A Thesis for the Degree of Ph.D. in Engineering

Multi-layer Key-value Cache Architecture with In-NIC and In-kernel Caches

February 2019

Graduate School of Science and Technology
Keio University

Yuta Tokusashi

Preface

In datacenters, e-commerce services and cloud computing services are deployed for customers, accommodating hundreds and thousands of computers. CPU power consumption and cooling systems take more than half of a datacenter's power consumption. Thus, building highly efficient datacenters is a significant challenge. In recent years, the growth of network interface speed is increasing, while the growth of CPU performance is leveling off. This gap would be increasing continuously. That is, network-based applications would be CPU bound for more performance. To solve this, field programmable gate arrays (FPGA) based solutions and in-kernel/kernel bypassing solutions have been explored over the last five years. Traditionally, cache hierarchy has been installed and has been studied when we faced the gap between CPU and memory. Thus, this thesis discusses cache hierarchy for network-based application, where a cache is built in both network interface card (NIC) and network protocol stack as network data path. Key players start to deploy FPGAs in their datacenters to accelerate their services and to achieve high energy efficiency. In these five years, key-value store acceleration using an FPGA has been actively studied.

This dissertation explores cache hierarchy on network-based applications. There are a variety of design options such as write policy, eviction policy, inclusive cache vs. non-inclusive cache and so on. The designs are implemented on the NetFPGA platform and its performance is measured and analyzed. Furthermore, this thesis shows the memcached, one of key-value stores, architecture for an FPGA equipped with NIC. The FPGA card has two types of store regions: an on-chip RAM and on-board off-chip DRAM modules. The proposed architecture consists of the first level cache with on-chip RAM and the second level cache with on-board DRAM modules. This thesis shows practical performance results using this NIC, compared with an existing hardware-based memcached and software. Latency, throughput and power consumption are drastically improved, compared to the existing systems.

This hardware solution is applied to distributed denial of service attack (DDoS) as a practical case, since recently DDoS amplification attack traffic has critically increased. An ICMP-based DDoS detection scheme is proposed to fit with key-value store. This thesis conducts an experimental evaluation and shows the feasibility.

Thoughout this dissertation, in-NIC cache, in-kernel cache and cache hierarchy are studied for a network-based application. It is demonstrated that multi-layer key-value cache architectures will be helpful to bridge speed gap between the growth of CPU performance and of network performance.

Acknowledgments

My doctoral study has been supported by a number of people. I would like to thank all of them.

My supervisor, Associate Professor Hiroki Matsutani, has always supported my study. He has always given valuable suggestions whenever I had a difficulty in my research. He has always reviewed every paper that I have written and given me valuable comments to improve the papers. I would like to thank for his constant support to my study.

I am grateful to my doctoral committee members, Professor Hideharu Amano, Professor Kenji Kono, and Professor Osamu Nakamura for their careful reviews and comments to my thesis.

I appliciate to Dr. Noa Zilerman, who has been my collaborator since I was visiting student at Computer Laboratory, University of Cambridge. She gave me a bunch of research oppotunities to collaborate with researchers who actively researched in the area of networking and system research. In writing this dissertation paper, she gave me a lot of comments. I am really grateful for her help and her concern.

Many thanks for Dr. Yohei Kuga and Dr. Takeshi Matsuya, who have been my collaborators since I was an undergraduate student. They gave me a lot of advices for research and technology. I appliciate their moral support constantly. I am also gratefull to Dr. Michio Honda and Dr. Andrew W. Moore for giving me an oppotunity to collaborate with NetFPGA team in University of Cambridge.

Finally, I am grateful to my parents and family.

Yuta Tokusashi
Yokohama, Japan
Feburary 2019

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

A datacenter serves cloud computing and e-commerce services with accommodating hundreds of thousands of computers, which consume a lot of power due to CPU and cooling systems, for the purpose of satisfying customers requirements. In the article [2], CPU and cooling systems took up half of all power consumption in a datacenter. In 2015, Microsoft started to operate a field programmable gate array (FPGA) in their datacenters in order to accelerate their search engine system Bing, the Catapult project [3]. They achieved high energy efficiency by using an FPGA. In addition, Amazon AWS provides F1 instances serving FPGA logic cells to their customers in a cloud computing environment [4]. An FPGA, therefore, has been considered to reduce power consumption and to improve the performance of their applications.

In a datacenter, a variety of network services are served. A representative one of them is key-value store. Key-value store is commonly used as a cache and storage of online services such as e-commerce and social networking services, mostly running in cloud computing environments [5]. For the purpose of performance improvement [6], key-value store deployments in datacenters are often scaled-out, leading in turn to increased power consumption. Thus, this dissertation focuses on key-value store as a network service running in a datacenter.

S1 - Yahoo! Sherpa database platform system measurements, version 1.0 (33 K)The growth of network interface speed is increasing, e.g., to 100GbE and 400GbE. On the other hand, the growth of CPU performance is holding steady due to recent slowdown of Moore's law. Applications along with network processing would be CPU bound due to this limitation because it cannot be expected that CPU performance would be improved to satisfy further network speed. Traditionally, to solve the speed gap problem, cache hierarchy has been used in the case of the speed gap between CPU and memory. We had faced the "memory wall" problem [7] where CPU and memory have both unbalance performance and speed. Therefore, CPU cache hierarchy [8] has been studied and installed into computer architecture. At the beginning of CPU, CPU and memory did not have a significant speed gap. Thus, CPU had access to DRAM directly until the speed gap became conspicuous. Due to

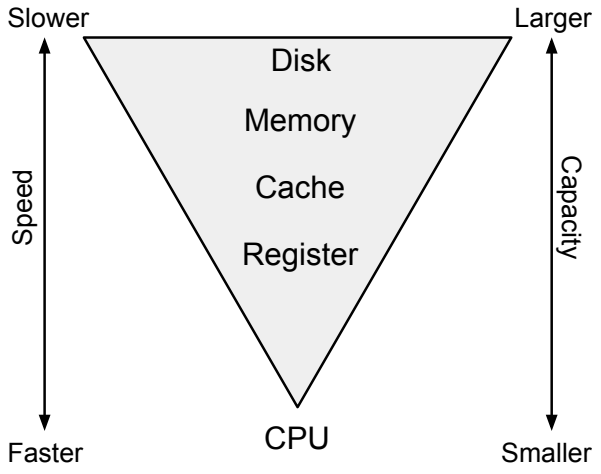


Figure 1.1: Typical CPU cache hierarchy.

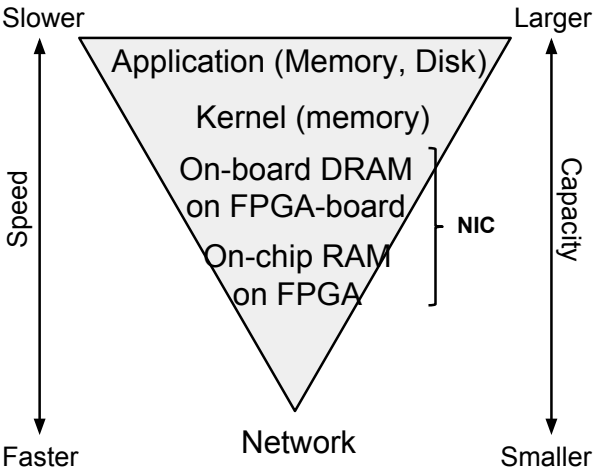


Figure 1.2: Network-based cache hierarchy.

miniaturization of transistors and evolution of microprocessor, we, however, faced a memory wall where speeds of CPU and memory do not match, resulting in installing cache hiearchy on current microprocessor architecture as shown in Figure 1.1.

