Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Multitenancy and Resource Management for In-Network Caching

Author: Bart de Haan (2656607)

1st supervisor: Lin Wang
2nd reader: George Karlos

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

June 15, 2021

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

A recent trend in software-defined networking has seen the creation of P4, a domain-specific programming language for the programmable data plane (PDP). This development offers new possibilities for in-network computing (INC) and it has great potential for the cloud and data centers. However the current PDP has some limitations. It does not support multitenancy and there is a lack of good resource abstractions. As a consequence, these limitations impact the adoption of INC as a new platform. Additionally, most recent INC applications also share the same limitations.

In this thesis we focus on in-network caching and aim to address these challenges. We design and implement MADINC, a Multitenant And Distributed In-Network Cache. We specifically store key-value items. This new key-value cache supports multiple users and provides an abstraction for the cache resources. We enable multitenancy by extending existing key-values with a secondary key. This secondary key is used as a user ID and allows the cache to distinguish between items of different users. Next, we create a resource abstraction for the cache memory. We partition the cache space into fixed sized chunks. These chunks can be handed out to users and they simplify resource management, support multiple users, and enable our cache to scale to multiple switches. We implement a prototype of our system and test it on Mininet. Our results showed that our cache performs and scales well. Both with multiple users and with multiple programmable switches. Moreover the cache works well for skewed workloads but packet loss significantly reduces the cache performance.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

Over the last few decades the amount of data generated, processed and stored is growing at a rapid pace [1]. These large volumes of data are often called big data. Big data has high processing and storage requirements. The cloud is a popular platform for providing these processing and storage services to users [2]. In the last decade we have seen platforms such as Microsoft Azure [3], Amazon AWS [4], and Google cloud [5] grow and become successful businesses.

The performance of storage systems is an important factor for handling big data. Big data is often unstructured [6]. Picking an effective storage system that stores big data must therefore store unstructured data. Relational databases do not fit this type of data. They store structured data. NoSQL databases are designed for storing unstructured data [7]. They are often used in big data for storing the unstructured data.

We in particular are focusing on one type of NoSQL database, the key-value store. A key-value store is a simple and effective databases that stores values. Each value is linked to a unique key and the key is used in order to retrieve the value. An example of such a database is Dynamo [8].

The performance of in-memory key-value stores is limited by the hardware that it runs on. There are a couple of potential bottlenecks which could limit the performance. The network can become a bottleneck if the hardware cannot handle the received traffic. The file system and memory can also be a bottleneck. The speed at which memory and storage disks retrieve data cannot increase unless we use better hardware. Unless we can improve the hardware on which in-memory key-value stores run, increasing the performance of in-memory key-value stores is a challenge.

In-network caching is a solution to increase the performance of key-value stores. A cache stores a small amount of items and it is placed on-path between a sender and receiver. For

example IncBricks [9] created a middlebox solution and NetCache [10] used programmable switches. They both showed that in-network caches improved throughput and latency. A cache benefits from the shorter network route which reduces latency. Additionally, switches and middleboxes can process packets at a faster rate than commodity servers which improves throughput and latency. They also showed an in-network cache acts as a load balancer for the key-value stores. In-network caches reduced the peak load on in-memory key-value stores which increased the overall throughput.

In-network caching can be achieved by software-defined networking (SDN) and the programmable data plane (PDP). SDN is a networking paradigm that separates the data plane and control plane. The data plane, also known as the forwarding plane, handles packet processing and forwarding. The control plane on the other hand is responsible for routing and decision making. SDN separates them and removes the control plane from switches and routers. The control plane is centralized and handles the control logic for multiple network devices.

Traditionally the data plane in SDN consists of simple forwarding devices. This limits the possibilities on what can be done on networks. For example it is not possible to implement a cache on switches. However the introduction of Reconfigurable Match Tables (RMT) model [11] introduces new possibilities. It specifies a hardware model that allows developers to target programmable network devices. The most promising language that targets the RMT model is P4 [12]. It is a domain specific language that expresses a series of match-action stages to process packets. The recent developments of the programmable data plane allows us to implement applications such as a key-value cache on a programmable switch.

We noticed two limitations of recent in-network caches. Firstly, they lack multitenancy support. This comes from limitations in the hardware. For instance programmable switches cannot run multiple applications at the same time [13]. Thus we cannot run multiple caches on the same devices. The only option is to run a single application that serves multiple users [14].

The second limitation is good resource abstractions. Abstractions hide complexity. For example on commodity hardware, the operating system virtualizes the memory which is an abstraction of the physical memory. This virtualization provides resource isolation and it makes resource management simpler. The PDP on the other hand, does not support virtualization [15]. It therefore does not gain the benefits of virtualization and makes resource management harder.

We want to research if we can find solutions to the limitations and challenges we described of in-network caching. We are going to focus on two topics: multitenancy and resource management. We have formulated two research questions, one for each topic. They are as following:

- How to enable multitenancy for in-network caching?
- How to manage the resources of multiple in-network caches?

We applied two new concepts which enables multitenancy and helps us manage the cache. Firstly, we introduced a secondary key to the key-values. This secondary key represents a user ID and is used for identifying cache items of different users. This solution allows the cache to be shared by multiple users. Secondly, we partition the cache space into equal sized blocks. We do this for every switch cache and call each block a cache page. We assign each page with a unique ID which specifies two things: a switch cache and the memory range on that cache. This second concept allows us to manage multiple caches for multiple users by managing the cache pages of the users.

In this thesis we make the following contributions:

- A design for a multitenant and distributed in-network key-value cache
- A prototype implementation of our design
- Experimental results of our prototype

This thesis is structured as follows. Section 2 provides background information on topics that are relevant to this thesis. Section 3 explains the design of our distributed and multitenant switch cache. Section 4 provides insight on our implementation. In Section 5 we evaluate our system and discuss the results of our experiments. Section 6 discusses our work and some of our findings. We also give our thoughts on how to improve our cache for future works. Finally, we conclude our work in Section 7.

2

Background

In this section we provide the background information needed to understand future sections. We start by explaining SDN. SDN plays a large role in our caching application and we explain what it is, and how it is used. Next we will explain the programmable data plane and the P4 language [12]. Lastly we will explain NetCache [10], a in-network key-value caching application implemented in P4. Our own multitenant and distributed cache is based on their design and we want to explain how it works, what it achieves and lastly what its shortcomings are.

2.1 Software-defined networking

SDN is a network paradigm that separates the control plane from the data plane. The control plane manages the forwarding rules and the data plane forwards packets on the network. Traditionally, networking devices had these two planes combined together [16]. This resulted in resilient and and decentralized networks. This approach has worked well so far but current networks are complex and hard to manage [17]. SDN is a solution which attempts to improve managing networks.

SDN separates different planes which are traditionally tightly coupled. Figure 2.1 shows an overview of the different planes and how they are connected. The data plane is at the bottom and contains devices such as switches. The control logic has been removed from the data plane and it is instead centralized by a controller in the control plane. The planes can interact with each other using an API which is called the southbound API.

The most prominent southbound API is OpenFlow [18]. A controller uses OpenFlow to program flow tables of devices in the data plane. A flow table contains flow entries which are forwarding rules for network traffic. This API allows a controller to manage the

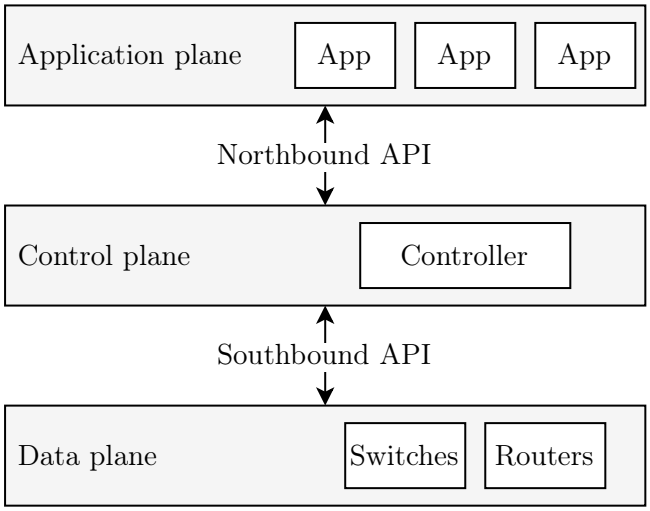


Figure 2.1: Overview of the three planes in software-defined networking.

traffic of the data plane. However, OpenFlow has its limits. Specifying packet processing behavior is rather difficult because OpenFlow focuses mainly on traffic flows. Therefore, it cannot be used for implementing a cache in the data plane.

SDN adds flexibility to the network via software control. Controllers can be programmed to manage the data plane and they can use a southbound APIs such as OpenFlow, to communicate to the data plane. Such a centralized and programmable controller simplifies network management. The controller is sometimes also seen as a network OS [19] because it manages the network. This view of SDN introduces a third plane: the application plane. This abstraction allows developers to write applications which run on the network. The controller therefore acts like an OS. This approach is adopted in controllers such as ONIX [20] and ONOS [21]. The interface between the application plane and the control plane is called a northbound API.

2.2 Programmable data plane

Traditionally, in SDN the data plane consisted of simple forwarding devices. It was not possible to specify packet processing behavior. The programmable data plane (PDP) allows developers to implement their own packet handling. The PDP provides new possibilities. Researchers can implement new networking protocols and we have seen in-network applications like caching [9] [10] [22] [23], network consensus [24], and data aggregation [25] [26] [27].

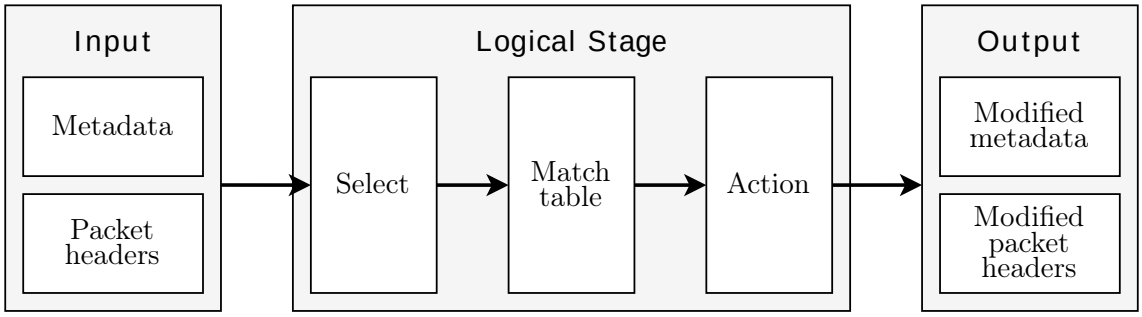


Figure 2.2: Reconfigure match table model with a single logical stage.

The PDP has its own unique set of requirements and limitations. Networks are optimized for high throughput and low latency. Applications running on the PDP must therefore also be performant so they do not slow down the network. As such the PDP has a couple of limitations to meet the performance requirements. There is a limited memory, set of instructions, and operations per packet [26].

The most prominent language for the PDP is P4 [12]. It is a domain specific language and is designed to target the Reconfigurable Match Tables (RMT) model [11]. We will discuss both topics in the next subsections.

2.2.1 Reconfigurable match tables model

RMT is a model designed for handling packets in the data plane [11]. It defines a match-action stage which allows actions based on header fields. A simplified view of the RMT model is shown in Figure 2.2. The logical stage is the important component and it is based on the match-action principle. This principle first matches header fields against a table that contain match entries. The entries contain actions which update header fields. For example we could match on the destination IP address from the IP header and then perform actions for routing the packet which could update its header fields.

The first step in a logical stage is selecting data from the input. This can be header fields or metadata. RMT allows for arbitrary field selection. It is then matched to entries in a reconfigurable and arbitrarily sized table. The last step in a logical stage is performing actions which modifies the input. Actions can add or remove headers, and update any header or metadata fields.

