AN IN-SWITCH ARCHITECTURE FOR LOW-LATENCY MICRO-SERVICES

An Undergraduate Research Scholars Thesis

by

JACKSON PETROLL

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Riccardo Bettati

May 2022

Major: Computer Engineering

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Jackson Petroll, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty, Riccardo Bettati, prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

An In-Switch Architecture for Low-Latency Microservices

Jackson Petroll
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Riccardo Bettati
Department of Computer Science and Engineering
Texas A&M University

In recent time, there is has been a movement away from standard monolithic architecture in cloud and web services towards what is known as a microservice architecture. Microservice architecture decomposes the previous monolithic architecture into multiple independent services called "microservices". Examples of applications that use a microservice architecture include Netflix and Amazon [1]. These applications typically send large numbers of microservice requests, which go through the OSI network layers to establish a client server connection. This trend towards microservices has developed interest by other researchers to make improvements in this field, due to the growing reliance importance on such architectures by consumers. There have been studies regarding the security of these microservices, performance analysis of various applications, and the use of these microservice applications in cloud technology. Any improvements in the speed, security, or organization of such network architecture would be very beneficial of these popular API's, and their user base. This project's objective is to investigate the potential of moving some of the processing that is done for these microservices within a network switch, and as a result the performance at the application level, by alleviating network communication. We formulate a high-level design for an in-switch architecture for low-latency microservice leverag-

ing existing programmable-switches support. We investigate the implementation of NetCache as a microservice in our model and predict a significant latency reduction and subsequent performance increase.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Riccardo Bettati, for their guidance and support throughout the course of this research.

Thanks also go to my colleagues and the department faculty for making my time at Texas A&M University a great experience. I have enjoyed the wide diversity of course offered, which has helped in finding and pursuing my specific interest. The faculty has always been very supportive of my academic pursuits, and I appreciate everything they have done for me.

The data analyzed/used for this paper were provided by the specified publications.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

No funding for this thesis was received.

1. INTRODUCTION

As cloud computing has become more pervasive, designers have been on the lookout for software and systems architectures that allow computing applications to grow more scalably, both in terms of numbers of clients as well as numbers of developers and users. One popular such architecture is microservices, where the application is partitioned into a set of interoperable services called microservices, each with a well defined application program interface (API). A microservice is an alternative to monolithic architectures, where the codebase is hosted on one singular server. Microservices give rise to a software cloud architecture structure whereby applications are constructed as a collection of services or API's. These microservices can be deployed, scaled, and tested independently, as each of them maintains a single responsibility or function [2]. In recent times, there has been a growing movement away from monolithic applications and towards a microservice architecture as modern cloud applications evolve [3]. The microservice architecture has become popular and used in large web applications such as Netflix, Amazon, and Ebay being used by millions of users making billions of API calls [1].

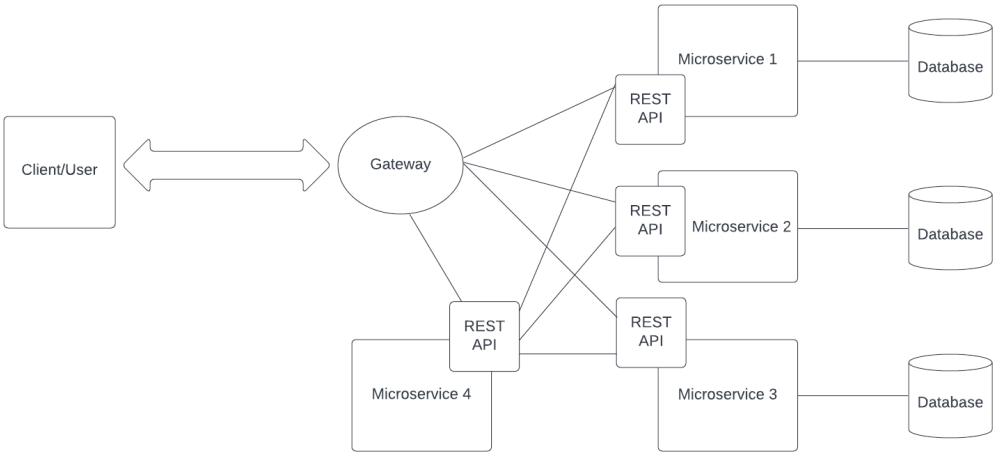

Figure 1.1: Gateway Microservice Architecture.