In client-server model, both a client machine and a server machine need to process networking. As mentioned above, the gap between network interface speed and CPU performance would be increasing. Considering FPGA trends, network-based cache hierarchy using FPGA-based NIC equipped with DRAM modules can be considered as shown in Figure 1.2. Similar to CPU cache organization, various design options could be considerable, e.g., write policy, inclusive or non-inclusive and eviction policy, etc.

In this dissertation, network-based cache hierarchy is proposed and design options are explored for building cache organization of FPGA-based NIC. This thesis also shows a concrete key-value store architecture on FPGA integrated with the NetFPGA platform [9], which is an open source platform for prototyping network devices. Finally, this thesis introduces a practical application, DDoS mitigation, one of the important Internet security issue.

1.2 Objective

In this dissertation, the research objective is to explore the design space on multi-layer key-value store combining in-NIC cache design and in-kernel and to improve performance and energy efficiency of key-value store. To bridge the speed gap between CPU and network interfaces, an FPGA- based key-value store acceleration and software optimization which use kernel-bypassing approach to remove overhead of network processing on CPU and is used for building a cache on kernel have been studied.

An FPGA-based key-value store acceleration is built on an add-in card with PCI express as an interface with a host machine or an FPGA board. Due to the physical size of an add-in card and the limitation of FPGA's I/O pins for differential signals, DRAM modules which could be assembled

is limited. That is, cache capacity on FPGA board is small, compared with general key-value store running on a host. Regarding the energy efficiency, an FPGA-based key-value store acceleration is higher than CPU-based key-value store.

On the other hand, in-kernel key-value store and kernel-bypassing key-value store remove the bottleneck of CPU bound, especially, at the network protocol stack. These key-value stores utilize large main memory on a motherboard as a cache. Since these approaches run on CPU, they would lead to high power consumption. That is, it is hard to reduce power consumption rather than FPGA-based approach.

Hence, an FPGA-based key-value store and in-kernel key-value store are a trade-off concerning cache capacity and power efficiency. Similar to cache hierarchy on CPU, cache hierarchy for a network-based application can be considerable — an FPGA-based key-value store as the first level cache (L1 cache) and in-kernel key-value store as second level cache (L2 cache). To design cache hierarchy based on network datapath, various cache design options could be possible. Through this dissertation, design options for network-based application and FPGA-NIC design along with a key-value store processing core are explored.

1.3 Contributions

To address this issue, we introduce multi-layer key-value cache architecture, which is a networked system for high energy efficiency in a datacenter. First of all, we introduce a concept of multi-layer key-value cache architecture using FPGA-based in-NIC cache and in-kernel cache. As level one cache (L1 cache), we use in-NIC cache leveraging on-board DRAM on an FPGA for a cached media. As level two cache (L2 cache), we use in-kernel cache using host memory. In addition, we explore the design options, similar to CPU cache hierarchy: write through vs. write back, cache associativity and eviction policy, etc.

The second, we provide a concrete FPGA design using the NetFPGA platform for memcached, one of key-value store deployments. Similar to the CPU, we built a first level cache on on-chip RAM and a second level cache on on-board DRAM. When a query is hit on a cache of on-chip RAM, the response can be handled without DRAM, so latency and performance can be improved. We provide more information about design and performance analysis.

Finally, we adopt an FPGA-based key-value store acceleration to DDoS mitigation platform on the Internet. We also propose a detection scheme and maintain fine-grain rules on key-value store. We assume that we place it on the wire, thus mitigation hardware requires line-rate processing.

1.4 Thesis Organization

We introduce related work on in-NIC cache using an FPGA and in-kernel cache, including cases which used kernel-bypassing acceleration, in Chapter 2. We show the relationship of each Chapter

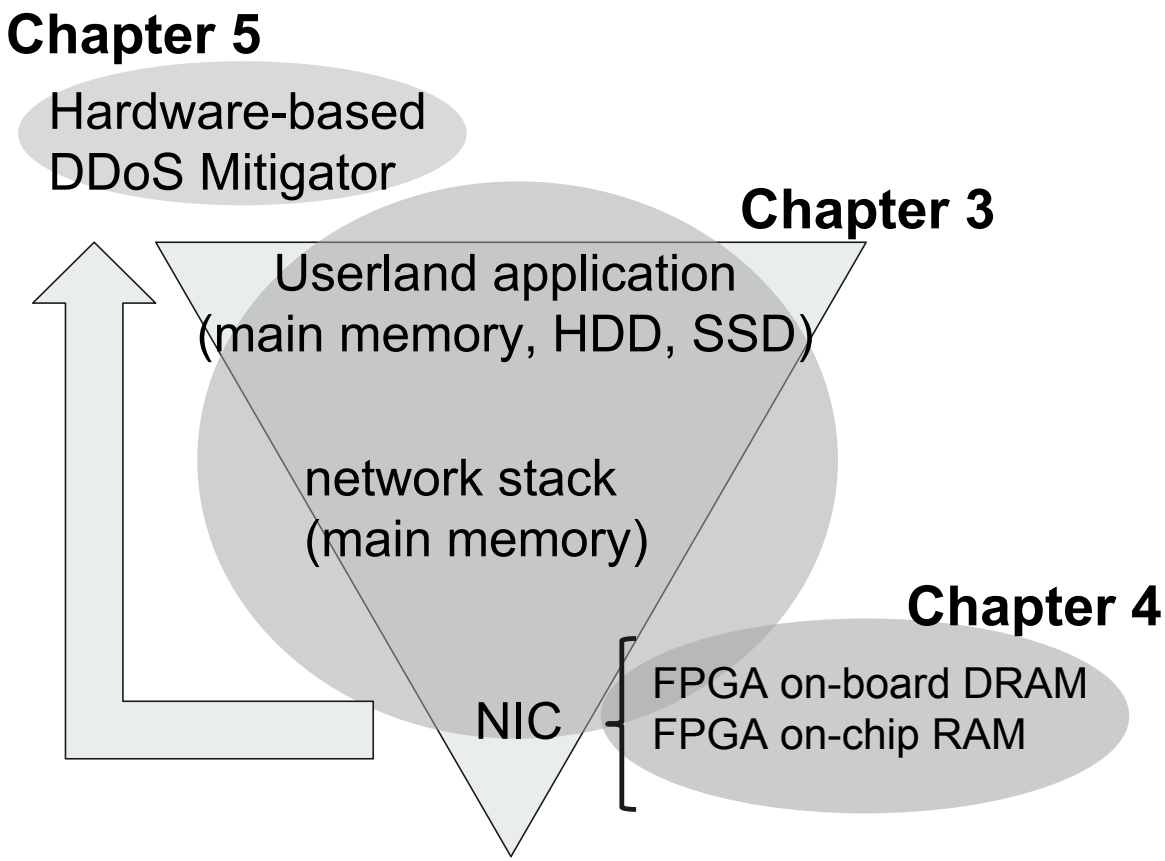


Figure 1.3: Relationship of chapters.