Multiple of these stages can be chained together to form a packet pipeline. RMT allows for an arbitrary number of stages to be chained together. We show in Figure 2.3 a simplified overview of a RMT packet pipeline. Packets enter this pipeline on the left where they are just a sequence of bytes. A packet is first parsed to extract the header fields. These fields

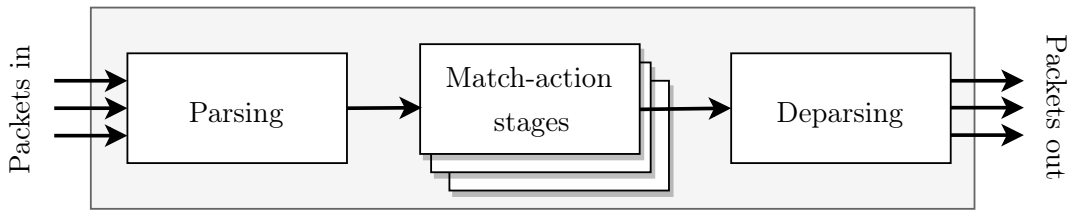


Figure 2.3: Reconfigurable match tables pipeline

are coupled with metadata and inserted into the the match-action pipeline. At the end of the pipeline the packet is reassembled and pushed out of the pipeline. The packet then leaves the device on a network port.

2.2.2 P4 language

The P4 language [12] is a domain specific language for the data plane. In P4 developers express packet processing behavior that targets the RMT model. P4 has three core principles:

- Field reconfigurable
- Protocol independence
- Target independence

Field reconfigurable means the control plane is able to change the way the data plane processes packets while it is deployed. P4 has two options for field reconfiguration. Firstly the control plane can replace a running P4 application with a new one. This action loses the state of the running application. Secondly, the control plane can reconfigure match-action tables of running programs.

The second principle, protocol independence, indicates that P4 does not assume any network protocols. This is great for a number of reasons. It allows developers to create and experiment with new protocols, and existing protocols can be updated to fix or add new functionality.

The last core principle is target independence. It is aimed at avoiding vendor lock-in where applications can only run at vendor specific hardware. With target independence P4 programs can be deployed to multiple architectures. For example we can deploy to ASIC's, FPGA's or even commodity servers. The only requirement is that the architecture supports the RMT model.

A P4 program defines a parser, a deparser, and control blocks. This follows the RMT model and creates a pipeline similar to Figure 2.3. We must first define a parser. In P4 we create a state machine to parse the packet headers. Each state in the state machine can extract headers from the packet and make a transition. The transition is either to a different state or the final state. Control blocks define match-action stages such as the one displayed by Figure 2.2. In a control block we can define a table and actions. The table allows P4 to match to header fields and actions can modify the header fields or metadata. The last thing in P4 we define is a deparser. A deparser reassembles the packet from the modified headers.

2.3 NetCache

NetCache [10] is an in-network caching application. Caching applications are designed to store a small amount of items. For example on CPUs there are multiple cache layers. It takes a CPU less cycles to access data in the cache than data in RAM. An in-network cache has the same idea but instead it operates on the network.

It is designed to store key-values on programmable switches. It makes use of SDN and the PDP in order to create this caching application. One of the limitations of NetCache is that it does not support multiple users.

Figure 2.4 shows a simple architecture of NetCache. It is designed for a ToR switch but we show a more simplified view of the architecture. There are clients which use the key-value stores. There are key-value stores which host the data. These components do not directly make up the NetCache application but they are two necessary parts of a key-value store. The switch data plane runs a P4 application which stores key-value items and processes key-value queries. This switch is the network cache and it is controlled by the NetCache controller.

NetCache uses multiple match-action stages but the first stage is the most important one. In the first stage NetCache matches on the key field. If the key matches to an entry then the key is stored in the cache alongside its value. On a cache hit the value will be fetched from memory. If the client has requested this key-value item then NetCache can immediately return this value.

All the components in Figure 2.4 communicate with each other using a custom application layer protocol. This is the NetCache protocol and Figure 2.5 shows its fields. There are four different fields. First there is the *OP* field and is used for specifying a cache operation. For instance a load, store or delete operation. The second field *SEQ*, is used as a sequence

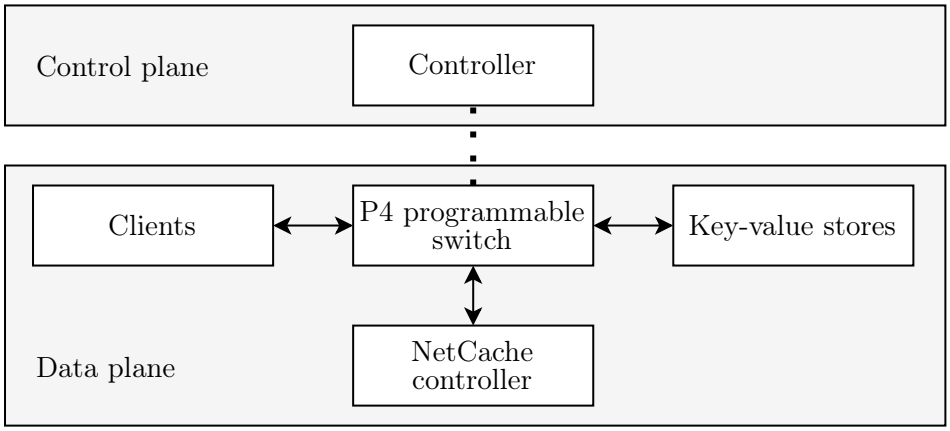


Figure 2.4: Overview of the NetCache architecture

OP	SEQ	KEY	VALUE

Figure 2.5: NetCache application layer protocol

number which is useful for sending multiple messages. However the authors of NetCache did not use this field and have also not implemented it on their git repository [28]. They are *KEY* and *VALUE* which store a key and a value respectively. These last two fields are used for transferring key-value items.

As mentioned earlier a controller manages the content of the cache. But it is the data plane that tracks hot key-value pairs. A key-value pair becomes hot if it is requested often over a short time span. Once an item becomes hot the P4 application will send a heavy-hit report to the controller. The controller will then try to cache the hot key-value item.

3

Design

In this section we explain the design of our network cache. We want to reiterate the research questions which we described in the introduction. These questions are:

- How to enable multitenancy for in-network caching?
- How to manage the resources of multiple in-network caches?

These two research questions specify two main goals which we must address. Firstly how to share the switch cache and secondly how we extend our cache to use multiple switches. Implementing these two goals should provide a multi-user and multi-switch caching system. We will start this section with an architectural overview of our system which shows every component. We then zoom in on each component and explain its requirements and design.

3.1 System Overview

The core functionality of our system is to provide a multitenant in-network key-value cache. We specifically designed the cache for programmable switches. This cache supports multiple users and scales to multiple switches. We designed our cache using the principles of SDN. We have components running in the data plane, control plane, and application plane. We call our cache: MADINC (Multitenant And Distributed In-Network Cache). Figure 3.1 shows an overview of our caching application with all the relevant components.

In the center of Figure 3.1, we show two programmable switches. They are part of the data plane and forward traffic just like normal switches. But unlike normal switches, they also host our key-value cache. The switch processes every packet and looks for a specific packet header which is used for accessing and operating the cache. In Section 3.2 we explain our application protocol.

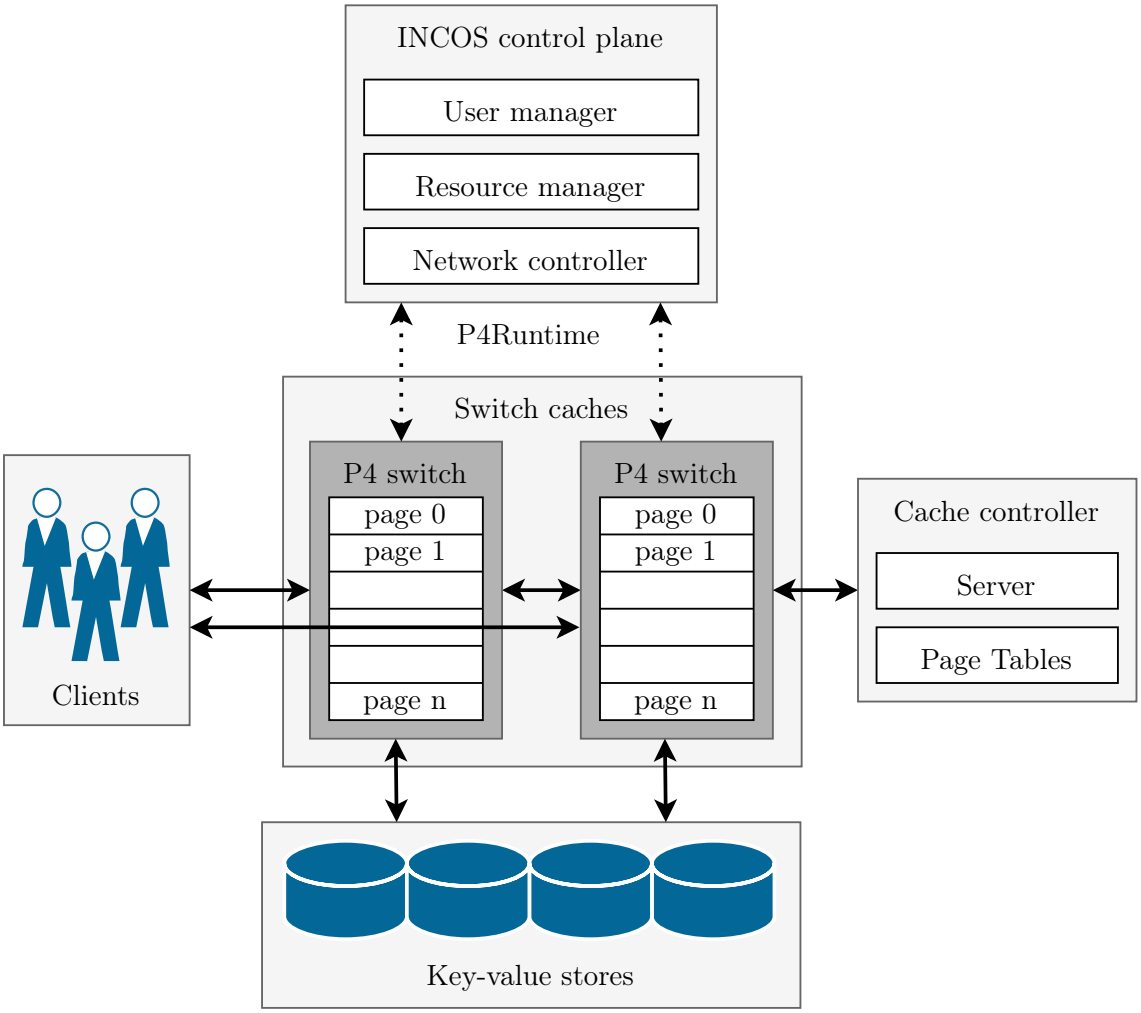


Figure 3.1: MADINC design architecture

One of the most important cache functions is the lookup. When the switch cache receives a read or write query, it starts to look for the data in the cache. The concept of this lookup is similar to that of CPU caches of commodity hardware. The lookup produces either a cache hit or a cache miss. On a cache hit we have found the data and can return it to the sender. Otherwise we have a cache miss and forward the query to a key-value store. In Section 3.3 we will provide more details about our switch cache design and its functionality.

The switch cache does not manage itself. It only performs basic cache operations such as read and write, and heavy-hitter detection. It does not manage what is inside the cache. That task is performed by the cache controller. This component is shown on the right side of Figure 3.1. The cache controller is capable of managing the cache of multiple users. It receives heavy-hit reports from the switch cache and updates the cache based on the

reports. In Section 3.4 we will explain the design of the cache controller.

Our control plane controller, INCOS is designed as a network operating system [29]. It performs the traditional tasks of a control plane such as creating routing rules and installing them onto network devices. Additionally we extend it to also manage users and resources. These resources are in our case memory used for caching key-values. These resources are given to users and are used by the cache controller for managing the memory. Besides managing and controlling the network it also updates the cache table on the switch caches. But all the logic for updating, inserting and removing table entries are handled by the cache controller. In Section 3.5 we explain our design of the control plane.