Microservices vary in network architectures, most use either gateways, service meshes, or a combination of the two [4]. In a basic gateway topology, there is a host machine, the gateway, and a client for each microservice. The host application sends requests over the internet to the gateway, whereby the requested microservice then enters a gateway [2]. The gateway is its own separate instance that receives API request from users and retrieves results to return. Depending on the service being requested, the gateway will send the request to the associated microservice to perform the associated function. Microservices are typically hosted on server rack clusters hosting virtual machines that run these microservices. The microservice itself is composed of one set service. This service could vary from taking inventory too managing account information, so long as it provides one service. This microservice may potentially communicate with other microservices and servers through API calls as depicted in **Figure 1.1**. Upon completing the request, the gateway retrieves the result and sends it back to the client. The service mesh differs from the gateway topology. The service mesh network pattern creates an interconnecting 'mesh' between the various micro-service proxies called 'sidecars'. Sidecar proxies are essentially service proxies that are hosted in a container that runs alongside the service container providing security and various other control functions between the associated microservice and network mesh [5]. The network mesh enables microservices to communicate with one another while maintaining security. While mesh architecture is a relevant architecture, it is rarely seen implemented stand alone, and is usually integrated with the gateway architecture. **Figure 1.1** depicts a typical microservice application layout. As described, there is a client/user that communicates to a gateway and then to the subsequent microservice. This microservice may utilize a database and may communicate with other microservices depending on its design. In comparison microservices divide monolithic applications into individual microservices that are hosted on separate servers. While there are many benefits to microservice applications, the network architecture suffers from a variety of different issues namely centered around complexity. A set of solution criteria can be defined from the issues present in microservices, and we hypothesize in-switch computation as a way to address this criteria specifically in the context of low latency microservices. There has been similar work

done utilizing in-switch computing to address latency issues, but none specifically in the context of microservice applications.

1.1 Microservice Drawbacks

While addressing the problems related to monolithic applications, microservice applications suffer from a variety of different issues namely centered around complexity. Microservices suffer in terms of network and system complexity, security, and end-to-end system latency [1]. Dividing monolithic components into smaller services means that the interconnection complexity of communication, security, and error handling of the system is increased. Having many different microservices means having a more points of failure. The computational model for the average failure rate of a microservice application is defined by the number of microservices for the application, the failure rate of each microservice, and the number of request being made to a microservice. Thus, increasing the number of microservices inherently increases the average failure rate of the application. In the case of a failure or error in a microservice application, if one microservice stops working the other microservices still work. Handling this situation makes the development of these microservices more complex. With movement from monolithic to microservice architecture, code that was once never accessible from the outside is now exposed through API's. This creates multiple venues for cyberattacks on the service, virtualization, and network levels [6]. Service level security problems such as broken authentication, SQL injection, and others, can affect each microservice individually. The increased number of microservices increases the likelihood of service specific attacks occurring. Virtualization and deployment of individual microservices affect security as well. Most microservices are deployed on top of operating systems with multiple services on the same system. These services are separated by containers and VMs. While this does offer more protection again compromised services on the same system, a variety of attacks exists to harm the system still. Example of these attacks include Hypervisor compromise, shared memory attacks, and the use of harmful images [6]. Virtualization of many microservices on one system increase the likelihood of being affected by these threats. Finally, each microservice is also subject to communication level threats. This includes attacks on network stacks and proto-

cols specific to the API the microservice is using for communication such as REST, SOAP, and others. Attacks such as spoofing, eavesdropping, Man-in-the-Middle and Denial of service can all affect a single microservice [6]. These threats are all the result of dividing monolithic services into individual microservices and exposing more avenues of attack. Dividing Monolithic applications into microservices introduces performance overhead on the communication done over the network. In services using REST API, for example, the calls add over head to the network latency of inter-service communication. The more complicated the microservice application, the higher the network complexity and subsequent number of network jumps being performed. Finally, there is end-to-end network latency incurred from traveling the OSI layers of the Host machine.

1.2 Addressing Microserivces

In this work we focus on architectural support to improve the performance of microservices. As we identify a solution, we need to keep in mind a set of criteria:

1. *Reduce the end-to-end system latency of microservice invocation.* There is latency incurred from the system that a microservice is hosted on as packets travel up the OSI layer and kernel of a microservice. Reducing the end-the-end system latency would mean reducing the time spent from the source to the host and up the host machine's system.
2. *Reduce network communication overhead.* There is communication overhead generated from the complex network interconnections formed in a microservice architecture. Reducing the communication overhead would mean reducing the overhead generated traversing through this complex network connection.
3. *Transparent to the user.* A microservice must be transparent to the user, meaning that it is hidden from the user. The whole microservice application will appear as one application to the user despite being divided into individual services.
4. *Maintain the benefits of a microservice application over monolithic application.* Any improvement to a microservice application will maintain the previous benefits that it had over monolithic applications.

1.3 Supporting Microservices Through In-Switch Computing

Our paper proposes a possible solution to these issues through the use of in-switch computing. In-switch computing is a method of performing application-level functions within network

switches. Instead of the typical rerouting, our model of a switch would now have the role of performing basic micro-service computations, by reading the payload of an HTTP protocol used by REST, SOAP, and other API's. We argue that a significant number of these microservice functions are so simple one can implement them on a network switch directly. The in-switch model we propose has the potential to reduce latencies end-to-end service request latency by reducing the required number of network jumps and latency incurred by the server.

1.4 Related Work

Transitioning computation workloads from computing nodes to the network infrastructure has been used in many different forms under many settings.

1.4.1 In Switch Computing and Accelerated Distributed Reinforcement Learning

In a recent study [7] by the authors introduced a model of in-switch computing for Accelerating Distributed Reinforcement Learning. They presented an in-switch accelerator model, whereby the gradient aggregation, a common step in the development of machine learning models, is moved into a network switch. As a result, the authors are able to reduce the network and server latency that would otherwise be incurred if the gradient aggregation was computed on a computational node. For the synchronous approach there was a speed up of 1.72–3.66×; similarly, the asynchronous approach provided a speed up of 3.71× for distributed training.