3-5, as shown in Figure 1.3. Chapter 3 proposes multi-layer key-value cache architecture, which is a core proposal in this thesis. We also provide the detail of design options building the architecture, which is our first contribution in this dissertation. Further, Chapter 4 focuses on in-NIC cache design with hierarchy design combining on-chip RAM and on-board DRAM, which is the second contribution. In this chapter, we take up in-NIC cache of the proposed multi-layer key-value cache architecture described in Chapter 3, and introduce cache hierarchy inside in-NIC cache and the design. Chapter 5 introduces the application case for DDoS mitigation, which is the third conribution. The hardware-based key-value store described in Chapter 4 is applied to DDoS mitigation. We conclude this dissertation in Chapter 6.

Chapter 2

Related Work

The bottleneck on key-value store is considered to be the network protocol stack of the operating system [10]. To solve this problem, two representative approaches have been proposed: hardware-based key-value store acceleration and software-based key-value store acceleration. This chapter will survey the related work: the detail and pros-and-cons on hardware-based key-value store acceleration and software-based key-value store acceleration. Section 2.1 introduces key-value store and the challenges for performance. Section 2.2 presents how an FPGA is used in two cases: industry and academy to assist network protocol stack on kernel. To scale the performance of key-value store, the remaing sections surveys the works realted to key-value store acceleration in aspects of hardware-based and software-based acceleration. Section 2.3 introduces hardware-based key-value store and Section 2.4 introduces software-based key-value store. Section 2.5 shows the hierarchy key-value store design which uses different storage devices.

2.1 Key-value Store

Key-value store is one of Not only SQL (NOSQL) databases [11]. NOSQL database compensate the disadvantages of RDBMS (e.g., transaction is supported in RDBMS). NOSQL databases have simple data structure, which some databases have schemaless. Some NOSQL databases sacrifice consistency in order to distribute queries to machines.

Key-value store traditionally is used for web cache and web application's backend storage in an aspect of simple API and scalability using consistent hashing, compared with RDBMS (Relational DataBase Management System) [12]. Key-value store writes key-and-value pairs on host's main memory or storage such as SSD and HDD. Simple APIs consist of primitive GET(key), SET(key, value) and DELETE(key), issuing a read request, a write request, and a delete (writing zero) request respectively. Key-value store calculates a hash value of its key to retrieve the descriptor on hash table entries when key-value store receives a query. Generally, a hash function (e.g., lookup3 [13, 14], md5, cityhash [15]) is used for hash calculation. Hash table contains arrays of a data structure which holds the descriptor to the key-value data. When key-value store identifies the requested key from

the table, the paired value with the key is returned to the client.

The survey from [16] described that memcached which is on e of key-value store implementation is CPU-demanding application. Further, it reported that CPU utilization from 55% to 85% takes Linux network stack. Rosenblum, et al also indicates that GET request of memcached latency analysis shows memcached's user application processing time is around 7% [10,17]. From this surveys, key-value store processing is CPU-demanding application, which especially taking Linux network protocol stack. Silver bullets for these problem is taking an dedicated hardware (e.g., FPGA and ASIC) and software optimzing (e.g., Intel DPDK and netmap).

2.2 An FPGA

A common and promising approach to significantly improve the energy efficiency of key-value store is to replace a part of the software with an application specific custom hardware. To build such custom hardware, recently, FPGAs have been widely deployed for datacenter applications [10,13,14,18–25], due to the reconfigurability, low power consumption, and a wide set of IP cores and I/O interfaces supported.

2.2.1 Use Cases in Industry

Microsoft started to integrate FPGA in their datacenter to accelerate their search engine and cloud computing services as Catapult project in 2015 [3]. Microsoft Bing serves web search engine to customers and embedded an FPGA into a server to accelerate ranking stack [18]. This was the first attempt to deploy FPGAs in a datacenter. Microsoft then accelerated their cloud computing platform, Azure [26]. In Azure, the SmartNIC equipped with an FPGA is used as a network interface card. SmartNIC has some acceleration functions related to networking. Amazon AWS provides FPGAs on virtual machines(EC2 instance) as F1 instance [4]. Customers can easily develop a service with an FPGA via PCI express to accelerate an application by using "FPGA Developer AMI (Amazon Machine Image)" and "Hardware Development Kit". One virtual machine instance is able to utilize up to eight FPGAs.

2.2.2 Use Cases in Accademia

NetFPGA project [9] has developed a networking platform under an open source license. NetFPGA project released two types of FPGA boards equipped with 10GbE connection. The one is NetFPGA-10G. An FPGA device used is Xilinx Virtex-5 XC5VTX240T. Four 10GbE network interfaces are used for communication. 288MB RLDRAM-II memory and three x36 QDR II memory are available.

Another FPGA board is NetFPGA SUME [27] as next generation model of NetFPGA-10G. NetFPGA SUME card is equipped with four SFP+ ports, 8GB DDR3 DRAM modules, three x36 72Mbits QDR II SRAM, PCI Express Gen3 x8, and Xilinx Virtex-7 690T FPGA. Since NetFPGA project

Table 2.1: Summary of in-NIC processing approaches.

Ref.	Type	Platform	Storage	Parallelism
[10]	Standalone	FPGA+1GbE	NIC DRAM	Two cores
[19]	Standalone	Dedicated SoC	Host DRAM	Single accelerator is depicted
[14, 20]	Standalone	FPGA+10GbE	NIC DRAM	Deep pipeline
[25]	Standalone	FPGA+10GbE	NIC DRAM+SSD	Deep pipeline
[29]	NIC+host	FPGA+40GbE	Host Memory	Fully pipelined
Proposed	Cache	FPGA+10GbE	NIC DRAM	Many cores (crossbar connected)

provides reference switch, reference NIC and reference router design for targetting each board, researchers can develop custom hardware based on these reference designs under the license.

For further network interface card (e.g., 40GbE, 100GbE), since memory and CPU processing overhead is inefficent, FlexNIC [28] was developed to acheive efficient packet transferring on DMA. Moving applications from userland to NIC achieves high throughput and low latency. FlexNIC was used for a variety of applications for functionalities: key-value store. apache storm and intrusion detection.

2.3 Hardware-based Key-value Store Acceleration

Table 2.1 summarizes the above-mentioned existing designs and our L1 cache design of the multi-layer key-value cache. The existing designs can be used as a standalone key-value store and the combination of a host and FPGA-NIC, while dedicated hardware as an L1 cache of the proposed multi-layer key-value cache is used in this thesis. Since complete key-value servers are assumed to be running on an application layer, dedicated hardware is operated just as an L1 cache and sophisticated functionalities (e.g., logging, error handling, and data replication) can be left to the software key-value servers. In the existing designs, GET requests are processed by dedicated hardwares, while SET requests are processed by software or hardware, depending on how complicated memory management is implemented.