The final two components in Figure 3.1 are the clients and the key-value stores. They are not part of the key-value cache. Instead they act as users of the cache. The clients generate queries which can be handled on a switch cache. The key-value store also handles queries and some of the key-value items are placed in the cache for faster data access.

3.2 MADINC protocol

We have created our own application layer protocol. We needed our own protocol in order to support communication between the switch cache, the cache controller, and the control plane. We refer to our protocol as the MADINC protocol and we have drawn inspiration from the NetCache protocol which we described in Section 2.3.

Our protocol has three main tasks. Firstly, it needs to be able to communicate basic key-value store operations such as read and write. Clients can use these operations to query key-values from our cache or a key-value store. Secondly, the protocol needs functions for moderating the cache. We need to insert, evict, and replace items from the cache. The last task is to support resource allocation. This will be useful for communicating resource allocation from the control plane to our cache controller.

Besides the core functionality we also need to add support for multiple users. A user must only be able to query the contents of its own cache. If two users have the exact same key in the cache then we need to identify exactly to which user that key-value item belongs to.

Figure 3.2: MADINC application layer protocol.

The MADINC protocol has four fields as shown in Figure 3.2. We kept the *OP*, *KEY*, and *VALUE* fields from the NetCache protocol but replaced the sequence field for a user ID. The *KEY* and *VALUE* fields are needed for transporting a key-value item. Both fields have a fixed sizes which we set to 16 and 128 bytes respectively. We support multiple users with a user ID. Each unique user gets their own ID. With a unique ID we can identify cache items and messages. This solves the problem where two users can have the exact same key but with different values. With a user ID we can separate the key space of multiple users.

The final field is the *OP* field. Here we specify different message types for our cache. Table 3.1 shows the different opcodes with some additional info about each opcode. There are a few different message categories. The first four opcodes are for read and write operations. They are used by a client for reading or writing key-values from the cache and key-value stores. A *HOT_READ_REPORT* is used for sending heavy-hit reports from a switch cache to the cache controller. We will go into more details on heavy-hit reports in Section 3.3.3. The next five opcodes: *CONTROL_UPDATE*, *CONTROL_INSERT*, *CONTROL_EVICT*, *CONTROL_REPLACE*, and *CONTROL_REPLY* are used for controlling the content of the cache. These opcodes will be explained in Section 3.4. The final two opcodes: *CONTROL_ALLOC* and *CONTROL_FREE* are used for assigning memory chunks to users. We will explain this in Section 3.5.

3.3 Switch cache

The switch cache needs to be shared among multiple users. We are using P4 to develop our application so we need to share the hardware of a P4 switch. Sharing a P4 switch is difficult because only a single application runs at all time. There is some research that tries to work around this limitation. For example HyPer4 [30] and HyperVDP [31] created a P4 virtualization application that can host match-action tables of other applications. And P4Visor [32] uses code merging which merges multiple P4 programs into a single application. These solutions work but they have some trade-offs. They offer multitenancy at the cost of performance, memory and reconfigurability.

This is problematic for a cache because we do care about all these drawbacks. Our switch cache needs to perform well to get a better throughput and latency than servers. There is no point in using a switch cache if it performs worse than a key-value store. We also care about memory usage because there is only a limited amount available on switches. Wasting memory will reduce the capacity of the key-value cache. Lastly we care about

Opcode	Info
READ_REQUEST	Request a key-value item.
READ_REPLY	Reply message for READ_REQUEST.
WRITE_REQUEST	Request to write a key-value item.
WRITE_REPLY	Reply message for WRITE_REQUEST.
HOT_READ_REQUEST	Heavy-hitter report of a switch cache.
CONTROL_UPDATE	Request from the cache controller for updating a value in cache.
CONTROL_INSERT	Control message for inserting a key-value item into the cache.
CONTROL_EVICT	Control message for evicting a key-value item from the cache.
CONTROL_REPLACE	Control message to replace an existing key-value item with a new key-value item.
CONTROL_REPLY	Reply message for CONTROL_INSERT, CONTROL_EVICT, and CONTROL_REPLY
CONTROL_ALLOC	Control message for the cache controller for updating the page table with a new page allocation.
CONTROL_FREE	Control message for the cache controller for freeing a page from a page table.

Table 3.1: Opcode table for the MADINC protocol.

reconfigurability because the cache needs to be updated frequently with new key-value items.

We did not use virtualization or code merging to create a multi-tenant cache. Instead we designed a P4 application with multi-user support builtin. This way only a single application runs on a switch. There are two important problems with this approach. First, how can we distinguish memory of different users? And secondly, how can we distinguish messages of different users? We solved both problems with a user ID with which we can distinguish messages and memory of different users.

We will discuss in the following subsections the design of the P4 switch and explain the inner workings.

3.3.1 Packet processing pipeline

In Section 2.2 we explained that a P4 application consists of multiple stages. Each stage has the ability to transform or do some processing on the packet. Figure 3.3 shows our switch pipeline with multiple stages. On the left packets flow into our pipeline and on the right packets flow out of our pipeline.

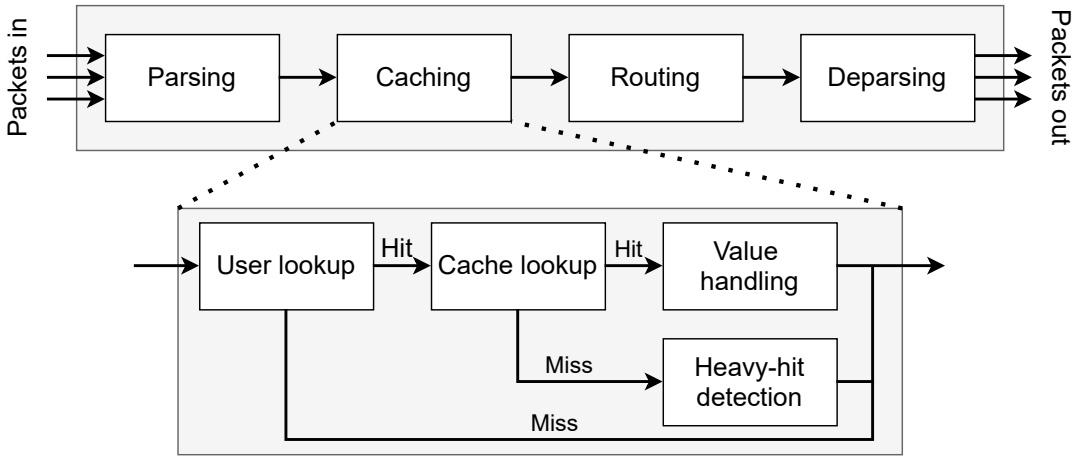


Figure 3.3: MADINC switch cache pipeline

After a packet flows into our pipeline the first stage we encounter is the parsing stage. Here we extract header information from the packet. In this stage we parse Ethernet, and IP headers. Either UDP or TCP headers, and finally our own MADINC headers. The first four we mentioned are all default network protocols which are needed for communication between different devices. The last one, MADINC protocol, is needed for using and controlling the switch cache.

With the extracted header information we continue to the caching stage. This stage handles all our caching functionality. In Subsection 3.3.2 we will go in-depth on this specific stage. After the caching stage we execute the routing or forwarding stage. Forwarding is a core task of any switch. Therefore we must do some processing in order to forward the packet towards its destination. The most important thing this stage does is providing an egress port on which the packet will leave the switch.

The final stage in our pipeline is deparsing. Here we rebuild the packet with updated headers and push the packet out of the pipeline and onto a network port. At this point the packet has left the switch and this also marks the end of our switching pipeline.

3.3.2 Cache lookup

The basic operation of the cache is a lookup. This checks if a user, and a key is present in the cache. Then depending on the request we can have range of operations which we can apply on the value. For example we can return or update the value. Figure 3.3 showed an overview of our switch pipeline. It also shows an in-depth overview of the caching stage.

The caching stage can actually be viewed as its own pipeline. We only enter this pipeline if our MADINC header is detected on a packet. If it is present then we do a user lookup with the user-id. This step verifies that a user has allocated memory on the switch. If a user does not have memory on the switch then we know that any cache lookup will fail. We want to avoid doing any unnecessary cache lookups. Therefore by checking the user early we can exit the caching pipeline early.

If the user lookup results in a hit we proceed to the actual cache lookup. We use a user-id and a key to find a value. There are two possible options: either a hit or a miss. If the lookup failed then we check if the requested key-value should be cached. We do some heavy-hitter analysis to determine if the item should be cached. We are going to discuss this topic in more detail in Subsection 3.3.3.

If the cache lookup results in a hit then we have found a value. It now depends on the opcode in the MADINC protocol header what we are going to do with the value. For example if the opcode was READ_REPLY, then we send a reply to the sender with the value.

3.3.3 Heavy-hitter detection

In case a user has memory on a switch cache but a cache lookup failed, we do heavy-hit detection. We attempt to detect if a key-value is frequently requested such that caching it will result in a performance gain or reduce the load on key-value stores. The switch cache is a good place to do heavy-hit detection since it is on the data path where it can evaluate the traffic. Our heavy-hit detection does not manage the contents of the cache. It only detects hot items and notifies the cache controller about these items. It is the cache controller which manages the cache. More on the cache controller in Section 3.4.

Our approach differs from CPU caches where every piece of data fetched from memory is automatically placed into the CPU cache. We do not automatically cache everything because notifying the cache controller about every cache miss can result in a lot of network traffic. We fear that this hurts the overall performance of the cache. Instead we do heavy-hit detection using a count-min sketch [33] and a bloom filter.

We use a count-min sketch for detecting frequent items. This is a space efficient method. Moreover Liu et al. [34] have shown that count-min sketch can also be implemented in P4 which means it works on a P4 switch.

After the count-min sketch has identified a hot item we apply a bloom filter. A bloom filter can identify membership [35]. This is useful for detecting key-values only once. After a heavy-hit report is generated the key-value item is added to the bloom filter and we do not generate a new heavy-hit report every time the count-min sketch detects the same hot item.

3.4 Cache controller

The cache controller is responsible for managing the contents of a switch cache. It needs to be able to manage memory of multiple switches. Furthermore, it needs to be able to handle multiple users. Figure 3.1 showed the cache controller with two main components: a server and a page table. We will first discus in Subsection 3.4.1, how we manage the memory of multiple switches and users. Then in Subsection 3.4.2 we will explain how the cache controller handles heavy-hit reports and inserts new items into the cache.

3.4.1 Cache memory management

The memory in a switch can be viewed as one big array. In this array we store fixed sized values. During a cache lookup the switch uses a user-id and key pair to access the value in memory. By using a user ID in the lookup the cache guarantees to only access memory of a specific user. However, the underlying memory is just one array which is shared among multiple users. We need a way to partition the memory such that we can assign partitions to users. This enables a multi-tenant cache and ensures that the memory is not shared by multiple users.

One problem with space partitioning is memory fragmentation. A fragmented memory space is hard to manage. We want to have a simple approach which limits the amount of fragmentation. We avoided using index range because it can become hard to keep track which memory range is in use. We decided to partition the memory space into fixed sized chunks. We got the idea from memory pages in computers. A memory page is a fixed size block of memory [36]. For example the x86-64 architecture uses 4kB memory pages. We think this approach is simple for managing memory and reduces the amount of fragmentation. For instance for allocating memory to a user, we only need to find an

available page and assign it to them. De-allocation is the opposite, where we only reclaim blocks.

Now that the memory space is partitioned in fixed size chunks we need a data structure that store the memory of a user. Operating systems manage memory pages with page tables where each process gets its own page table. The page table ensures that processes can only access their own memory.

Our approach for a page table is different but we do store pages and we also isolate the page tables of different users. From our page table we need to find a cache page. This cache page will be used for storing new items into the cache. In our system the table is a circular linked list where the cache pages are used in rotation. See Figure 3.4 for an example of our circular linked list. Each node in the list represents a switch cache and we store the cache pages on a least recently used (LRU) queue.