1.4.2 Programmable Switches

Software-defined networks rely on the ability to flexibly deploy forwarding and control plane capabilities onto some form of programmable switch. P4[8] and PISCES[9] are examples of such Programmable switch architecture. P4 was a language protocol developed to support SDN and programmable switches, and it allows a programmer to define the forwarding plane in a switch. Subsequent simulators for SDN have been developed to help model and test these SDN switches. To name a few: PFPSIM [10], NS4 [11], and Druzhba [12] are simulators that have all been developed to test programmable switches in some manner. PFPSIM is a programmable forwarding plane simulator for software defined networks. It uses leverages P4 as described to run and sim-

ulate the architecture of programmable switches. PFPSIM advertises automatic model generation of complex switches, fast and scalable host-complied simulation for pre-silicon analysis, and easy debug and optimization of the application on a target model. NS4 is similar to PFPSIM, however, it allows the entire emulation of P4-enabled networks. Finally, Druzhba is a simulator like NS4 and PFPSIM, but focuses on modeling low-level details of the switch pupline instruction set for testing compliers.The low-level modeling allow Druzhba to simulate machine code. P4 also offers simulation of through the use of mininet network simulator, a p4 compiler, and python. The p4 code can be compiled to target a specific SDN architecture. Our implementation of a switch give the switch more processing capability around specific API calls. This is where our switch implementation differs from the current switch models out there.

In recent years, network designers and operators have acknowledged the need for a more flexible control plane in the network, in order to accommodate needs across a wide area of data-center networks. This has given rise to so-called software-defined networks (SDNs) [8], which allow many network operations to be flexibly implemented in software. Programmable switches are used to implement more functionality to existing switches. The primary benefit of software defined networking it that it enables centralized network control by physically separating the control plane from the forwarding plane. This enables changes in the networking control data-plane to occur without a redesign on silicon that would previously occur.

2. IN-SWITCH COMPUTING FOR MICROSERVICES

As we pointed out earlier, a user-transparent way to reduce the latency of microservice invocation is to eliminate the majority of the non-core portions of the microservice processing (i.e, request routing, forwarding, protocol processing). One way to achieve this is to move the microservice processing into the network switching, with the use of in-switch processing.

2.1 Design Criteria

The creation an implementation of such an in-switch processing capable switch needs to meet the following design criteria:

1. *Identify and forward packets to local processing unit.* API call headers for the given microservice are able to be extracted and identified. Once Identified the packet is forwarded to a processing unit where the microservice is performed.

2. *Standard forwarding of non-identified packets.* The inclusion of standard forwarding of Non-identified packets should incur minimal delay from the addition of microservice processing. The switch should effectively still act as a switch for non-identified packets.

3. *Transparent processing of identified packets and generation of response packets.* Identified packets should add minimal latency to standard switch processing, effectively being invisible to switch users. Identified packets should be packaged back together into a packet to forward the response appropriately.

2.2 In-Switch Solution to Microservices

In switch computing is a form of in-network computation that is localized within a network switch. In-switch computation is typically used to perform specific packet level processing that benefits from localization within a switch. The typical in-switch model, for example, queues incoming packets from the MAC port, and then allows for definition of parsing, match-action tables, and deparsing. In-switch computation has been applied to higher level packet processing as well. We describe above how In-switch computing has been exploited and applied to other applications such as gradient aggregation of distributed reinforce learning [7] and we will describe in the following how programmable switches have been used to host simple Netcached server [13]. In both of these papers, the switch acts as the host machine for processes that would otherwise run

on server or more complicated machine. In-switch computing is done on programmable switches with the intention of enabling users to change how the switch processes packets. The advantage of this is that the programmers are able to change the way switches process packets, makes switches more generalized and not tied to one specific protocol, and have target independence. *Target independence* means that the programmer is able to describe packet functionality independent of hardware. The introduction of programmable switches are what facilitate in-switch computation. Prior to programmable switches, switches were defined in hardware and did not allow for changes in the switch packet processing to occur without the redesign and testing of the silicon [2].

We hypothesize that an in-switch computation can be applied to meet the solution criteria for low latency microservices. This is because of the inherent advantages of switches in the network model. Network switches are in a position in the network model where all of the network traffic that travels is processed and directed right before reaching the target machine. It has been observed in other uses of in-switch computation that in-switch application models can be used to solve network communication overhead, are easily scalable, utilize rack-scale network architecture, and reduce end-to-end system latency. In the context of microservices, in-switch computation fits the solution criteria that is proposed.