A key application in datacenters is a distributed data store, such as memcached [30], used as a large-scale data store and memory caching system. Because FPGA devices can be tightly coupled with I/O subsystems, such as high-speed network interfaces [9, 27], their application to memcached has been extensively studied recently [10, 13, 14, 19–25, 31]. An experimental result in [14] shows that an FPGA-based standalone (i.e., FPGA board only) memcached accelerator improves the performance per Watt by 36.4x compared to an 8-core Intel Xeon processor. Even with a host, it improves the performance per Watt by 15.2x. However, a serious limitation of such FPGA-based memcached accelerators is that their cache capacity is limited by DRAM capacity mounted on the FPGA boards. As a DRAM module typically has more than 200 I/O pins (e.g., 204 pins for DDR3 SO-DIMM

package), the number of DRAM modules handled by a single FPGA cannot be increased easily, as mentioned in [25]. As DRAM capacity for a host main memory is growing, the capacity gap between host memory and FPGA-based NIC should be addressed.

This approach first appeared in [10], which proposed an FPGA-based standalone memcached appliance that utilizes DDR2 memory modules and a 1GbE network interface on an FPGA board. The memcached appliance consists of dedicated hardware modules. A network processing block and a memcached application block with slab-based memory management modules for supporting various data lengths. Both GET and SET requests are processed by the dedicated hardware modules. It can be parallelized by duplicating the memcached appliance cores.

Another memcached accelerator is designed as a dedicated SoC in [19]. It leverages the hardware prototype proposed in [10] for GET requests, while it relies on general-purpose embedded processors for the remaining functionalities, such as memory allocation, key-value pair eviction and replacement, logging, and error handling.

A notable FPGA-based standalone memcached accelerator was proposed in [14,20]. It leverages DDR3 memory modules and a 10GbE network interface on an FPGA board. To fully exploit the parallelism of memcached requests, the request parsing, hash table access, value store access, and response formatting are pipelined deeply. GET requests are processed by the pipelined hardware, while a host CPU assists with memory management functionalities required for SET requests. To handle key collisions by hardware, up to eight keys mapped to the same hash table index are looked up in parallel (i.e., 8-way). Hash items with an expired timestamp in the same set are freed when they are accessed for SET requests. Recently, this design is extended to support SATA3 SSDs in addition to DRAM as storage [25] so that key-value pairs are stored in SSD or DRAM regions, depending on the value length.

KV-Direct [29] demonstrated a SmartNIC achieving a high query rate (e.g., ~180Mqps), but was limited to key-value store operations only, was a proprietary solution using 8B query size, batching multiple queries in a single packet, and processing vector queries.

2.4 Software-based key-value store acceleration

In addition to such FPGA-based solutions, software-based optimizations have been studied to improve the performance of data stores [15,16,32–36]. A latency breakdown of memcached reported in [10] shows that a packet processing time for NIC and kernel network protocol stack is longer than that spent in memcached software. Actually these software-based optimizations mainly focus on how to reduce the processing time for kernel network protocol stack, and they can be classified into two approaches: kernel bypassing and in-kernel processing.

2.4.1 Kernel-bypassing Approach

One solution to the bottleneck in the network stack is a kernel-bypassing approach. Luigi Rizzo developed netmap [37], which is packet I/O framework using kernel bypassing to reduce overhead on network protocol stack of an operating system. In the netmap framework, Ethernet device driver passes a packet to netmap ring buffer to access userland application without any specialized hardware. However, network processing is needed on userland application to develop netmap-based application. Intel DPDK (Data Plane Development Kit) framework is a library feature which packets can access userland application directly and are processed on a specific core, resulting in achieving high performance.

To improve the throughput of key-value store, a holistic approach that includes the parallel data access, network access, and data structure design was proposed in [15]. As the parallel data access, data are partitioned and a single CPU core exclusively accesses a partition. As the network access, Intel DPDK is used so that the server software can directly access NIC by bypassing the network protocol stack to minimize the packet I/O overhead. As the data structure design, different data structures are proposed for the cache and store modes. The network access optimization (e.g., kernel bypassing) is mainly taken into account in this thesis. The other optimizations are orthogonal to the multilevel NOSQL cache and can be applied for further efficiency.

2.4.2 In-kernel Approach

Another solution to the bottleneck is to build an application cache within kernel stack directly in order to reduce overhead on the network protocol stack.

As proposed in [16], moving the key-value store into the OS kernel is an alternative approach to remove most of the overhead associated with the network stack and system calls. In a kernel layer, received packets are hooked by Netfilter framework [38], and only key-value store queries are retrieved. The retrieved queries are processed inside the kernel with an in-kernel hash table. The response packet is generated and sent back to the device driver. The received *sk_buff* is reused for the response to reduce memory copies in the kernel.

In the proposed multi-layer key-value cache architecture, since in this work complete key-value servers are assumed to be running on an application layer and thus sophisticated functionalities, such as data replication, are left to the key-value servers, the in-kernel cache approach is embedded as an L2 cache in the proposed key-value cache hierarchy. The L2 cache design will be illustrated in Section 3.5.

2.5 Hierarchical Key-value Store Design

In 2018, hierarchical key-value store design appeared in [39]. They built a cache organization on a host CPU with NVM. Specifically, they use DRAM, NVM, and TCL Flash as the level one cache, the

level two cache and main storage, respectively. Recent NVM serves persistent storage with middle latency and middle capacity between DRAM and Flash. Different from this approach, this thesis focuses on network datapath.

2.6 Summary

This chapter surveyed FPGA-based key-value store acceleration and software-based key-value store acceleration and showed that there are trade-offs in terms of cache capacity and power efficiency. The following chapter introduces the proposed multi-layer key-value cache architecture combining FPGA-based and software-based acceleration to complement their disadvantages and to bridge network speed and CPU performance speed.

Chapter 3

Multi-layer Key-value Cache Architecture

This chapter introduces multi-layer key-value cache architecture. Section 3.1 provides an overall view of cache hierarchy concept. Section 3.2 introduces design options to building the proposed multi-layer key-value cache architecture. Section 3.3 and Section 3.4 show details of L1 key-value cache and L2 key-value cache, respectively. Section 3.5 shows a simple implementation for a proof of concept. In addition, a simulator is built for each design option and evaluates them in Section 3.6. Lastly, this chapter is summarized in Section 3.7

3.1 Key-value Cache Hierarchy

This section introduces a multi-layer key-value cache architecture combining in-NIC and in-kernel cache, which is the main contribution in this dissertation. Figure 3.1 illustrates our multi-layer key-value cache architecture that complementally combines the in-NIC and in-kernel caches in order to fully exploit the highly energy-efficient in-NIC processing, while addressing the capacity limitation by the in-kernel cache that utilizes a huge host main memory. It is assumed that one or more key-value databases, such as key-value store, column-oriented store, document-oriented store, and graph database, are running as software servers on a machine, where FPGA-based network interfaces (FPGA NICs) are mounted for receiving and responding key-value queries. On-board DRAM capacity of the FPGA NICs is used as the L1 key-value cache, while a host main memory allocated by the kernel module is used as the L2 key-value cache.