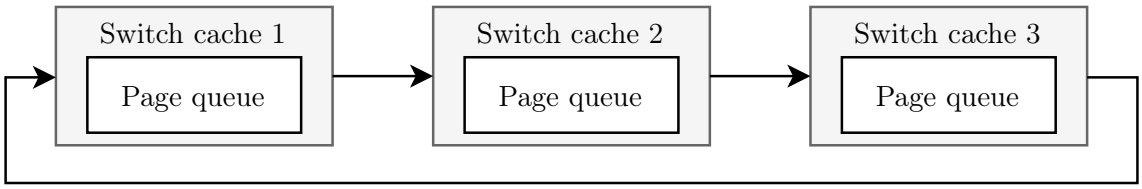


Figure 3.4: Circular linked list of cache pages. This data structure is used for managing the cache space of a user.

As mentioned the pages are used in cycles. There are two steps for finding a page which we will be used for inserting new values. We first select a switch. This selection is round-robin because we cycle through our circular linked list. Then from the LRU queue we take a page. Every time we need a new page we first select the next switch cache in the linked list and then take the least recently used page from that switch. This approach evenly selects the switch caches but if the memory of different switch caches are imbalanced then certain pages will be used more often. That is because of the smaller LRU page queue. For example we have memory on two switches. The first switch cache has two pages and the second cache has one page. Every time we select memory from the second switch we will always use the same page but the memory of the first cache will have a longer use cycle because there are two pages.

3.4.2 Cache coordination and heavy-hitter handling

In the previous subsection we explained our data structure for tracking a user's memory. That approach works on fixed size blocks of memory. However the cache works on smaller

units. The cache controller receives heavy-hit reports from the switch cache. These reports only contain a single hot key-value pair. For managing the content of the cache we need a more fine-grained solution. Moreover it also needs to work for multiple switch caches because one of our research question states that we want to manage the resources of multiple caches.

Inserting a new item into the cache starts by receiving a heavy-hit report from a switch cache. It is possible that multiple switches send a report about the same hot key because each switch does its own heavy-hitter detection. We think having the same item on different switches wastes memory space. Since the memory space on switches can be small we avoid duplicate items in the cache. Therefore we reject any heavy-hit report if the item is already in the cache. We used a hash table for detecting cached items. In the hash table we store every cached item. Checking if an item is in the cache is a fast operation since reading from a hash table is fast. If the item is not in the hash table then we will insert the item in the cache.

For inserting an item into the cache we must place it somewhere in memory. A cache page can store multiple items. The first step we take is grab a page from the page table and set this as the active page. Any new item is inserted on this page until it is full. After it is full we return the page to our data structure and grab the next page.

After we obtained a memory location we can start to insert the item into the switch cache. We would like to do this directly from the data plane but P4 only allows the P4Runtime to update P4 tables. This can only be accessed by the control plane. We will explain this implementation problem in more detail in Section 4. In order to complete the insertion we need to contact the control plane which will finalize the cache insertion and send a reply back.

The only thing the control plane inserts is a new record in a P4 table. The record does not contain the value. To insert the value we need to fetch it from a key-value store. With some routing we direct the request through the cache where the value will be stored.

3.5 INCOS control plane

We called our application that runs the control plane: INCOS. It has a few specific tasks which we have already highlighted in Figure 3.1. The most important task is to control the behavior of the switches in the data plane. There are three specific behaviors that the control plane controls for our caching application. First, it controls the users in the network. We will discuss user management in Subsection 3.5.1. Second, it needs to control

the cache pages which we will explain in Subsection 3.5.2. Finally, it needs to update the routing tables on the switches. We will highlight this in Subsection 3.5.3.

3.5.1 User management

In Section 3.3 we explained that one or more users can be active on a switch cache. In Section 3.4 we explained that the cache controller controls the content of the cache. However the users arrive and depart on the control plane. The control plane can be viewed as the operating system of the network [29]. We thought it would make sense that users are controlled by a central entity. Therefore similar to an OS which manages processes, INCOS manages users. We provided an user abstraction which is used by the NetCache controller.

Once a user arrives in our system we need to notify the cache controller and the switch cache itself. We install users on a switch cache in order to start tracking traffic for heavy-hit detection. We only insert a user on the switch when we allocate memory to that user. And we inform the cache controller about a new user so that it can create a page table which allows it to manage the cache content of the new user.

3.5.2 Resource management

We described in Section 3.4.1 that the memory in a switch can be viewed as one large array where we store the values. This memory space is used by multiple users and must carefully be partitioned. We partition the memory into fixed sized blocks which we called cache pages. The cache pages are used on the switch controller to manage the cache but it does not manage the cache pages. We manage the cache pages on the control plane because the control plane is similar to an operating system. This central entity which is responsible for controlling the network is an ideal location for also managing the memory allocation.

It is important to note that users will never directly interact with cache pages. We hide this complexity from users because users do not have direct control over the cache. It is our system that fills the cache and it fills the cache based on how frequent items are requested. Instead the cache pages are a simple method for managing the memory and handing it out in bulk to users.

Allocating pages is straight forward on a single device. Users request cache space and the requests are handled on a first come first serve basis. We keep handing out memory until we run out of available space. Once we run out of memory we reject the request. Users

can also leave from the caching service. After a departure, the pages are reclaimed on the switch. When the pages are reclaimed, they are available for reuse and can be handed out to different users.

Our cache also work with multiple switch caches. Here the memory is distributed over multiple devices. In this system allocating pages becomes harder because we need to choose on which device we allocate memory. We make the observation that users will only benefit from the cache if it is placed on path between the sender and receiver. Here, the sender is a client and the receiver is a key-value store. Allocating cache space which is not on the route of a request will never benefit the user. Therefore we should always allocate on the path between a sender and receiver.

In our design for allocation on multiple switches we assume there is only a single sender and a receiver. This avoids allocation problems when there are multiple clients and key-value stores such as multiple available routes. With such a problem pointing out good candidate switches on which we allocate memory to the user becomes unclear since there are multiple viable devices.

With this assumption we implemented an allocation policy that first computes the route from a client to a key-value store using Dijkstra's shortest path algorithm [37]. This results in a list of candidate switches which will benefit the user because all those switches are on path between the client and the key-value store. There are now multiple different methods for allocating memory on the list of caches. We think that allocating closer to the client will reduce the delay and network traffic. If requests are served closer to the client than the request has traveled less hops which reduces network latency. Furthermore serving requests early means it is not forwarded to the key-value store which results in less traffic on switches and servers downstream.

We favor the policy for allocating close to the client. A downside of this approach is that if possible everything is allocated on a single switch which is closest to the client. This approach can lead to load imbalance. In Section 5.3 we will experiment with three different policies in order to find the best approach for allocation with multiple caches.

3.5.3 Network controller

One of the core principles of software-defined networking is to decouple the forwarding plane from the control plane [38]. Our cache is placed in the data plane where it performs two tasks. It host a key-value cache which is controlled by our cache controller. The other task is packet switching where it forwards packets toward their destination. INCOS is our

control plane and not only must it manage the cache memory, but it also needs to manage the forwarding rules on the switches.

We run Dijkstra shortest path algorithm [37] for finding routes. We start from the switch and create a route to a destination. With this route we can find a port onto which we must forward traffic so it ends up at its destination. The last step for INCOS is to insert the new forwarding rule into the data plane.

4

Implementation

This section covers the implementation of all our components. These include the switch cache, the cache controller, the control plane, the client, and lastly the key-value store. For each component we will explain our implementation decisions and provide more insight into how the project was constructed. We start with the switch cache in Subsection 4.1. In Subsection 4.2 we go into detail on our cache controller. In Subsection 4.3 we discuss the implementation of our control plane. We finish this section by briefly explaining the implementation of our client and key-value store in Subsections 4.4 and 4.5 respectively.

4.1 Switch cache

We based our cache implementation on NetCache [10] and used the open sourced project [28] as our starting point. NetCache was written in P4-14 which is an older version of P4. When we started development we noticed that the compiler and software switch used by NetCache were deprecated. Therefore we decided to port the project to P4-16 which is the latest version of P4. This enables us to use currently supported P4 reference compiler p4c [39]. And we could use the supported reference software switch bmv2 [40]. With the upgrade to the latest version we were up to date with the latest technology surrounding P4.

Next, we started replacing the NetCache protocol with our own protocol described in Section 3.2. We added a user stage in the cache pipeline in order to recognize the active users of the cache. We modified the cache lookup to include user id's. We perform a cache lookup with a user ID and key pair. Figure 4.1 shows the lookup procedure. We have defined a match-action table which matches on user ID's and keys. If a match is found it will execute an action with a parameter. In our case we call an action which fetches the

value from memory using an index as an argument. The value array is programmed as a *register* type. Registers in P4 are pieces of stateful memory which can be modified in the data plane. This allows us to read and write the memory during packet processing.

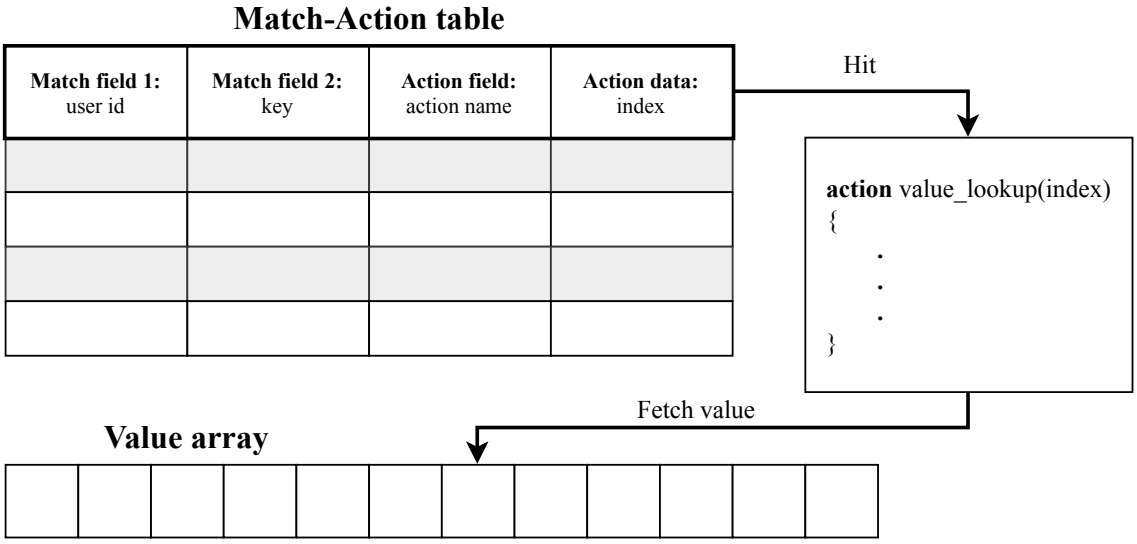


Figure 4.1: Cache lookup on our P4 switch.

The heavy-hitter detection is also implemented in P4. We slightly modified our detection from the NetCache implementation. NetCache only uses keys for detection. But this is not enough for our use case. We do heavy-hit detection on user-id's and key pairs. This allows us to identify hot items of different users even when they have the same keys.

4.2 Cache controller

We developed our own cache controller and chose not to reuse the NetCache controller. The NetCache controller we found could receive heavy-hit reports but it ignored them. Instead a predefined list of key-values was inserted into the cache at start up [28]. We wanted a system that is more flexible and does not ignore the heavy-hit reports.

We used Python3 to create our cache controller. The controller runs a socket server and most of the communication relies on UDP to reduce network latency. On top of the server we implemented our MADINC protocol to parse the packets we receive and also to encode message for communication with other components.

The cache controller cannot directly update the match-action tables of a switch cache. The only way to insert a new item into the switch cache is via the P4Runtime. This is a southbound interface for communication between the control plane and the data

plane. The cache controller is not part of the control plane and as such it is not exposed to the P4Runtime. Instead we communicate with the control plane for updating match-action fields. We have three opcodes which we described in Table 3.1 for messaging the control plane. We can use *CONTROL_INSERT*, *CONTROL_EVICT*, and *CONTROL_REPLACE* for inserting, evicting and replacing key-values in the cache respectively. The MADINC control plane will simply handle the updates from there but all the decision making is done on the cache controller.