2.2.1 OS Delay

Microservices are typically hosted on server rack clusters hosting virtual machines that run these Microservices. These virtual machines utilize some form of operating system that contains a kernel. Gan et al. [14] analyze the end-to-end latency of microservices by creating a movie streaming service application. This sample application makes use of 33 unique microservices, which make use of popular open source microservices including nginx, memcached, mongoDM, xapian, and node.js [14]. The authors analyzed the os vs. user-level cycle breakdown of their test application. They found that a large fraction of the execution occurred in kernel mode and another fraction going towards other libraries like libc, libgcc, and others. This is because microservices such as memcached and MongoDB spend a majority of their execution time in the kernel, handling interrupts, processing TCP packets, and scheduling request [14]. At low loads,

the TPC processing and network traversal would take a majority of the execution time. This is because the microservices are sufficiently simple that there is hardly any processing done. Because a significant number of request to different microservices are invoked in sequence to satisfy as single user request, in-switch computation solves this problem by effectively eliminating the OS and associated virtualization elements as well as packet forwarding to the servers hosting the microservices. Packets would be identified as microservice requests and immediately processed in the switch. The end-to-end latency would be significantly reduced due to the lack of system layers and network elements to traverse to traverse, and allow for this low latency microservices to be optimized.

2.2.2 Network Delay

Microservices communicate between the gateway, servers, and other microservices using API calls. This subsequently results in a high network communication overhead and makes up a majority of the latency for simple microservices. By localizing the microservices within the switch, this intuitively reduces the number of network hops required to communicated between gateways, servers, and other microservices. In a network model, packets typically go through a gateway through a switch, and then to the service. However, in an in-switch computation model, the packets travel to the switch, are processed, and then rerouted back to the client or to another microservice. This would reduce the number of jumps performed in the network communication between microservices by one jump per microservice. This would make a big difference in latency especially for simple low latency microservices where the network overhead makes up the majority of the latency besides the end-to-end system latency. This can be depicted mathematically through the equation of the Network transmission Latency among microservices [15]. This equations shows that the network transmission latency among microservices is related to four key factors: (1) The number of requests between the two microservice instances, (2) the bandwidth, (3) network distance between edge node microservices, and (4) the size of the data transmission in a request between two microservice instances [15].

$$transLatency = \sum_{k=1}^{m}\sum_{q=1}^{n} y_{k,q} \sum_{i=1 \wedge ms_k \varepsilon preset_i}^{m} \sum_{p=1}^{n} y_{i,p} * lc(i,k,p,q), \quad \text{(Eq. 2.1)}$$

$$lc(i,k,p,q) = \frac{link(ms_i, ms_k)}{scale_i * scale_k} * (\frac{trans(ms_i, ms_k)}{b_{q,p}} + \frac{d_{q,p}}{c}) \quad \text{(Eq. 2.2)}$$

Refer to Appendix A for variable descriptions of this equation. The application model is modeled as a graph $G_a =< msSets, msRelation >$ where $msSets$ is the microservices of the micoservice and $msRelation$ is the dependencies between the microservice applications. The microserivce ms_i is defined as a tuple of the required computation resources, storage resources, and maximum number of request the microservice can manage $< calcNeed_i, strNeed_i, maxLink_i >$. In the underlying network model of a microservice, the environment is defined by fully-connected directed graph of nodes $G_e =< nodeSet, linkSet >$ where nodeSet is the set of nodes and link set is the set of links between these nodes. Each link between two nodes has a related bandwidth and network distance. A node is characterized as a tuple of the the computing capacity, the storage capacity, and the failure rate $< calc_j, str_j, fail_j >$. $lc(i,k,p,q)$ is the latency between two microservices deployed on two network nodes. We can include switches in this model in two ways depending on the assumptions made. If the network model considers switches to be a node, then by moving a microservice within the switch node, we effectively reduce the number of nodes and subsequently the overall latency of the application. If switches are not considered a node, then the latency is apart of the the communication link between each node. In this instance, when a microservice is moved within a switch, the network distance is reduced. In any case, moving microservices within a network switch reduces the network transmission latency and overall latency of the application.

2.2.3 User Transparency

An in-switch implementation for microservices would supply user transparency for the application by containing the microservice within the switch. while the microservices would be debuted in switches, the entire application would be appear as one to the user.

2.2.4 *Maintain the Benefits of a Microservice Application Over Monolithic Applications*

An in-switch implementation would maintain the benefits of a microservice application over monolithic applications so long as it provides ease of deployment, testing, and scalable. This means easy access to the individual microservice codebase on the switch.

2.2.5 *Netcache: An In-Network Key-Value Store on a Network Switch*

Switches have been recognized as a solution to solve low latency in-network computation in areas similar to microservices. Netcache is one such implementation, whereby a programmable switch is utilized as an in-network cache to handle queries on hot items [13]. Netcache is essentially a simpler implementation of memcached that targets hot items in an attempt to solve load imbalance in other key-value store implementation. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. Switches are optimized for data input-output and maintain lower latencies when compared to tradition network caches and servers. Netcache is implemented on a programmable network switch through the use of P4. P4 allows for easy implementation of packet format, custom processing graphs, and the ability to specify the match fields and actions of tables within the switch. For these reasons, the developers of netcache utilized P4 to compile their program onto a programmable switch. The implementation of Netcache on a switch effectively reduced the load imbalances and improved the throughput. Overall, NetCache improved the throughput by 3.6×, 6.5×, and 10× over the NoCache implementation under Zipf 0.9, 0.95 and 0.99, respectively. The implementation of NetCache is significant to our hypothesis, because Netcache could essentially be used as a function for a microservice. The only requirement for a microservice is that it performs some form of set function, and one microservice could simply be a form of Netcache on the switch. This Netcache could be extended to fit within the microservice architecture and use API calls for communicate with other microservices. The Netcache paper does not analyze the Netcache implementation in the context of microservices, and still requires adjustements to meet the solution criteria that we proposed. The implementation of multiple Netcaches, or a Netcache

with another API would also better satisfy the description of a microservice implementation on a switch.