When the FPGA NIC receives a packet, the received packet header is parsed, and if it is a key-value query, it is processed inside the FPGA NIC; otherwise, it is transferred to an Ethernet device driver as well as common TCP/IP packets. For key-value queries, a key-value pair is extracted from the packet and the corresponding key is looked up from a hash table in the FPGA NIC. If the key is found in the hash table, the value stored in the on-board DRAM is returned to the requestor (i.e., L1 key-value cache hit); otherwise, it is transferred to an Ethernet device driver as well as common TCP/IP packets (i.e., L1 key-value cache miss).

In the Ethernet device driver of our L2 key-value cache, the received packet header is examined

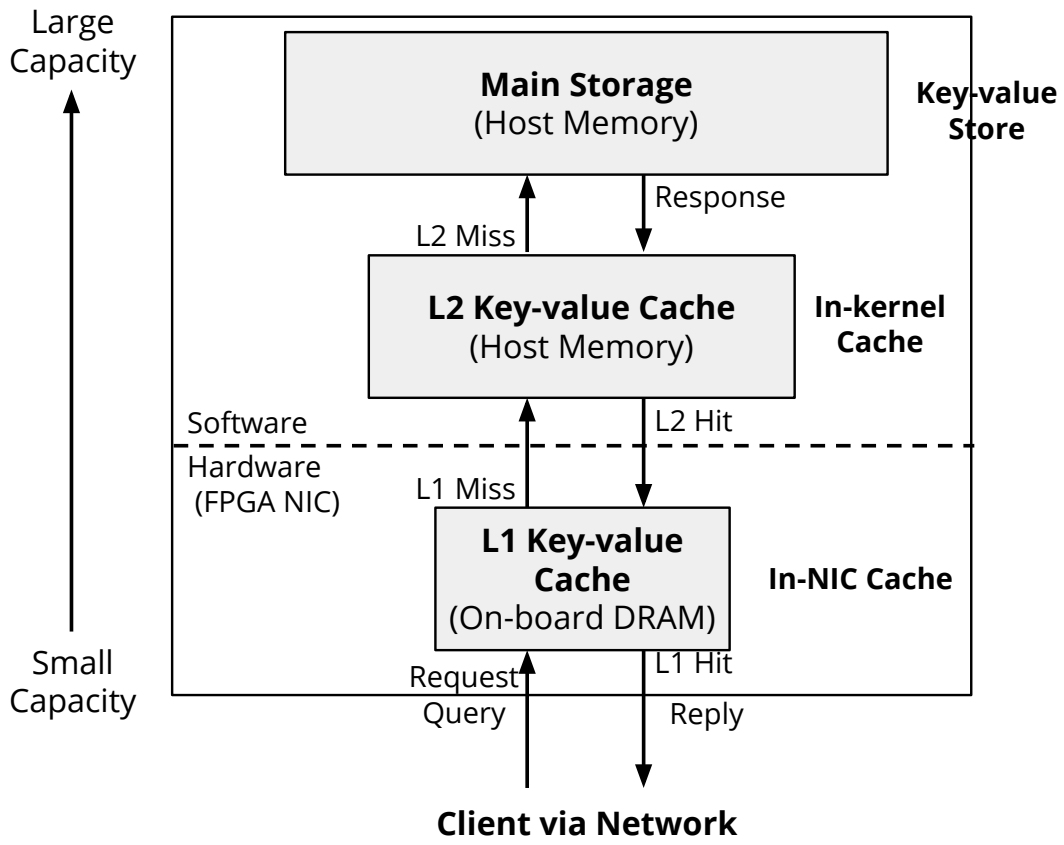


Figure 3.1: Relationship between L1 and L2 key-value caches.

again, and if it is a key-value query, it is processed inside the in-kernel cache module; otherwise, it is transferred to a standard network protocol stack as well as common packets. For key-value queries, a key-value pair is extracted from the packet and the corresponding key is looked up from a hash table in the in-kernel cache. If the key is found in the hash table, the value stored in the in-kernel cache is returned to the requestor (i.e., L2 key-value cache hit); otherwise it is transferred to a network protocol stack as usual (i.e., L2 key-value cache miss). In the case of L2 key-value cache miss, the key-value query is transferred to an application layer and processed by a corresponding key-value software server.

Similar to CPU cache hierarchy, there are a variety of design options to build multi-layer caches. Here, we re-visit design options in CPU cache and memory management system used in general opearation systems.

1. **Cache write policies between L1 and L2 caches**

   To update cache, there are two ways for cache: write-through and write-back. Generally, write-through is writing data from CPU is updating to both cache and memory. An advantage is that data consistency is kept since the contents on both cache and memory always correspond.

Meanwhile, write-back can write only cache then, data is returned back to memory later. An advantage is high availability of cache management, compared with write-through. Drawbacks are keeping consistency, which is difficult to identify the inconsistency data and to control the data to keep consistency.

2. **Cache associativities**

   To map memory, contents on cache has mainly three ways: direct map, set-accosiative, and full-associative. **Direct map** is that cache line is determined by memory address. Advantages are that developing cost is cheap and the replacement algorithm is simple. **Set-associative** cache is that N-way data is stored per memory address. Set-associative needs to select which data should be chosen. In addition, if data is needed to be replaced, the one of N-ways must be selected. Against this replacement, an algorithm could be FIFO, LRU, and RANDOM. **Full-associative** cache is not accessed by the pointer. All cache lines are retrieved. Hit ratio can be improved. However, design cost is high and the design itself is complicated.

3. **Inclusion policies between L1 and L2 caches**

   The content of cache and memory could be shared or not-shared: inclusive policy and non-inclusive policy. In inclusive policy, the memory contains cache contents, whereas memory excludes cache contents in non-inclusive policy. Therefore, when replacement happens on cache, the data removed from cache and the data is transferred to memory. This function takes development cost for the building. Meanwhile, inclusion policy is a simple way in an aspect of building cache.

4. **Cache eviction policies on L1 cache**

   Eviction policy happens when set-associative cache is used and the cache is needed to be replaced. Representative algorithms are LRU, Random, and FIFO. **LRU** (Least Recently Used) is that the data which is not used the most frequently, is replaced of N-ways. **Random** algorithm chooses one of N-ways to replace with random algorithm. **FIFO** (First-in, first out) algorithm evicts the firstly accessed data of N-ways.

5. **Memory management**

   Generally, key-value store uses slab allocator or log-structured merged-tree as a memory management mechanism. **Slab allocation** is utilized on kernel's memory management in addition to memcached. (e.g., Linux, FreeBSD, and HP-UX). **Log-structure merged-tree** is used for database systems. This memory management takes inserting operation for write. Since random access is not used, this write scheme is faster. Inserting operation takes $O(1)$ on avarage, but find-min takes $O(N)$ on avarage. Therefore, LSM-tree is suitable for a write-intensive application. This memory management is used general key-value store (e.g., LevelDB and RocketDB).