Once this cache update is completed by the control plane we expect a reply from the control plane. This lets us know that the insertion was successful. The only thing that is missing are the values which we must request from the key-value store. We setup a TCP connection between the cache controller and a key-value store. We send an update request using our MADINC protocol opcode: *UPDATE_REQUEST*. After the key-value store replies back the switch cache will automatically learn this new value and write it to memory when the reply passes through the switch.

This communication is the only time we opted to use TCP. The downside of UDP is that the communication is not reliable. During testing we noticed that packets could drop under stressful loads. TCP can guarantee reliable data transfer. By using TCP we could reliably update the value on a switch. However for everything else we used UDP because it has less network overhead. Moreover using TCP for communicating between a switch cache and the cache controller would be difficult because then we need to establish a connection with a switch. We were unsure if we could implement the necessary code in P4 for creating TCP connection.

The control plane can also send allocation information to the cache controller by using the *CONTROL_ALLOC* and *CONTROL_FREE* opcodes in our MADINC protocol. In the value field of the protocol we encode specific page information. We explain the specific details in Section 4.2. The *CONTROL_ALLOC* message adds a specific cache page to a user while a *CONTROL_ALLOC* removes a page from a user.

4.3 Control plane

We also created our own control plane called INCOS. We implemented it in Python. At start up we provide a network view which is represented as a graph. We also provide the compiled P4 switch application and a p4info file. The P4info file is automatically generate after we compile the P4 application. This file contains additional information about the P4 application such as match-action tables.

In the control plane we created abstractions for the network, users and resources. For identification we assign unique id's to switches and users. With a unique ID we can more easily address specific switches and users. Moreover we use the unique switch ID's in our cache pages. Each cache page is defined as the following tuple: *(switch ID, page index)*. The switch ID specifies a switch cache and the page index defines a specific page on the switch. Each cache page has space to store 128 values.

For updating match-action tables on the switch cache we use the P4Runtime [41]. With the P4Runtime we can install users on a switch, update the cache lookup table and lastly update the forwarding table. The match-action tables are defined in a p4info file which are generated at compile time of the P4 application. We also use the P4Runtime for sending and receiving packets directly from a switch. The packets we receive do not pass the operating system but are provided straight to us. When we receive them they still contain the Ethernet, IP, UDP and MADINC headers. We chose to use UDP for all our communication because implementing TCP in user space would require too much time.

During allocation we communicate the *(switch-id, page index)* tuple to the cache controller. We encode this tuple into the value field of the MADINC protocol. The cache controller should have enough information to process the new allocation because have send a user ID along side the page tuple. We also receive information about cache updates from the cache controller. The control plane will simply perform the operations needed for inserting, evicting or updating the match-action tables.

4.4 Client

Clients are not part of our core caching design but clients are users that will utilize the cache in order to access data faster. We implemented our own clients so we can run our experiments. Each client implemented a socket server which would send queries to a key-value store. We provided each client with a workload file which contains a list of queries. The messages were all send using the UDP protocol because UDP has less overhead than TCP. This improves the performance for accessing data in the key-value store.

We created different types of clients. Each client was made specifically for an experiment. They differed on the fact that they measured different things. For example we implemented a client which measured the round-trip time(RTT) and we implemented a separate client which measured throughput.

4.5 Key-value store

The last thing we implemented was a key-value store. We implemented it with around 200 lines of in Python3 code. The implementation is quite simple. There is a socket server which handles the interaction with the network and a hash table which maps keys to values. On start up we provided a file containing different key-values to our key-value store. With the information in this file we fill our hash table.

We, initially, also wanted to use other key-value databases such as Redis. However we decided not to use them. We thought that creating a small socket server with a hash table was less work than trying to interface with an actual key-value store. It is still possible to use other implementations of key-value stores but you will need to implement a layer which translates the MADINC protocol to the API of a key-value store.

5

Experiments and results

In this section we describe the evaluation of our system. We evaluate our system on two main topics which are related to our research questions. We look at multi-tenancy and resource management. We start this section with a description of our setup. We explain the used hardware, software, and workloads. We then describe three experiments where we test user scalability, allocation policies, and caching distribution of different workloads.

5.1 Setup

5.1.1 Hardware and software

Our experiments are all running on a common testbed. We are using an Intel(R) Core(TM) i9-10980XE CPU which has 18 cores and 36 threads. We developed and ran all our experiments on a virtual machine. We used VirtualBox version 5.2 for our environment and it hosts the Linux distribution Ubuntu 16.04.

For our network we are using Mininet [42] which setups a virtual network. In Mininet we can quickly configure the network topology to our liking. In our virtual network there are 3 things we specify: hosts, switches, and links. For each experiment we will individually specify our topology.

We will not run large networks because Mininet does not scale well for large scale experiments [43]. Mininet runs every single component of the same hardware. With only 1 CPU we can only run so many components before it becomes too much and the network slows down due to hardware bottlenecks. Therefore we will limit the amount of software switches in our experiments to a maximum of 3. With this small amount of switches we will not get any issues.

The switches in Mininet are software switches which are capable of running our P4 application. The software we used is bmv2 [40] and is developed by the creators of P4. The performance of the bmv2 switch is not the same as real hardware and as such we will need to reconfigure our setup in order to match real hardware. See Section 5.1.2 for the software configuration.

The last two software applications are the clients and key-value stores. They run on hosts in our virtual network and are written entirely in Python3. We have written both applications ourselves in order to use our caching protocol and to run our experiments. We used multiple client scripts where each client measured different metrics. For instance, we have different clients for measuring throughput and delay. The key-value store on the other hand remains the same in all experiments. It is a simple application that runs server code and handles any requests for a key-value store.

5.1.2 Configuration

The final step in our setup is configuring the emulation to mimic the performance of real hardware. We use software switches which run in a virtualized environment on a CPU. This is not the same as the specialized hardware that is found in the real world. For instance the Tofino switches outperforms our software switch running on a CPU. In our initial tests we found that under heavy-load servers would outperform or match the throughput of our software switch. This is not a realistic setup and we decided to configure our setup to match the performance of other reported experiments.

The authors of the NetCache paper [10] reported that their servers could handle 35 million queries per second however their key-value store lowered this amount to 10 MQPS. Their Tofino switch could handle 4 billion queries per second but in reality their switch only managed to handle 2.24 BQPS. Their servers could not sent queries at a higher rate.

We want to replicate these numbers in our setup. The throughput ratio between a key-value store and the switch must be the same as the reported numbers in the NetCache paper. The reported performance ratio was 10 MQPS : 4000 MQPS = 1 : 400. Thus the throughput ratio for a key-value store and a switch is 1 : 400.

We measured our setup with the following experiment. We have 1 client that generates queries, 1 switch running our cache and 1 key-value store to handle read requests. we cached only 1 key in the cache and measured the maximum throughput of the switch. The single key throughput was 30 kQPS. With this number we determined the maximum capacity of a key-value store is 75QPS.

We think that experimenting with a single key-value store is not enough to fully test our switch cache. With a capacity 400 times smaller than a switch the key-value store will be the main bottleneck of our system. But we are more interested in testing our switch cache. Therefore we decided that a single Mininet host will have the throughput capacity of 32 key-value stores. This keeps the Mininet network small and reduces the key-value store bottleneck which allows us to benchmark the software switches.

5.1.3 Workload

We are going to run our experiments with different workloads. We used a uniformly distributed workload and three skewed workloads. Skewed workloads are common for key-value stores and they often follow a power law [44]. Therefore we will generate our workloads from a Zipf distribution which is also often used in evaluating key-value stores [45] [46] [47].

The Zipf workloads are generated with skewness 0.9, 0.95, and 0.99. There are 1000 keys in all four workloads and we generated at least 1 million requests for each workload. In experiments described in sections 5.2 and 5.3, we used larger workloads in order to increase the runtime.

5.2 User scalability

Our first experiment addresses the scalability of a single switch with multiple users. We want to evaluate if there is a performance difference between a single user and multiple users. We start with 1 user and scale our system up to 3 users. We could test the system with more than 3 users but we try to use a small amount of users because with more users we need to simulate more hosts on Mininet which increases the computational cost of running the experiment.

The experiment uses a simple topology, a star topology. Every user gets 2 hosts where the first host generates queries and the second one runs a key-value store. Every host connects to a single switch which therefore creates a star topology. Each user receives 25 cache pages which is more than sufficient cache space for running our experiment.

In order to benchmark the switch we have to be careful with the amount of requests we send to the switch. If we send too many requests then our software switch will start dropping packets. And if we send to few then we will not fully utilize the switch capacity. In a simple experiment where we cached a single key and always requested that same key. We found that the median throughput of the switch was 30 kQPS. For this experiment We

decided that we will send a maximum of 30 kQPS to the switch. The capacity of 30 kQPS will be shared equally by the active users in the experiment. For instance if we test 1 user then that 1 user sends 30 kQPS. If we have 2 active users then each user sends half of that.

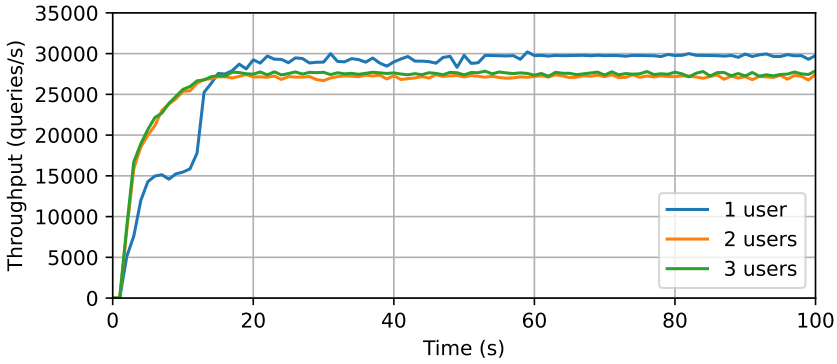
We used all the workloads mentioned in Section 5.1.3. In order to get a sufficiently long run we run our workload for at least 100 seconds. For each run we combined the throughput of the active users to get the throughput of the entire system. The results of the experiment are shown in Figure 5.1.

For all four workloads we noticed two distinct phases. A startup phase followed by a steady state phase. Initially the cache on the switch is empty and must first be filled with key-values. In the startup phase key-values are recognized as hot items and then placed into the cache. This startup phase lasts for about 20 seconds and then transitions to a steady state phase.

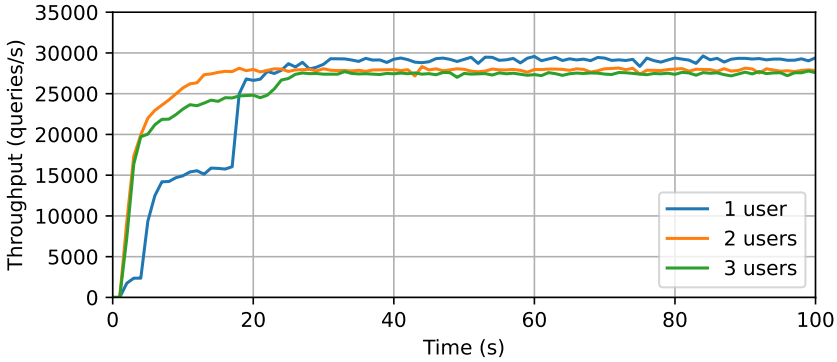
For the Zipf 0.9 and 0.95 workloads with a single user shown in Figures 5.1a and 5.1b respectively, we noticed the throughput levels off at around 15kQPS before it rises to 30kQPS. We thought this was strange and we expected it to smoothly rise to a steady state. We found out that not every key-value item was cached on the switch even though the switch would generate heavy-hit reports for every key. For Zipf workloads more than 90% was cached. Somewhere at the switch, cache controller, or the control plane the cache requests gets lost.