2.3 In-Switch Computing Architectures

There two primary routes for this in-switch computation: purely hardware description, or using P4 and a programmable switch implementation. The purely hardware design approach has the benefit of being much more capable in terms of hardware. We can design a switch that has the most ideal hardware which is required to support the various microservices that might be implemented on this switch. The programmable switch route has the benefit of being software defined. This would allow changes to be made to the switch without having to completely redesign the hardware. Considering how microservices are ideally easily updated and scalable, this may be preferable long term and more realistic route for hosting microservices on switches. The primary downside of this form of implementation is that the implementation will be heavily dependent on the hardware capabilities of the switch architecture. Programmable switch architecture targets for P4 are broken down into various pipe section that perform functions as information flows. These pipe have limited memory and processing capabilities. We describe the layout of both of these architectures as well as what implementation on a programmable switch will look like on them.

Figure 2.1 depicts our proposed high-level architecture of a hardware design approach to In-switch microservices. This design was informed by the previous in-switch architecture described by the authors of an in-switch computing and accelerated distributed reinforcement learning. The authors propose an in-switch accelerator that performs gradient aggregation for a machine learning model in the switch. The design was created with the intention of identifying microservice packets, and rerouting them to the processing unit portion of the switch. Incoming packets would enter through the physical transceiver of the switch, and then move through the Media Access Control (MAC). The packets are subsequently added to the ingress queues. These queue act as buffers and feed the packet into the Filtration and Arbitration Unit in the appropriate order. Here packet header information is processed and then fed into either the Processing Unit or Packet Processing Unit depending on the read packet information. The Processing Unit executes the microservice

application and the Packet Processing Unit contains the header parser, ethernet lookup tables, and IP lookup tables. Recognized API call packets will be filtered into the Processing Unit to perform the microservice API call within the switch, and unrecognized packets will be sent straight into the Packet Processing Unit. The Processing Unit performs the API call for the particular microservice that is being hosted within the switch and compiles the resulting data into a packet to be sent back to the host. The compiled packet will subsequently be sent back into the Filter and Arbitration Unit, which then reroutes the packet to the Packet Processing Unit. The header information is parsed and compared to the lookup tables for the appropriate destination. Packets are then forwarded accordingly to the corresponding TX Queue, MAC, and PHY Transceiver.

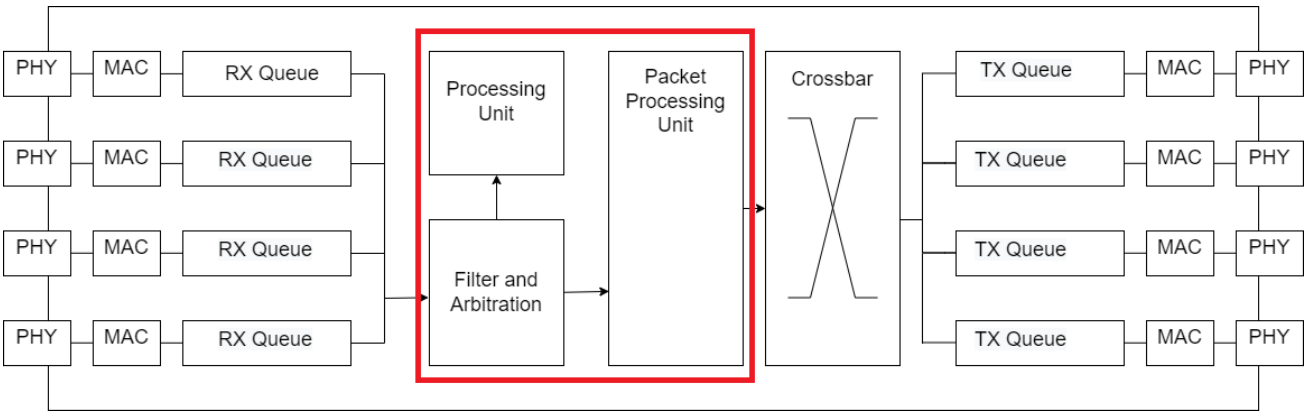

Figure 2.1: In-Switch API handling model.