3.2 Design Options in Multi-layer Key-value Caches

In this section, design options in multi-layer key-value caches are overviewed, based on the previous section.

3.2.1 Write-Back vs. Write-Through

Cache write policy is an important design choice for multi-layer cache design. In the proposed multi-layer key-value cache, write-back policy can reduce the traffic amount between L1 and L2 key-value caches, because written data are not transferred from L1 to L2 key-value caches until modified (i.e., dirty) cache blocks in the L1 key-value cache are evicted. Although in write-back policy the cache blocks in L1 key-value cache are not consistent with those in L2 key-value cache internally, such inconsistency is never exposed to applications. In write-through policy, cache blocks in L2 key-value cache are updated whenever those in L1 key-value cache are updated, and thus the traffic amount between L1 and L2 key-value caches increases. The problem of such write-through policy is that the write performance of L1 key-value cache is restricted by the L2 key-value cache bandwidth. Section 3.6.2 will evaluate the write-back and write-through policies.

3.2.2 Cache Associativities on Hash Table

As illustrated in Section 3.3, to access cached key-value data in value store, the key is hashed to compute the index in the hash table where a start address of the cached key-value data in Value Store is stored. There is a possibility that different keys generate an identical hash value, but only one of them can be stored in hash table and the others will be evicted. Such a situation is called a hash conflict.

To avoid hash conflicts and key-value cache misses, we can increase the associativity of hash table. More specifically, in the n-way set associative hash table, up to n different keys whose hashed values are identical can be stored in hash table. As the number of ways increases, L1 key-value cache misses due to the hash table conflicts are typically decreased. For example, an 8-way hash table is used in Xilinx's FPGA memcached appliance to avoid the hash conflicts [14]. Note that in the proposed design the key length assumed is variable and if the key is too long to be stored in a single hash table entry, the key is stored using multiple hash table entries. In this design, a key with up to 64B size can be stored in a single hash table entry. A key larger than 64B is stored in multiple hash table entries. Section 3.6.3 will evaluate the set-associative design of hash table in terms of L1 key-value cache miss ratio.

3.2.3 Inclusion vs. Non-Inclusion

Inclusion policy (e.g., inclusive or non-inclusive) is an important design choice for multi-layer caches. When the proposed L1 and L2 key-value caches are inclusive, cached data in L1 key-value

cache are guaranteed to be in L2 key-value cache. When they are non-inclusive, cached data are guaranteed to be in at most one of L1 and L2 key-value caches.

Non-inclusive policy has an advantage in terms of cache efficiency especially when L1 key-value cache size is comparable to that of L2 key-value cache, while inclusive policy is simple to implement, as illustrated below.

- Assuming that a GET query is missed at L1 key-value cache and hit at L2 key-value cache, if non-inclusive policy is enforced, the cached block in L2 key-value cache is transferred to L1 key-value cache and then the cached block in L2 is deleted. If inclusive policy is enforced in the same situation, the cached block in L2 is not deleted.

- Assuming that an unmodified cached data in L1 key-value cache is evicted, if non-inclusive policy is enforced, the cached data in L1 key-value cache is transferred to L2 key-value cache and then the original cached block in L1 is deleted. If inclusive policy is enforced in the same situation, we do not have to copy the cached block in L1 to L2.

- Assuming that a query is missed at both L1 and L2 key-value caches, if inclusive policy is enforced, the requested data retrieved from key-value server are required to be cached in both L1 and L2 key-value caches. This behavior can be easily implemented, because the requested value retrieved from key-value store server is naturally transferred to the client machine via Ethernet device driver (i.e., L2 key-value cache) and FPGA NIC (i.e., L1 key-value cache).

Non-inclusive policy is advantageous in terms of cache efficiency, while inclusive policy is simple to implement in L1 key-value cache. However, inclusive policy increases the traffic between L1 and L2 key-value caches, which may degrade the throughput. Section 3.6.4 will evaluate the inclusive and non-inclusive policies in terms of total cache miss ratio when L1 and L2 key-value cache sizes are varied.

3.2.4 Eviction Policies on Hash Table

Section 3.2.2 introduced set associativities on hash table design to reduce L1 key-value cache misses due to hash conflicts. By building multiple ways for hash table, multiple key-value pairs whose hashed keys are identical can be stored in hash table. Assuming that a new key-value pair is going to be stored in Hash Table but all the ways are in use, one of existing ways in hash table will be replaced with the new key-value pair. Eviction policy defines which way will be evicted in such situations.

Here we discuss the following three eviction policies.

- Random: One of the ways is randomly selected to be replaced with a new key-value pair. A simple implementation without a random generator is that the hashed value of the new key-value pair is used to select one of the ways to be evicted.

- LRU: The least recently used way is selected to be evicted. The design complexity increases when the number of ways is greater than two.

These eviction policies will be evaluated in terms of L1 key-value cache miss ratio in Section 3.6.5.

3.2.5 Slab Size Configurations in L1 key-value Cache

Section 3.1 introduced memory management scheme on key-value store: slab allocation and log-structured merged-tree. For key-value store acceleration, slab allocation is selected for memory management system since slab allocation is used as preallocation memory, hence, it is easy to utilize it.

According to a memcached workload analysis [40], 90% of the requested data sizes are less than 1KB. The study shows that requested key-value sizes are mostly about 20B in USR workload, while large values with up to 1MB are requested in ETC and APP workloads. As illustrated in Section 3.3, in the proposed L1 key-value cache, a list of free memory blocks are preliminarily allocated for each value size (e.g., 64B, 128B, 256B). The number of memory blocks preliminarily allocated for each value size should be carefully selected. It can be further optimized if the workload is predictable so that the majority of requested key-value data can be fit to the allocated memory blocks.

In this work, L1 key-value cache is implemented on an FPGA board. As L1 key-value cache capacity is limited, allocating large memory blocks (e.g., 1MB blocks) may significantly reduce the number of small-sized cache blocks and increases the L1 key-value cache miss ratio. Therefore, it may be required to determine the upper limit of memory block sizes allocated. In this case, key-value data larger than the upper limit are not cached in L1 key-value cache.

Section 3.6.6 will evaluate various configurations of sizes and the numbers of memory blocks with memached traces in terms of L1 key-value cache miss ratio.

3.3 L1 Key-value Cache Architecture

Figures 3.2 and 3.3 show a datapath of the proposed L1 key-value cache implemented in an FPGA NIC. Only packets with key-value queries (i.e., key-value packets) are extracted based on their service destination port number and only key-value packets are transferred to dedicated PEs (e.g., PE1) for hash table lookup. Other packets are transmitted to a host machine with a DMA controller via PCI-Express. Thus, an arbiter is implemented in front of the DMA controller to arbitrate two input sources: key-value traffic which are not hit in hash table (i.e., L1 key-value cache miss packets) and non-key-value traffic.