The lost packets are even more clear with the uniform workload in Figure 5.1d. Here we got three times worse performance for a one or two active users compared to the Zipf distributions. With a uniform workload the key frequency is the same for every key. This results in reaching the threshold for generating heavy-hit reports at the same time. Thus in our experiment a uniform workload generates a burst of heavy-hits in a short period of time. With a burst of traffic we go over the capacity of our software switch which results in packet getting dropped. We saw around 40% of heavy-hit reports being processed. The other 60% got lost somewhere during generating heavy-hit reports and inserting a key-value into the cache. We believe the reduced throughput in the uniform workload is the result of packet loss.

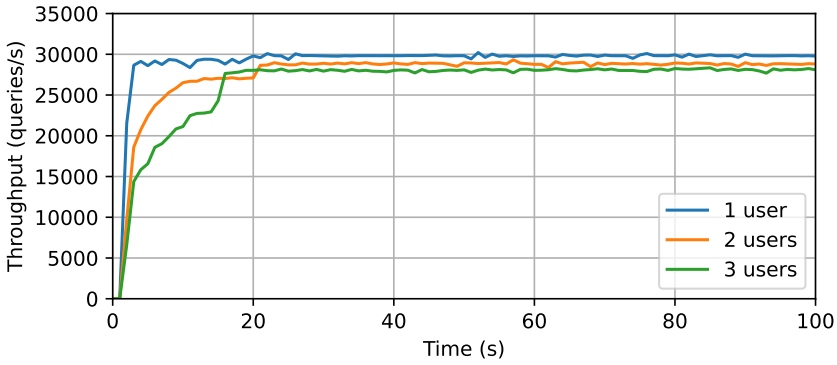
For our implementation we relied on UDP for sending control messages such as heavy-hit reports and cache insertions. If a packet is dropped with UDP then the information from that packet is lost. We did not implement a way to recover from packet loss. Thus in the startup phase of the experiment there is a spike of traffic. If this spike results in small packet loss then it can affect the caching logic. Which affects the hit rate of the cache.



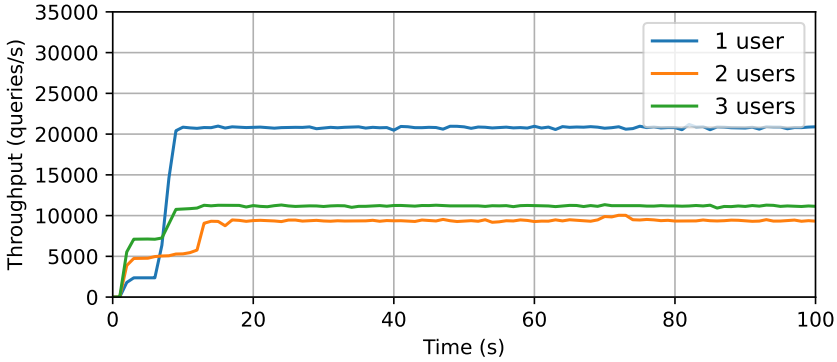
(a) Zipf 0.9 workload



(b) Zipf 0.95 workload



(c) Zipf 0.99 workload



(d) Uniform workload

Figure 5.1: Total query throughput of MADINC for 3 Zipf distributions and a uniform distribution. We experimented up to 3 active users.

This can in turn affect the throughput of the system if the key-value stores must handle the requests instead of the switch cache.

Eventually the experiment hits a steady state. For Zipf distributions we see that there is a small throughput difference between 1, 2 or 3 users active cache users. With a single user we were able to get the best performance for all workloads. For the Zipf 0.95 and 0.99 workloads, 2 users performs slightly worse than a single user and 3 users is slightly below that in terms of throughput. Interestingly for the Zipf 0.9 workload 3 active users performs better than 2.

5.3 Allocation policies

In this experiment we are testing the performance of different allocation policies in a multi-switch setting. With multiple switches we need an allocation policy for assigning cache space to users. We want to test different policies and compare the results. For this experiment we focus mainly on the throughput. We will first discuss our experimental setup. Then we will explain our tested allocation policies, and finally we will share our results.

Our setup is similar to our previous experiment but now we only use a single user. A single user limits the overhead of multiple users as we saw in Section 5.2. With only a single user we can focus the experiment on multiple switches. We use two switches in our topology and we connect them to each other. One switch is connected to a client and a cache controller. The other one is connected to a key-value store. This creates a line topology in which we run our experiment.

The client will reuse the four workloads described in Section 5.1.3 and sends queries at a rate of 30 kQPS. We allocate 20 cache pages to the user which will be partitioned by three different policies. 20 cache pages is enough space to fit the entire key space in the cache.

We have tested three different allocation policies. These policies provide in our opinion the simplest approach for allocating cache space. We labeled them as follows:

- Near-client
- Near key-value store
- Even split

Near-client allocates cache space as close as possible to the client in the network. We think this approach is ideal for reducing network latency of read request because the cache

can be accessed in the least possible amount of network hops. One disadvantage of this approach is that all the cache space is allocated on a single switch. Therefore, most of the workload is handled at a single point which results in an imbalanced workload on the available caches.

Near key-value store has the opposite approach of the near-client policy. Instead of allocating close to the client, we allocate close to the key-value store. We think this approach has worse latency than the near-client strategy because to access the cache, a request has to traverse more network hops which increases the latency.

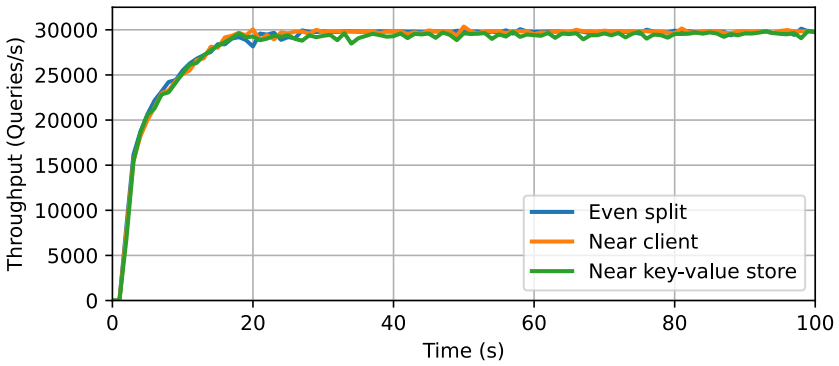
Finally, even-split tries to evenly allocate cache space along the route from client to key-value store. This is a more balanced approach compared to the other policies. We think this approach is similar to non-uniform memory access (NUMA) on commodity hardware. With even-split we effectively created multiple cache layers. Just like NUMA we think each layer adds a small delay to a memory request because the switches are physically separated from each other.

The throughput results of this experiment are shown in Figure 5.2. Just like our previous experiment there are two phases: a startup and a stable phase. The startup phase lasts between 10 to 20 seconds and then transitions to the stable phase. The Zipf distributions in Figures 5.2a, 5.2b, and 5.2c are showing similar throughput rates for each allocation policy. It seems there is no clear winner when running a Zipf workload.

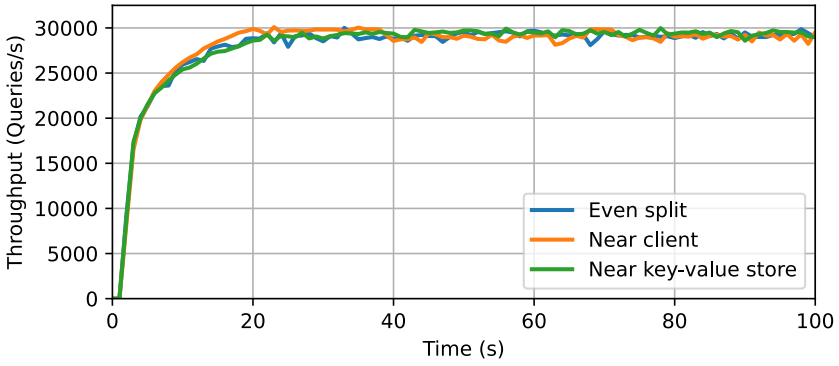
The uniform workload on the other hand is totally different as shown in Figure 5.2d. The throughput is much lower than the Zipf workload and it is more clear which approach is better. The even-split approach has roughly 2-3 kQPS higher throughput than the other two policies. But the throughput of even-split is more than 10 kQPS lower than our send rate. We are losing 10 kQPS with the uniform workload and even more for the other two policies.

We think the uniform workload shows similar issues as in our previous experiment. When there is packet loss we lose heavy-hit reports, cache insertions or cache read requests. Our system does not have any method to recover from packet loss. If the lost messages are critical for managing the content of the cache such as heavy-hit reports and cache insertions, then the cache loses performance.

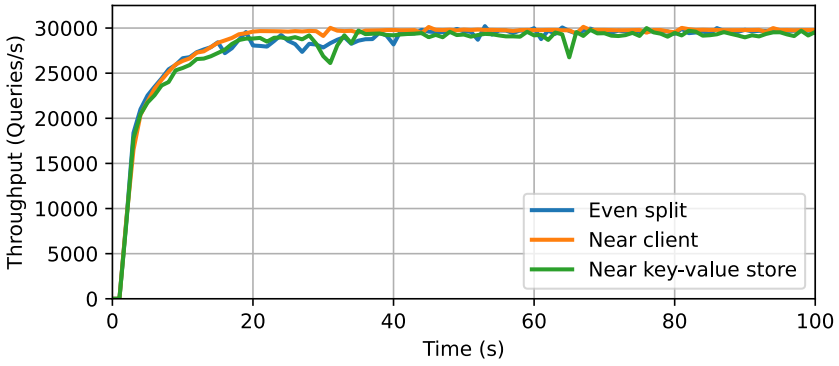
Next up we plotted the CDF of the throughput for each workload. The results are shown in Figure 5.3. For the Zipf distributions the results are close between the allocation strategies. But when we look at Figure 5.3c we see a small gap between the near-client and the other two strategies. This indicates that the near-client policy achieved consistently a higher throughput than the other strategies. We think that the near-client comes out just



