
Running the SOSD benchmark on the *books_200M_uint64* dataset *using* cold caches and memory fencing, differentiating between pure lookup time and last mile search time.



Running the SOSD benchmark on the *fb_200M_uint64* dataset *using* cold caches and memory fencing, differentiating between pure lookup time and last mile search time.

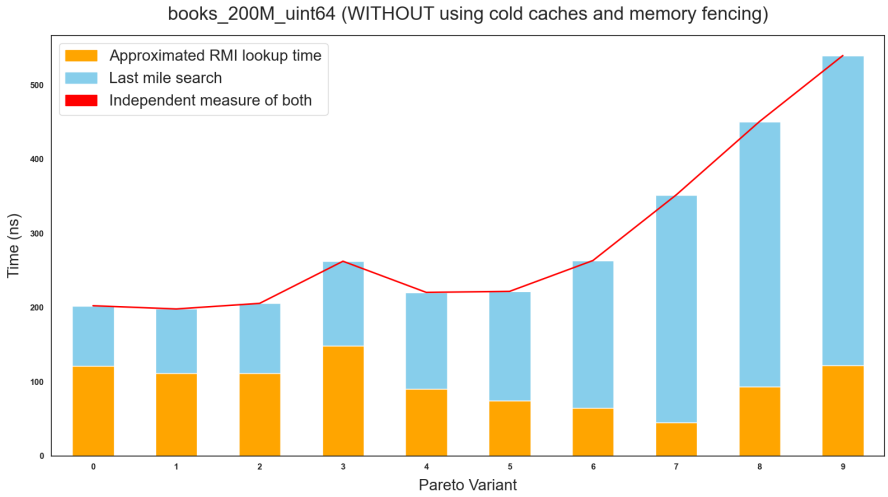


Running the SOSD benchmark on the *osm_cellids_200M_uint64* dataset *using* cold caches and memory fencing, differentiating between pure lookup time and last mile search time.

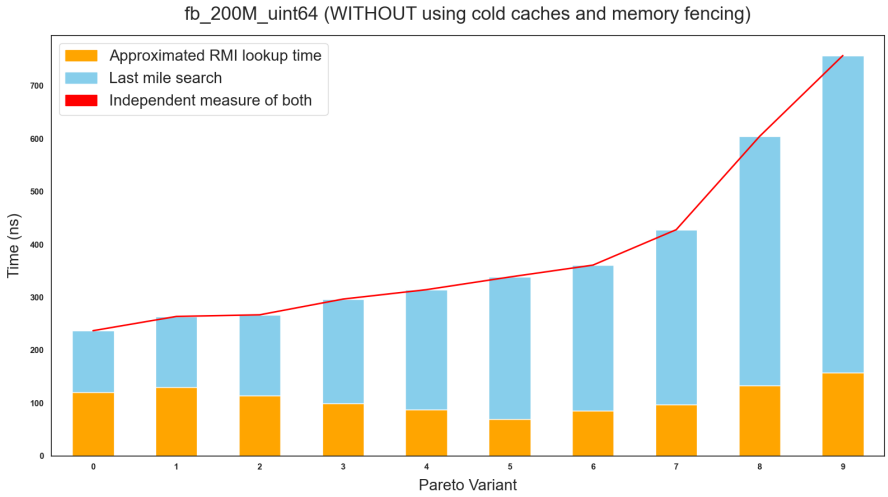


Running the SOSD benchmark on the *wiki_ts_200M_uint64* dataset *using* cold caches and memory fencing, differentiating between pure lookup time and last mile search time.

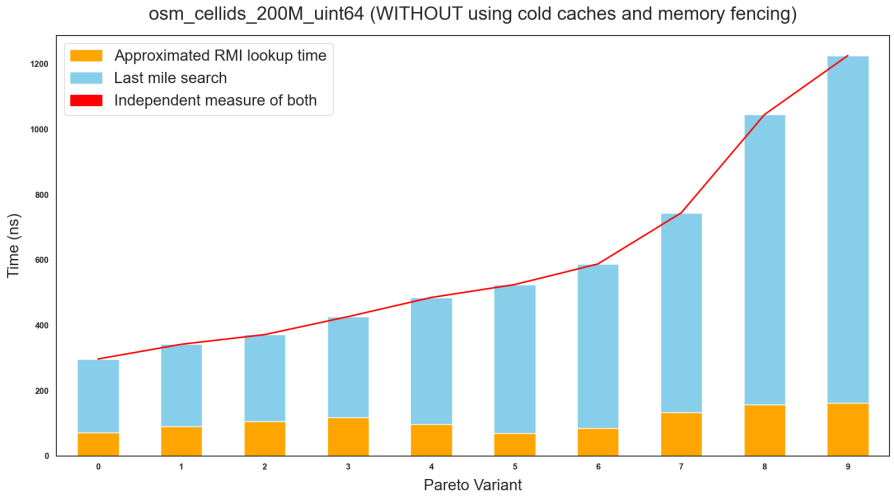
7.6 Experiment results calculating pure lookup time as the difference of the total time measured and last mile search time



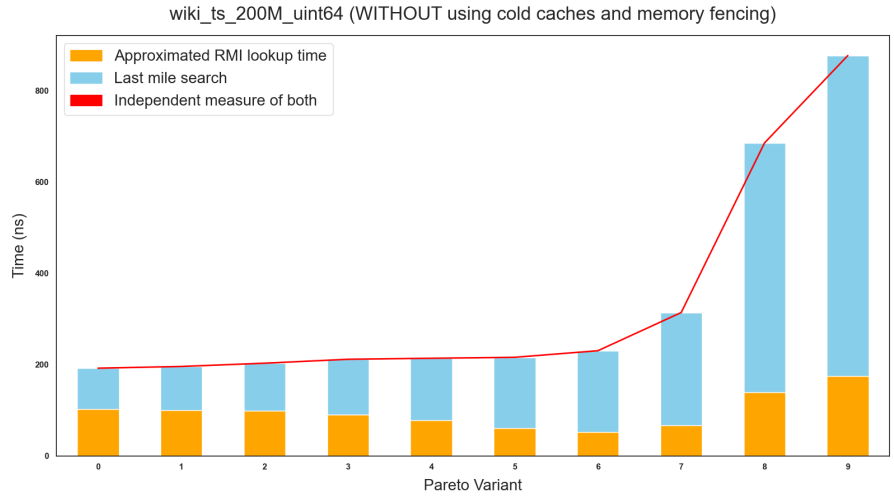
Running the SOSD benchmark on the *books_200M_uint64* dataset *without using* cold caches and memory fencing, trying to approximate pure lookup time by subtracting last mile search time from the totally measured time.



Running the SOSD benchmark on the *fb_200M_uint64* dataset *without using* cold caches and memory fencing, trying to approximate pure lookup time by subtracting last mile search time from the totally measured time.



Running the SOSD benchmark on the *osm_cellids_200M_uint64* dataset *without using* cold caches and memory, fencing trying to approximate pure lookup time by subtracting last mile search time.



Running the SOSD benchmark on the *wiki_ts_200M_uint64* dataset *without using* cold caches and memory fencing, trying to approximate pure lookup time by subtracting last mile search time from the totally measured time.