L1 key-value cache stores key-value pairs in the on-board DRAM modules equipped on an FPGA board. The key and value parts are typically processed as variable-length data. As surveyed in Section 2.3, existing in-NIC key-value accelerators [14, 20] employ a single deep pipeline in which value parts are processed as variable-length binary data. However, a wide diversity of value lengths (e.g.,

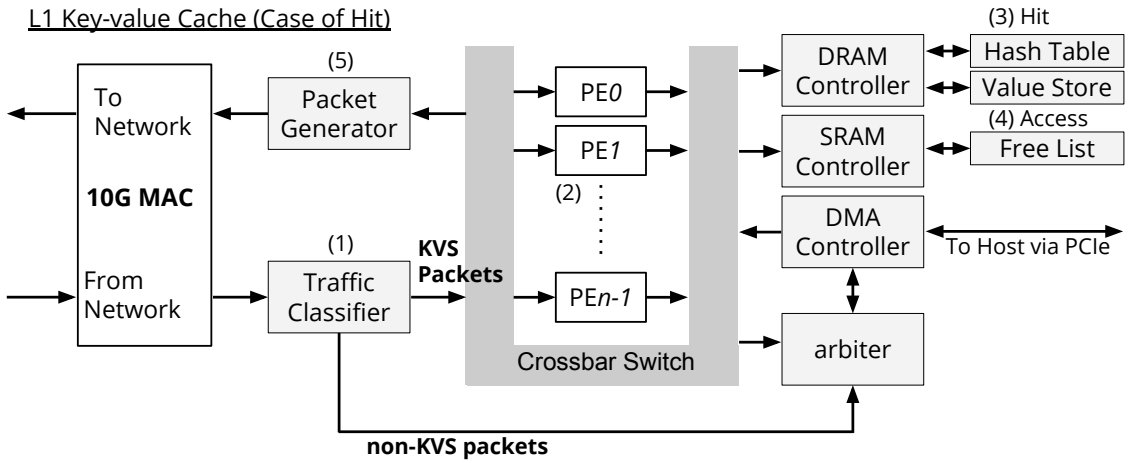


Figure 3.2: L1 key-value cache hit on heterogeneous multi-PE design of L1 key-value cache.

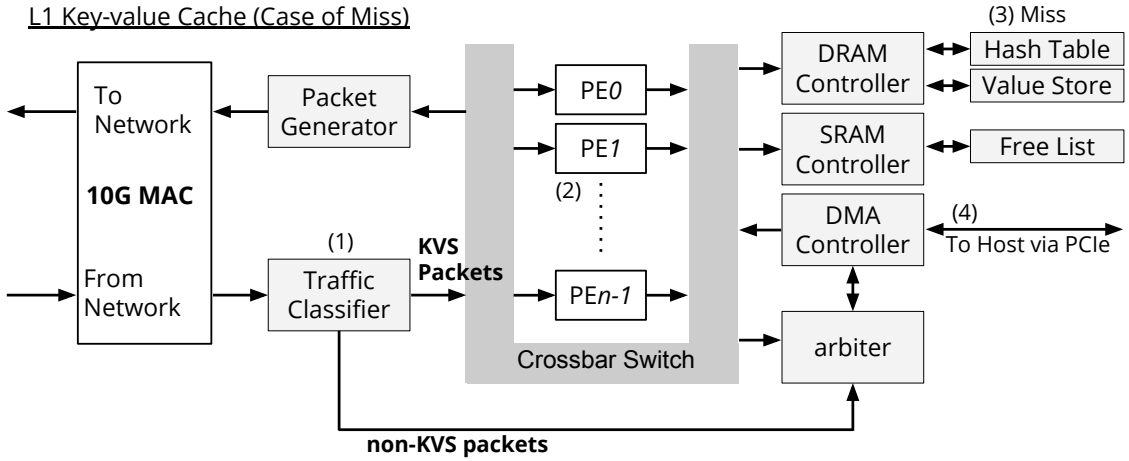


Figure 3.3: L1 key-value cache miss on heterogeneous multi-PE design of L1 key-value cache.

4B to 1MB) is observed in a memcached workload analysis [40]. In addition, a wide variety of value types, such as, string, list, hash, set, and sorted set [41], are useful for practical applications. Their processing cycles differ significantly depending on their value types. For example, the computational complexity of processing a string-type value is $O(1)$, while that for a list structure it is $O(n)$, where n is the number of items in the list. Thus, an optimized PE core is built for each value type, rather than a single deep pipeline where various key-value pairs are uniformly processed. While these optimized PE cores are simple and are not pipelined, the proposed design employs multiple instances of such simple cores to exploit a high query-level parallelism.

Below are value-types supported in our L1 key-value cache. They are the same as those supported in data structure server Redis [41].

- STRING: A variable-length value is stored as a string into a dynamically allocated free mem-

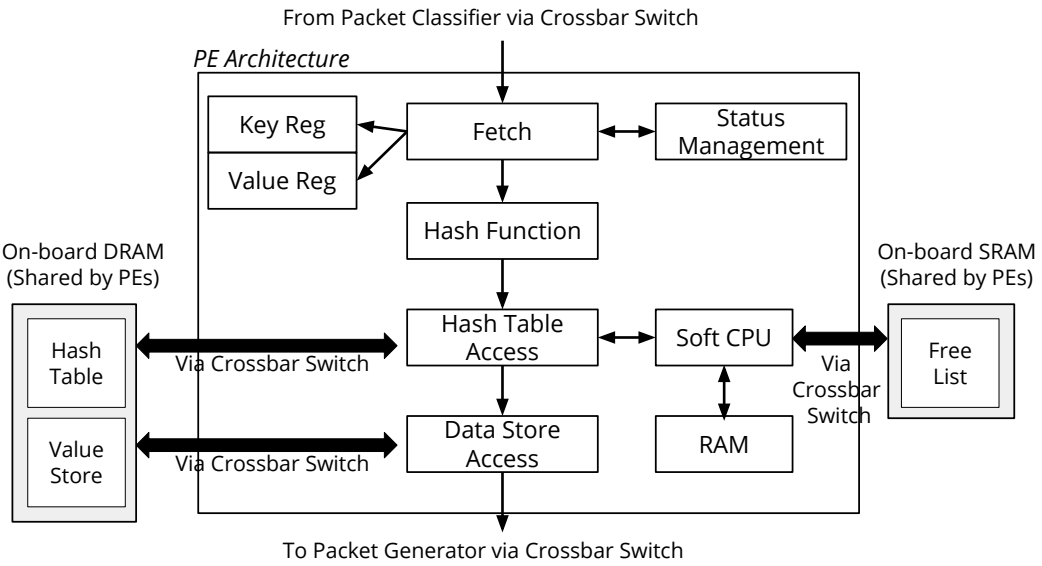


Figure 3.4: key-value store PE core architecture.

ory block, called chunk.

- HASH: A value consists of multiple pairs of field and its value, both of which are STRING type. These strings and their hashed values are stored as a table into a free chunk.

- SET: A value consists of a collection of unsorted strings. The collection of strings are stored in a free chunk. Each string is aligned in a fixed-size block in a chunk.

- LIST: A value consists of a list of strings which are stored in a free chunk as well as those of SET type. The proposed L1 key-value cache supports the list operations that insert and append a new element to the list.

- SORTED SET: It is similar to SET type, but each string has its own score and strings are sorted based on their score.