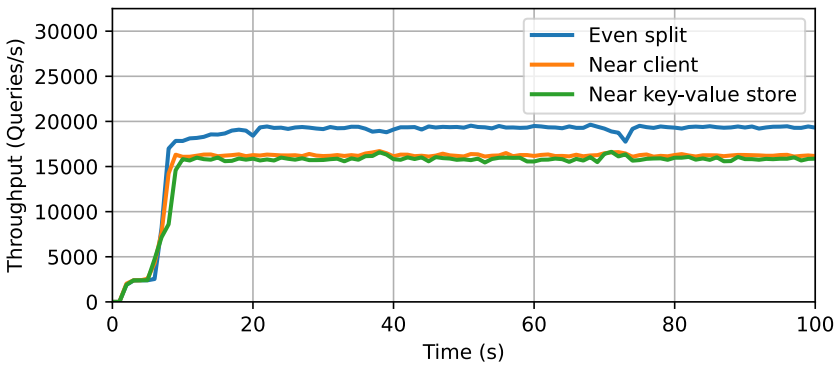
(a) Zipf 0.9 workload



(b) Zipf 0.95 workload

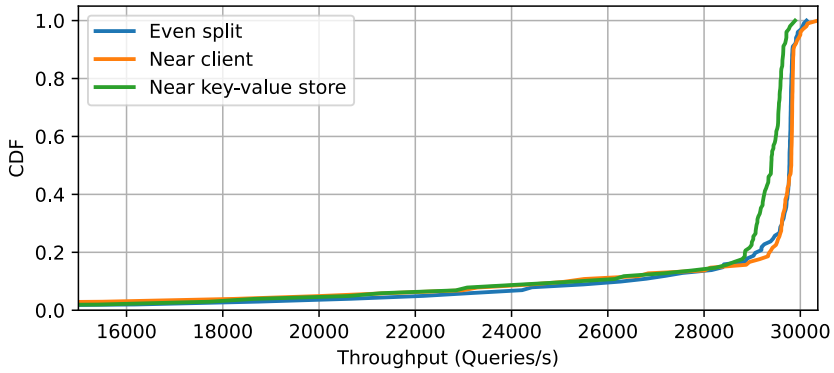


(c) Zipf 0.99 workload

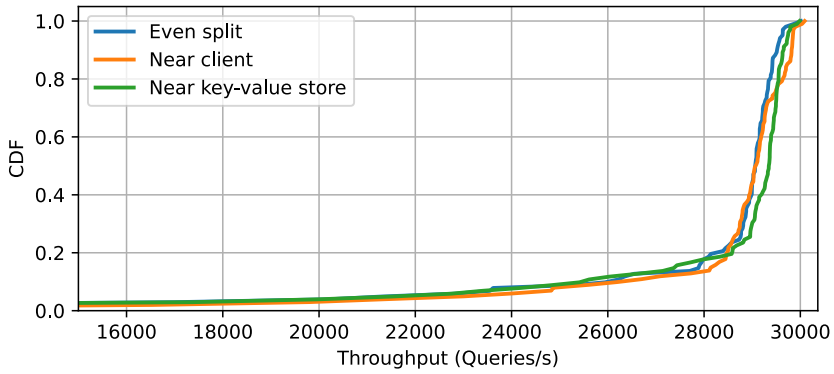


(d) Uniform workload

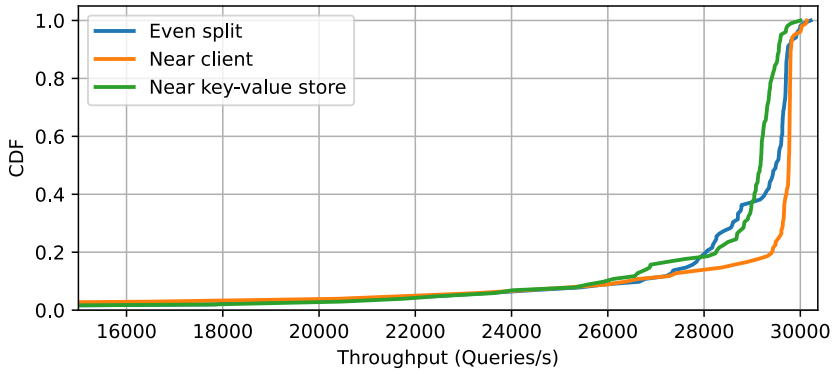
Figure 5.2: Query throughput of a single user with 3 different allocation strategies.



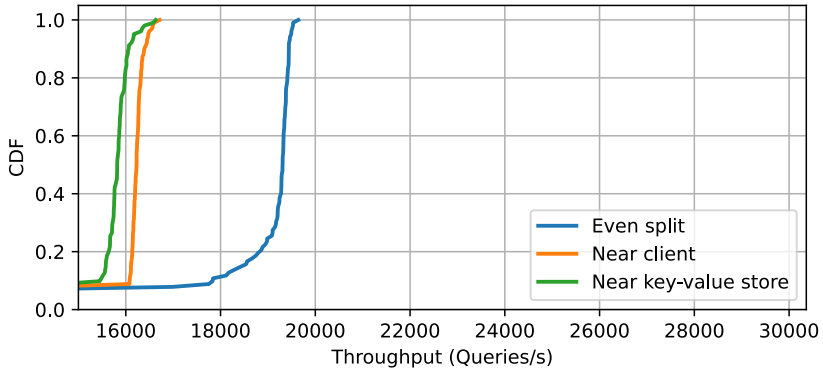
(a) Zipf 0.9 workload



(b) Zipf 0.95 workload



(c) Zipf 0.99 workload



(d) Uniform workload

Figure 5.3: Throughput CDF of 4 different workloads for each allocation policy.

ahead of the other policies for Zipf workloads because it performs slightly better with the Zipf 0.99 workload and similar to the other strategies with the Zipf 0.9 and 0.95 workloads.

Figure 5.3d shows the CDF with a uniform workload. We already explained that the uniform workload suffers from packet loss which affects the throughput. The reduced throughput is also visible with the CDF. The CDF also shows that the near-client performs slightly better than the near key-value policy because the CDF spikes up after 16 kQPS for the near-client strategy.

Overall this experiment shows that the even-split performs better for uniform workloads while the near-client performs slightly better in skewed workloads. But it is unclear if the result for the uniform workload still holds if our experiment did not suffer from packet loss for uniform workloads.

5.4 Cache distribution

In our third and final experiment we will measure the caching distribution. We are interested in how our system distributes the workload over the available switches and the key-value store. For our experiment we run multiple different workloads and measure how many requests a switch cache is serving. We achieved this task by assigning a unique value to each cache. A cache or key-value store will only reply with its unique value thus we can identify where a request was served by checking the replied value.

Next we also measure the round-trip time (RTT) of every request. We will use this data to plot a CDF of RTT. We think that caching key-values closer to the client, that generates queries, results in lower RTT because the packet traverses fewer network hops. In this experiment we used the four workloads described in Section 5.1.3.

We used a simple line topology with three switches. The client and cache controller are connected to the first switch and the key-value store is connected to the third switch. Every request from a client will need to pass three switches in order to reach the key-value store unless the request is handled at one of the switches.

We use a single user and give the user a total of 30 cache pages which is a sufficient amount for caching every key in the workload. The pages are allocated using the even-split policy described in Section 5.3. This policy evenly distributes the pages over the available switches. We chose not to use the near-client or the near key-value store policy because the 30 cache pages would all be allocated on a single switch and the other switch caches would be left unused. Measuring the caching distribution with a single switch is too predictable and we want to measure it with multiple switches.

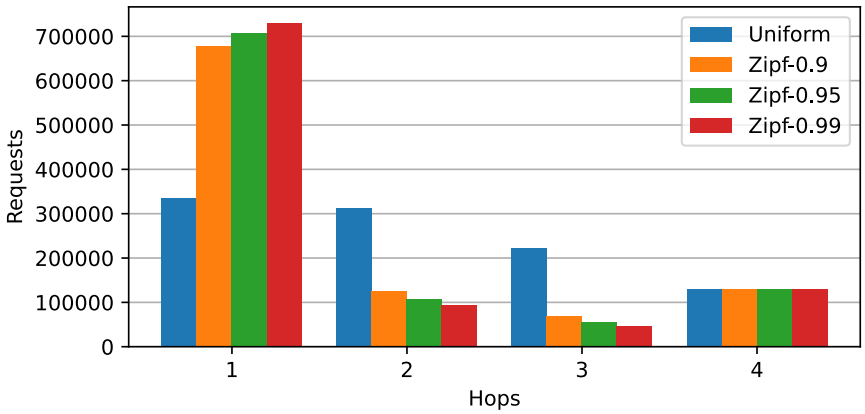


Figure 5.4: Network hop distribution of read requests. The first 3 hops are switches and the last one is a key-value store.

Figure 5.4 shows the results of this experiment. The Zipf distributions are all quite similar. We can see that from a cold start the most requested keys are all cached on the first switch. This result makes sense because the most frequently requested keys generate a heavy-hit report sooner than a less frequently requested keys. Therefore all the popular key-values are cached first and in our system this places them on the same cache page.

After the first cache page is full our system grabs the next page of a different switch using a round-robin strategy. The key-value items inserted on the next switch are less frequently requested due to the skewed key distribution. The lower the rank of the key the lower the key frequency. We see in Figure 5.4 that every Zipf distributions has a decline in workload on switch 2 and 3. These switch caches host less frequently requested key-values and as a result have server fewer request.

For the uniform distribution we noticed that every key-value pair was generating a heavy-hit report around the same time. We generate heavy-hit reports based on thresholds which are all triggered around the same time because the key frequency is the same. We also noticed that unlike the previous experiments, this one did not suffer from dropped packets because we are not sending requests at a high throughput. There is enough capacity on the switch to handle the experiment and not drop packets.

At first glance the uniform distribution is a bit strange. We expected that the switches would serve the same amount of requests because every cached item is requested with the same frequency yet there are fewer requests served on the third switch than there are served on the first or second switch. This workload imbalance can be attributed to cache page usage and the number of keys in the workload. Each cache page can store 128 key-

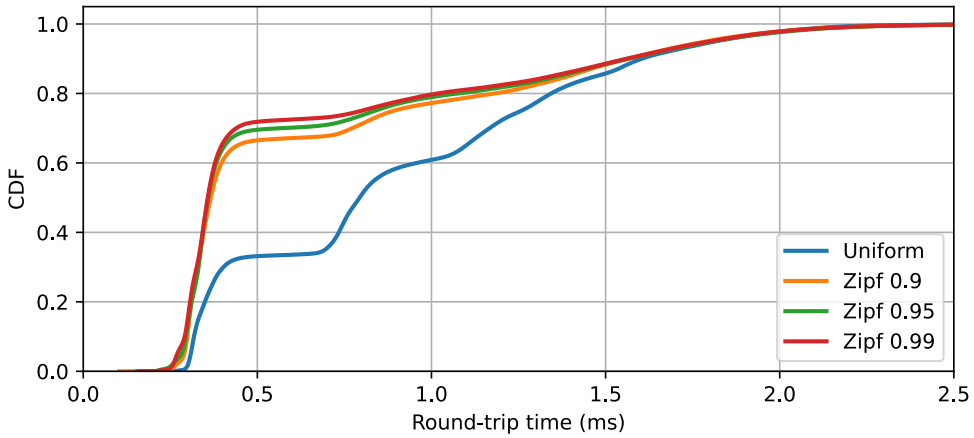


Figure 5.5: CDF of the round-trip time of read requests.

values. Our workload only contains a 1000 keys. In total we need around 7.8 cache pages to store the entire workload. We cannot distribute 7.8 cache pages over three switches evenly. With our round-robin cache page usage the first switch has three full cache pages, the second switch has 2.8 and the last switch has two full pages. Over time and with enough requests the first switch will have served more requests than the others because it hosts more key-values.

Another thing we noticed with the uniform distribution is that it takes more time for the cache to generate its first heavy-hit report. The reason for this is that a key from the uniform distribution has a lower frequency than the most frequent key from a Zipf distribution. As a result at the start of the experiment the cache remains unused for a longer period of time than with the Zipf workloads.

The last thing we noticed in Figure 5.4 is that the total amount of served requests on the key-value store remains the same for every workload. This result can be attributed to the fact that a key-value will be cached after being requested a fixed amount. This is true for every key in the workload. In addition after sending enough requests the entire workload gets cached. Thus in our experiment a key-value store would serve around: $n \times$ threshold requests where n is the number of keys in the workload.

We could lower the thresholds for generating heavy-hit reports in order to cache key-values sooner. This would increase the cache usage and could quickly shift the load from the key-value store to a switch. At the end we chose not to lower the thresholds and kept the same settings as NetCache. Optimizing the thresholds and improving the cache rate is something beyond the scope of this thesis.

Figure 5.5 shows the CDF of the round-trip time(RTT) of read requests. This figure shows us that there are steps in the RTT. For instance there is a step at 0.4ms, 0.8ms and 1.2ms. We think these steps are related to the number of network hops used for completing a query.

For Zipf workloads we see that more than halve of the workload has a RTT lower than 0.5ms. If we focus on the Zipf 0.99 workload, we see in Figure 5.4 that 70% of the requests were served on the first switch and in Figure 5.5 we see a similar amount of request that have a RTT of around 0.4ms. And if we focus on the uniform workload then around 33% of the requests were handled at the first switch. The CDF shows a step at 0.4ms with 30% of the requests, and a second step around 0.8ms with another 30% increase which in Figure 5.4 is similar to the amount of traffic handled on the second switch. The steps in the CDF must be related to the switches and the key-value store.

The CDF makes it clear that caching key-values close to the client reduces the RTT because requests travel fewer network hops. With the Zipf workloads the RTT is around 0.4ms for more than 60% of the workload.

This experiment shows that a cache we can reduce the latency of items by placing data close to the client. This latency reduction favors skewed workloads where we can cache the hottest items close to the client. However the uniform workload still benefits from caching items. For any workload caching items near the client is the best approach for reducing the latency as much as possible. But as Figure 5.4 shows, it can lead to load imbalance.

6

Discussion

In this section we are going to discuss the limitations and weaknesses of our in-network key-value cache. We start with the experiments where we discuss the limitations of our evaluation. Next, we discuss the two core goals of this thesis: multitenancy and resource management. We end this section with a discussion about our design. We will address some of the weaknesses and provide options for improvements.