The Processing Unit used in the switch would consist of one multi-core CPU. This CPU would have a small cache, memory, and program memory. There would be no kernel nor operating system, as assessed, in order to reduce the end-to-end system latency. One of the cores would be assigned as the 'control' core, which manages the other CPU's. This Processing Unit would repetitively run a program checking a software queue for a received packet. It would then appropriately place the packet information in memory for the other CPU cores to access and process. The program for running the respective microservice would already be in program memory and ready to execute the associated API call. The management CPU would take resulting data and recompile it

into packet format to be forwarded out. It would send this packet into a software output queue that would output into the Packet Processing Unit. This Architecture has the benefit of being able to explicitly defined processing unit to meet the hardware requirements of the hosted microservice. The primary issue with this architecture is that designing hardware around a specific microservice would take much more time to implement and update.

2.3.1 Programmable Switch Architecture

To understand how a microservice might be implemented on a programmable switch, we first analyze the target architectures for the P4 as well as its software capabilities. Current network switches that support P4, such as the Barefoot Tofino, utilize Protocol Independent Switch Architecture (PISA). PISA is the hardware architecture of the programmable switch being used, however, there can be specific architectures such as PSA, V1, and SimpleSwtich that can be programmed into this architecture.

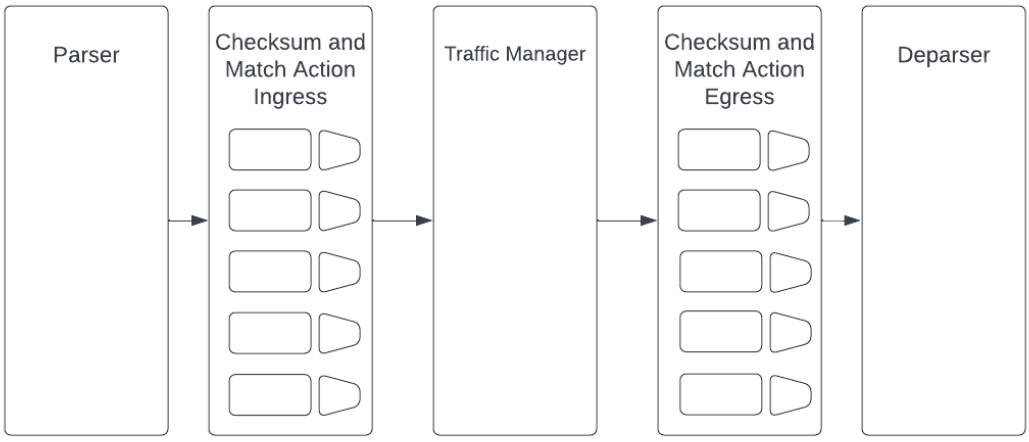

Figure 2.2: PISA high-level architecture [16].

The layout of the PISA architecture is depicted on a more abstract level in **Figure 2.2** and is elaborated further in **Figure 2.3**. The PISA Architecture is a pipeline that can be divided up into 5

main sections: The Parser, The Ingress Checksum and Match-Action Pipe(s), The Traffic Manager, The Egress Checksum and Match-Action Pipe(s), and the Deparser. The Parser, Traffic Manager, and Deparser are fixed function and specific to the hardware of the Switch. These parts cannot be changed by the programmer. The Parser simply extracts headers from the incoming packets, these packets are then sent to the next pipeline stages to be processed. The Ingress and Egress Checksum and Match-Action are the programmable sections of the network switch. The number of pipes is defined by the programmable and limited based on the switch hardware and the resources available. There are parallel memory and ALU components for each match action pipe. This allows for parallel match action execution, however, depending on the program there may be match or action dependencies. Match dependencies will result in a full unit of latency (Considering one match action as one unit of time), whereas action dependencies can have staggered execution and result in a half latency. This is because the match will occur prior to the action, so a staggered execution can occur. This is important to note in the context of this paper for optimizing microservice execution. Another important note is that the Ingress and Egress Match-Action have a shared memory. This allows for data to be easily accessible further down the pipeline, and supports continuous flow of data and execution.