6.1 Experiment limitations

We ran all our experiments on Mininet [42]. Mininet allowed us to quickly setup virtual networks on our computer. Setting up quickly and testing different network topologies was useful for development and testing. Doing the same on real hardware would be time consuming. However Mininet emulates the network in software. It does not physically create the network. Every host, switch, and link is therefore emulated. Even though Mininet tries to be accurate, software emulation is not the same as hardware. Consequently there is the possibility that our experiments produce different results on hardware.

In addition, we configured a key-value store to be more powerful than one running on actual hardware in order to simulate less hosts on Mininet. In Section 5.1.2 we explained that we wanted our setup to be similar to the NetCache setup which used hardware [10]. We reconfigured the performance ratio of our key-value store and switch to the same ratio as reported in the NetCache experiments. This reconfiguration resulted in a low query throughput for key-value stores. A single key-value store could not effectively benchmark our switch cache. To increase the workload we had to increase the number of key-value stores but Mininet does not scale well for larger networks [43]. Instead we chose to reconfigure a key-value store to be more powerful. In our experiments it is equivalent to 32

individual key-value stores. Because our key-value stores are more powerful and change the topology, we expect slightly different results on real hardware.

Another limitation of software emulation is that our P4 switch also ran on software. We used bmv2 [40] as our software switch. Just as with Mininet, a software switch cannot produce the same results as hardware. Additionally, bmv2 is not a production-grade software switch [48]. It is used for development, testing, and debugging. This can negatively impact the results we gathered.

Lastly we benchmarked our experiments with four different workloads. Every workload consisted of read-only queries with a 1000 possible keys. All the results we gathered, are based on read-only queries. Read-only was much simpler to implement and to test. Therefore we decided to only use read-only queries. As a consequence we cannot say anything about the performance of write queries.

6.2 Limitations on multitenancy support

We showed that our cache support multitenancy by adding a secondary key to the key-value items. The newly added key is a user-id. It is used for accessing user specific cache items. The original key is still needed to access a specific value. The two keys combined enables users to share a switch cache. This approach scales well with multiple users because each additional user only adds a small performance penalty.

Despite having a working solution for resource isolation, it is not that strong. The cache items all share the memory and they share the same match-action table. It is a user ID which separates the items of different users. A better approach for resource isolation is to not share the same application. Instead an OS can enforce better memory isolation. If we can run multiple cache applications on the same switch then an OS can isolate the memory. At the moment however, it is not possible to run multiple application on the same programmable switch.

Our approach also has its security weaknesses. Our switch cache processes every request as long as the switch cache can validate the user. We use a user ID for validating users on the switch cache. If an adversary knows a valid user ID and a valid key then an adversary gains access to a value. This is undesirable for sensitive data.

At the moment our user ID is defined as 1 byte in size. 1 byte is not a lot of possible values for a user ID. This field needs to be larger to allow more possible users and also to increase the amount of possible values a user ID can be. Adding more possible values makes it harder to guess a valid user ID. However we fear increasing the size of the user ID

will not do anything if traffic is not encrypted. Unencrypted requests can be intercepted by an adversary. To solve this problem, we can use encryption. However decrypting packets on the switch will decrease the performance because of the additional computing steps. Moreover encrypted traffic is one of the open challenges mentioned by [14]. Despite the security vulnerability we described, security is not one of our goals. We focused on other goals but we acknowledge that the problems exists. For any real world usage which includes caching sensitive data, our cache is probably not suitable to use.

6.3 Resource management limitations

Partitioning the memory of switch caches to fixed sized blocks allowed us to effectively manage the memory resources of a switch cache. It simplified the cache space into manageable blocks which reduced the complexity of memory management. Moreover our design and implementation showed we could manage the memory of multiple switch caches. Thus we were also successful in creating a distributed switch cache and managing their resources.

For future works we think a resource broker could be a valuable addition to our cache. A resource broker is a platform which offers the cache resources on-demand to its users. With a resource broker we can turn this key-value cache into a service. The platform should handle users and resource allocation. We did not implement a resource broker. Instead we implemented user management and resource allocation on the control plane at our INCOS controller. It handles at the start of our experiment all the allocations. For example, we ask for x amount of cache pages at start up and the INCOS controller handles this request. This was fine for our experiments but we cannot add or make new allocation via a network request. This limits the usability of the cache and we think that a resource broker can improve the usability of our system.

Switches often have limited memory. We therefore do not want to waste cache space. However our approach will only use allocated space. Everything that is not allocated remains unused. This approach wastes cache space which could be used by tenants. We could look into using the available space more effectively but it adds complexity to our existing allocation model. We did not have the time to investigate better approaches of resource utilization and instead focused on managing the resources.

6.4 System design limitations

One of the issues we identified in our experiments was packet loss. The packet loss occurred when we stress tested our system. The bmv2 software switch has some issues with performance and packet loss [48]. It sometimes drops packets when under high throughput or bust traffic. It is likely that the packets were dropped at the software switch but we were unable to confirm this suspicion. One thing we do know is that our cache performs worse when there is packet loss on the network.

Packet loss demonstrates a flaw in our cache design. Our cache is unable to recover from packet loss. As a consequence this halts cache updates that are in progress because the information is lost. For instance, the switch cache sends a heavy-hit report to the cache controller but this report gets dropped. The information in that report is lost. We will never recover this loss because the switch cache expects that the report is received. So it does not resend the report. Meanwhile the cache controller does not receive this report and will never start a cache update.

We used UDP for communicating between the different components. UDP is a connectionless protocol. It is a simple to use protocol which made it also simple to implement in P4. Moreover it was also simple to setup socket servers on the other components. UDP does not guarantee message delivery. If we want to recover from packet loss we must either choose a different transport protocol which does ensure message delivery or implement our own packet loss recovery on top of UDP.

For a different transport protocol we could use TCP or QUIC [49] for example. Both protocols ensure message delivery. Our preference would go to QUIC because it uses UDP. It is often implemented in libraries and thus it runs in user space. In our implementation of the INCOS controller we received packets via the P4Runtime. These packets still contained the headers and we had to write our own packet parser in Python. We did not want to implement our own TCP stack to parse the TCP headers. Therefore QUIC has our preference because it has some of the benefits of UDP and can be used with the help of libraries.

On the other hand, implementing TCP or QUIC in P4 will be more difficult. We are not sure if it is possible to establish connections with a switch or even implement these protocols. P4 is not a general programming language and it places limitations on programmability in order to create efficient programs. Using TCP or QUIC for generating messages on a switch will therefore prove difficult.

On another topic, we could improve our system design by looking how other applications use the principles of SDN. Currently we run the components of the application, control and data plane on different devices. For example the cache controller and INCOS controller run on different hosts. The Onix [20] and ONOS [21] controller are SDN controllers that run network code in applications on top of their system.

We can decide to package the cache controller as an application that runs on the INCOS controller. This could have a number of potential benefits. First this simplifies the system design since we removed one network component. Secondly we can simplify our cache protocol. At the moment our protocol has mixed-use. We provide user methods such as read and write but our protocol also support operations for cache management such as page allocation information. Page allocation does not provide keys or values so those fields are useless for allocations. Instead we use the key and value fields for sending allocation info. By removing operations related to page allocation we simplify the protocol.

The third and last benefit is potentially faster cache updates. We removed the cache controller from the network. This means message do not have to route to this component. Instead messages originally destined for the cache controller are send to the control plane. This change effectively reduces the amount of network communication which could result in faster cache updates.

However, there is a downside to this new design. The control plane controller needs more functionality which increases its complexity. Moreover, the control plane has to do more work since it also runs the cache controller logic. The control plane controller must be able to handle the additional load. Therefore scalability is a concern.

7

Conclusion

In this thesis we provide a design of multitenant and distributed in-network key-value cache. We enabled multitenancy by adding a secondary key to every key-value item. This secondary key is a user ID which allows us to distinguish cache items of multiple users. Next, we partitioned the cache space into fixed sized blocks which we call cache pages. Each page has a unique id which identifies two things. First, it identifies a switch cache and second, it identifies a memory range on that switch cache. The cache page abstraction allowed us to effectively manage a multitenant and distributed cache.

We implemented our design and evaluated the performance in terms of throughput and latency. We used both skewed and uniform workloads and we found that our system introduces a small performance penalty for each user active on a switch cache. Furthermore we found some evidence that cache space located closer to a user results in better latency and a small improvement in throughput. Despite these findings, we also discovered that packet loss has a significant impact on the performance. Our design does not recover from packet loss. This causes us to lose cache updates which as a result deteriorates the performance of our cache.

References

[1] N. Khan, I. Yaqoob, I. A. T. Hashem, Z. Inayat, W. K. Mahmoud Ali, M. Alam, M. Shiraz, and A. Gani, "Big data: survey, technologies, opportunities, and challenges," *The scientific world journal*, vol. 2014, 2014. 1

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009. 1

[3] "Microsoft azure - cloud computing services." `https://azure.microsoft.com`. Accessed on 2021-05-03. 1

[4] "Amazon web services (aws) - cloud computing services." `https://aws.amazon.com`. Accessed on 2021-05-03. 1

[5] "Google cloud - cloud computing services." `https://cloud.google.com`. Accessed on 2021-05-03. 1

[6] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014. 1

[7] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *2011 6th international conference on pervasive computing and applications*, pp. 363–366, IEEE, 2011. 1

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007. 1

[9] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *Proceedings of the*

Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 795–809, 2017. 2, 6

[10] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 121–136, 2017. 2, 5, 6, 9, 25, 32, 45

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013. 2, 7

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. 2, 5, 7, 8

[13] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. Ports, and A. Panda, "Multitenancy for fast and programmable networks in the cloud," in *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020. 2

[14] D. R. Ports and J. Nelson, "When should the network be the computer?," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 209–215, 2019. 2, 47

[15] T. A. Benson, "In-network compute: Considered armed and dangerous," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 216–224, 2019. 2

[16] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014. 5

[17] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013. 5

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008. 5

[19] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014. 6

[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, *et al.*, "Onix: A distributed control platform for large-scale production networks.," in *OSDI*, vol. 10, pp. 1–6, 2010. 6, 49

[21] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 1–6, 2014. 6, 49

[22] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pp. 143–157, 2019. 6

[23] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pp. 35–49, 2018. 6

[24] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pp. 1–7, 2015. 6

[25] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "Netagg: Using middleboxes for application-specific on-path aggregation in data centres," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 249–262, 2014. 6

[26] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pp. 150–156, 2017. 6, 7

[27] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019. 6

[28] X. Jin and Z. Bai, "Source code of netcache." https://github.com/netx-repo/netcache-p4, 2018. Accessed on 2020-08-31. 10, 25, 26

[29] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008. 13, 21

[30] D. Hancock and J. Van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 35–49, 2016. 14

[31] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019. 14

[32] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 98–111, 2018. 14

[33] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005. 17

[34] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 101–114, 2016. 18

[35] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004. 18

[36] "Linux memory management concepts overview." https://web.archive.org/web/20200617021515/https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html, Jun 2020. Accessed on 2020-09-28. 18

[37] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959. 22, 23

[38] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications surveys & tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014. 22

[39] "P4_16 reference compiler." https://github.com/p4lang/p4c. Accessed on 2021-04-22. 25

[40] "The reference p4 software switch." https://github.com/p4lang/behavioral-model. Accessed on 2021-04-22. 25, 32, 46

[41] "An implementation framework for a p4runtime server." https://github.com/p4lang/PI. Accessed on 2021-04-26. 28

[42] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pp. 1–6, 2010. 31, 45

[43] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–6, IEEE, 2014. 31, 45

[44] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pp. 53–64, 2012. 33

[45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010. 33

[46] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 31–44, 2016. 33

[47] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "{MICA}: A holistic approach to fast in-memory key-value storage," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pp. 429–444, 2014. 33

[48] "Performance of bmv2." https://web.archive.org/web/20210513133510/https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md. Accessed on 2021-05-13. 46, 48

[49] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, pp. 183–196, 2017. 48