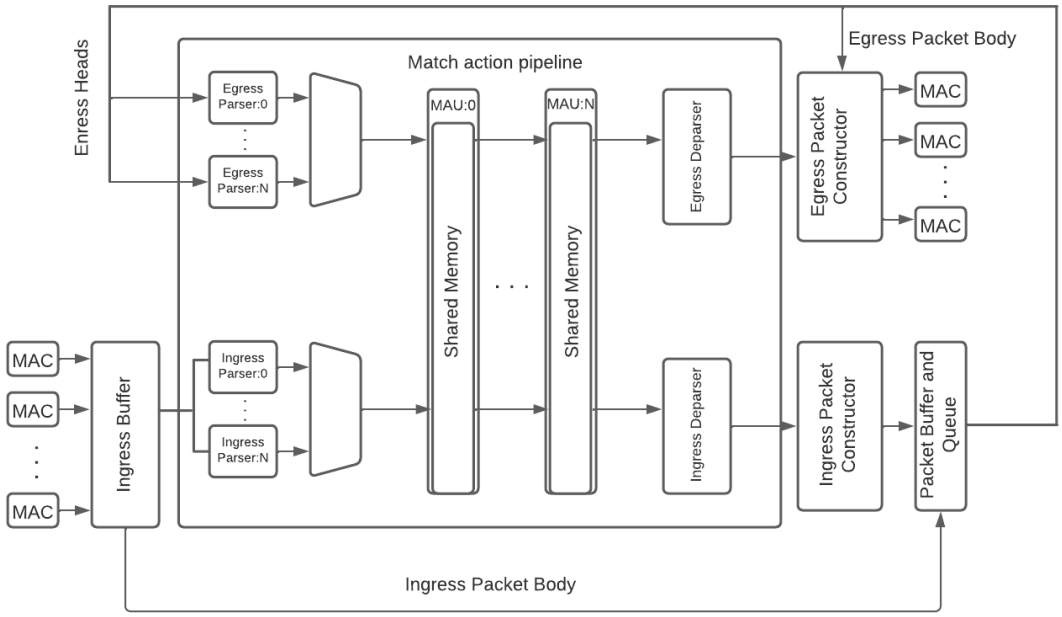

Figure 2.3: PISA detailed architecture [16].

The specific layout of PISA that describes all switch architectures can be observed in **Figure 2.3**. This figure depicts a number of MAC ports feeding packets into the Ingress buffer before proceeding to the same number of Parsers on the switch. These Parsers then direct traffic to the Ingress Match-Action stage, which executes the programmed logic prior to entering the Ingress Deparser. The Deparser send the processed information to the Ingress Packet Constructor which recreates a packet to output. The Headers are then sent to the Egress processing portion of the switch, where a similar process occurs. Notice how the memory between the Ingress and Egress is shared, so the information is essentially sent back to the same pipes to be executed. This allows for efficient use of hardware resources on the switch. This hardware architecture enables developers to easily define the specific switch architecture that they would like to implement on top. The primary difference between the switch architectures being implemented is simply the number of pipes for the Match-Action stages. Considering this, we are able to develop a theoretical switch architecture on top of this hardware. The P4 Language allows the developer to

easily define the number of pipes, actions, and a variety of other networking components. P4 programming is best described as C, but remove loops, pointers, malloc, and free. Programmers can define tables, actions, metadata, headers, and offers enough flexibility and resources to implement C-like programs. Looping can be resolved by using one of the rerouting lines within the switch, these line vary from switch to switch. Using P4, a developer can define a microservice within the switch and using the different pipeline stages do the Parsing, processing, and deparsing. Using programmable switches offers a software route to implementing an In-switch microservice. The primary drawback is that the microservice will be limited to two constraints: The P4 programming language, and the programmable switch architecture. As mentioned P4 is a limited version of C, so programmers will have to programmable the microservice to fit the P4 format. Depending on the programmable switch architecture being used, there will be certain hardware constraints placed onto the microservice.

3. ANALYSIS

The possible architectures for an In-switch microservice implementation have different advantages and disadvantages. The Hardware Architecture approach follows a more standard implementation of a switch with the exception of the added hardware elements. Our architecture model is based on the previous implementation of an In-switch computation model for Accelerated Distributed Reinforced Learning, whereby a processing unit is added to the current hardware of switches. This architecture had the advantage of being able to explicitly define supporting hardware and capable of utilize other multi-core processor models. The downside of this model is that it would be more complicated to update the codebase of this model when compared to Programmable Switches. We found that the Programmable Switch Architecture is preferable in the context of an in-switch implementation of microservices in the current space. While this architectural route has the downside of hardware and software limitations, it has the benefit of extensive documentation and support of other sources along side these ease of implementing existing microservices in P4. To analyze in-switch microservices, we would only need to analyze multiple services being performed within a switch. A switch capable of hosting multiple services would qualify as a microservice since a microservice is considered to host one or more services within itself.

For our analysis we observe data from the implementation of NetCache: Balancing Key-Value Stores with Fast In-Network Caching, which is written in P4 and implemented on a programmable switch and analyze it in the context of microservices. Netcache is essentially a simpler implementation of memcached that targets hot items in an attempt to solve load imbalance in other key-value store implementation. The only requirement for a microservice is that it performs some form of set function, and one microservice could simply be a form of Netcache on the switch. This Netcache could be extended to fit within the microservice architecture and use API calls for communicate with other microservices. If two Netcaches were to be hosted on a switch, this would

qualify as a microservice. NetCache has an average latency of 11- 12 microseconds per query when implemented on one 6.5Tbps Barefoot Tofino switch [13]. NetCache is also able to run on the Tofino at line rate at 2.24 BQPS. Netcache functions close to the latency and line rate of this programmable switch. This means that the NetCache application is contained within the switch node in the context of a network model. The PISA model for programmable switches, has resources that adjust according to the switch model. A theoretical model can then be made of two NetCaches on such a switch. This is feasible, considering how the memory and processing power can be defined for each pipe of the architecture. These resources are in parallel, meaning a separate NetCache could be hosted alongside another. This would define a microservice of two NetCaches running separately with the latency value of one NetCache. With this in mind, we have formed a theoretical model for an in-switch microservice with supported theoretical latency value. This implementation of multiple NetCache's on a SDN switch fits the design criteria we set. The header of the packet is identified as being NetCache in the Ingress Parser portion of the PISA pipeline. These packets are subsequently forwarded to the next pipeline stages to be processed. Non-identified packets pass straight through the other pipes without being processed. The standard NetCache Implementation runs at the line rate of the switch, meaning it maintains transparent processing and generation of packets within the switch. Because of the localization of NetCache within the switch, a node can be taken out of the graph of a microservice application. What would normally be a separate node is now contained within the network latency of a switch. This effectively reduced the communication overhead while also eliminating the end-to-end system latency that might otherwise be generated from system traversal. User transparency and previous benefits of microservice applications over monolithic applications are also maintained via P4. Because NetCache is software defined using P4, it is capable of being changed, scaled, and tested while providing an individual service.

4. CONCLUSION

In this paper we presented a set of criteria that addresses the drawbacks of microservices and hypothesized a potential in-switch model to low latency microservices. The in-switch model would reduce end-to-end service request latency by reducing the required number of network jumps and latency incurred by the server. We then defined the set of design criteria that this in-switch model would have to meet in order to address the drawbacks of the current microservice architecture. This criteria was centered around resolving the latencies incurred from the OS and Network communication done in a microservice architecture. We then gave a hardware and SDN model based on previous research. In our analysis, we found that multiple NetCache's are theoretically capable of being implemented in a programmable switch as a microservice and meet the defined design criteria. This implementation of NetCache as a low latency microservice in a switch, reduces the end-to-end and network latencies by being hosted on a switch. This would be a solution the drawbacks of a microservice as we defined and increase performance of microservices.

REFERENCES

[1] J. Thönes, "Microservices," *IEEE Software*, 2015.

[2] H. C. M. V. L. S. R. C. S. G. Mario Villamizar, Oscar Garces, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *2015 10th Computing Colombian Conference (10CCC)*, 2015.

[3] T. J. Yuqiong Sun, Susanta Nanda, "Security-as-a-service for microservices-based cloud applications," *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[4] W. Caban, *Architecting and operating OpenShift clusters OpenShift for infrastructure and Operations Teams*. Apress, 2019.

[5] U. Z. Amine El Malki, "Guiding architectural decision making on service mesh based microservice architectures," *Software Architecture*, pp. 3–19, 2019.

[6] A. H. B. Tetiana Yarygina, "Overcoming security challenges in microservice architectures," *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018.

[7] Y. Y. D. C. A. S. J. H. Youjie Li, Iou-Jen Liu, "Accelerating distributed reinforcement learning with in-switch computing," *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[8] G. G. Pat Bosshart, Dan Daly, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2019.

[9] B. P. C. K. N. F. Muhammad Shahbaz, Sean Choi, "Pisces: A programmable, protocol-independent software switch," *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[10] G. B. B. B. F. D. S. P. E. T. Samar Abdi, Samar Abdi, "Pfpsim: A programmable forwarding plane simulator," *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2016.

[11] P. K. C. F. Y. Z. C. Z. Jiasong Bai, Jun Bi, "Ns4: Enabling programmable data plane simulation," *Proceedings of the Symposium on SDN Research*, 2018.

[12] A. S. Michael D. Wong, Aatish Kishan Varma, "Testing compilers for programmable switches through switch hardware simulation," *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020.

[13] H. Z. R. S. J. L. N. F. C. K. I. S. Xin Jin, Xiaozhou Li, "Netcache: Balancing key-value stores with fast in-network caching," *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[14] C. D. Yu Gan, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, 2018.

[15] H. Y. W. Q. Guisheng Fan, Liang Chen, "Multi-objective optimization of container-based microservices scheduling in edge computing," *Computer Science and Information Systems*, 2021.

[16] K. O. Ismail Butun, Yusuf Kursat Tuncel, "Application layer packet processing using pisa switches," *Sensors*, 2021.

APPENDIX A: MICROSERVICE PARAMETER DESCRIPTION

Parameters	Descriptions
$G_a = \langle ms\_set, ms\_relation \rangle$	microservice application
m	the number of microservices in the application
ms_i	microservice with id. i
$(ms_i, ms_k) \in ms\_relation$	dependency link from microservice ms_i to ms_k
$calc\_need_i$	computing resources required by one request for microservice ms_i
str_need_i	storage resources required by one request for microservice ms_i
max_link_i	the maximum number of requests for one instance of microservice ms_i
pre_set_i	preceeding set of microservices of microservice ms_i
$direct\_reqst_i$	the number of direct requests for microservice ms_i from users
$link(ms_i, ms_k)$	the number of indirect requests from microservice ms_i to ms_k
$link_i$	the total number of requests for microservice ms_i
$trans(ms_i, ms_k)$	size of data transmitted between microservice ms_k and ms_i
$scale_i$	the number of instances of microservice ms_i
$G_e = \langle node\_set, link\_set \rangle$	edge computing environment
n	the number of edge nodes in the cluster
$node_j$	edge node with id. j
$calc_j$	computing resource capacity of edge node $node_j$
str_j	storage resource capacity of edge node $node_j$
$fail_j$	failure rate of edge node $node_j$
$l_{i,j}$	communication link between edge node $node_i$ and $node_j$
$b_{i,j}$	bandwidth of link $l_{i,j}$
$d_{i,j}$	network distance of link $l_{i,j}$

A.1: Microservice Parameter Descriptions [15]