As shown in Figures 3.2 and 3.3, PE cores (mentioned above) are connected to a crossbar switch. PE Affinity module is in charge of packet classification. It receives packets from Ethernet MAC (Media Access Control) and checks their destination port number. If the destination port number of a received packet is matched to one of key-value service port numbers, PE Affinity module passes the packet to one of PEs which are currently idle. A DRAM controller is also connected to the crossbar switch. It is accessed by the PE cores to read and write access hash table and value store in DRAM modules on the FPGA NIC board.

3.3.1 Processing Element (PE) Design

A prototype of an L1 key-value cache that consists of key-value store PE cores of the string type is implemented. The PEs process SET and GET operations. The proposed design accepts variable-length values, while the key length is fixed to 64B for simplicity. CRC32 is implemented as a hash function. PEs and a DRAM controller are connected via a crossbar switch. A simple fixed-priority arbiter is used for the crossbar switch. Data width of the crossbar is 128bit. UDP is employed as a transport-layer protocol as it is simple and low-overhead. UDP is supported in memcached in addition to TCP.

Figure 3.4 shows a block diagram of a string type key-value cache PE. Received key-value store queries are passed to Fetch module and they are parsed as operation type (e.g., SET, GET, and DELETE), key, and value. PacketFilter core extracts key-value packets from all the received packets. More specifically, packets with specific destination IP address and destination port number are marked as key-value packets and transferred to the L1 key-value cache for key-value processing. The other packets are simply transferred to a host machine via the DMA controller. The key-value queries are first stored in a FIFO buffer and processed by one of multiple PE cores. In the GET operation, the requested key is examined in the L1 key-value cache and if the requested key-value pair does not exist in the cache, the request packet is passed to the crossbar switch and then transferred to the host machine. If the requested key-value pair exists in the L1 key-value cache, the response packet is generated by swapping the source/destination IP addresses and source/destination port numbers of the original request packet and adding the requested value.

3.3.2 Memory Management

Memory management is in charge of allocating free memory chunks and freeing unused ones. For example, a write request (e.g., SET operation) needs to allocate a free memory chunk in value store to store the new value. Slab Allocator is employed for memory management in value store. Slab Allocator manages fixed-length memory chunks of several sizes where values are stored, as shown in Figure 3.5.

When it receives a SET query that stores a new value, an unused chuck with minimum size where the new value can be fit is removed from the Free List and then used. Assuming, for example, a SET query contains a 24B new value, Slab Allocator in Figure 3.5 allocates a 64B chunk to store the 24B new value. Such a memory management is one of complicated functions when it is implemented as a dedicated hardware. In the proposed L1 key-value cache, Slab Allocator is implemented as a microcode running on a tiny soft-CPU processor in each PE core to access Free List of each chunk size as shown in Table 3.1. The other options are using hard-macro CPU (e.g., Xilinx Zynq series) and dedicated hardware implemeted by HDL. Since hard-macro CPU runs on further faster speed, the performance of slab allocation can be improved. Chapter 4 shows further implementation for higher performance and uses dedicated hardware for slab allocation.

Table 3.1: Soft-macro CPU specification for slab allocator

CPU	MIPS R3000 Compatible
Instruction and Data Width	32bit
RAM	32kB (cache-less)
Frequency	80MHz

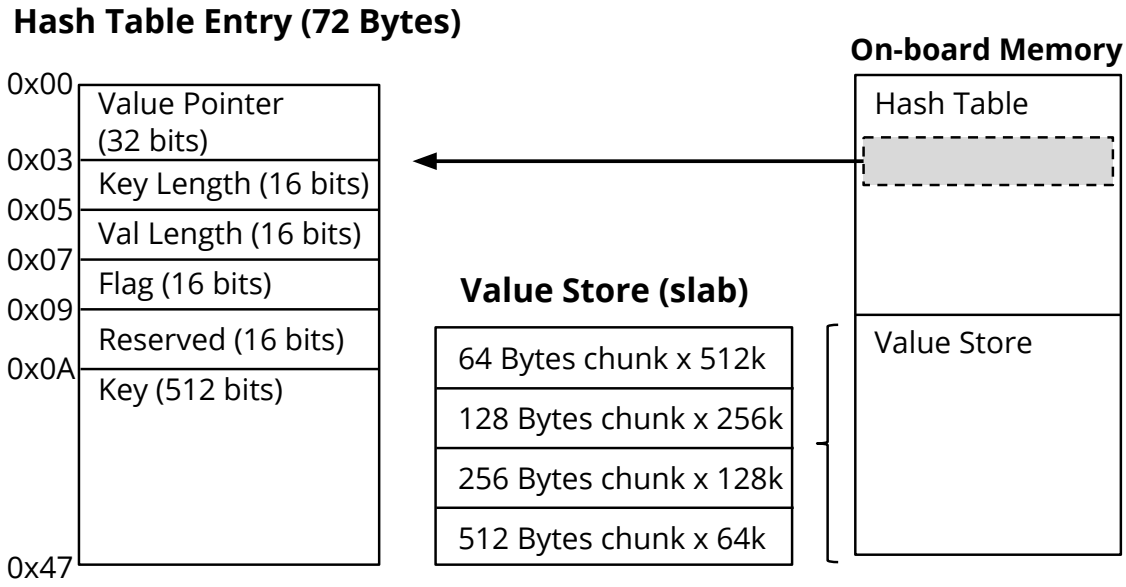


Figure 3.5: Hash Table and Value Store implemented on DRAM.

3.3.3 Hash Collision Handling

Figures 3.2 and 3.3 also illustrate its behavior of L1 key-value cache in the two cases where a requested key is hit and missed on the hash table, respectively. hash table is used to store pairs of a key and a start address of the chuck where the corresponding value is stored. A hashed value of a key is used as an index (address) of the hash table. To read or write the chunks stored in value store, a PE core performs the following steps.

1. An index in the hash table is calculated by hashing the requested key.
2. Content from the hash table is read based on the index. Then the requested key and the key read from the hash table are compared.
3. If both the keys are identical, a value is read from the Value Store based on the start address of the chunk (Figure 3.2). Otherwise, the requested key does not exist in the value store (Figure 3.3).

L1 key-value cache miss occurs when the hashed value of the requested key does not exist or it is conflicted with that of the other keys. To mitigate the hash conflicts, set-associative design was introduced in Section 3.2.2.

Various hardware-level NIC extensions, such as TCP offloading, checksum calculation, and Receive Side Scaling (RSS) supported in some commercial network adapters, are available for improving the network performance. Please note that the proposed L1 key-value cache is orthogonal to such hardware-level NIC extensions and can be used with them for further efficiency.

3.3.4 Interconnection

In this system, each PE is connected with the crossbar switch to accelerate the performance by parallelizing cores. Table 3.2 shows the specification of the crossbar switch used in this L1 key-value cache.

Table 3.2: Crossbar switch specification

Data Width	128bit
Arbitration	Fixed priority
Frequecy	160MHz

Figure 3.6 shows the frequency and occupancy when only crossbar switch is synthesized on an FPGA (Xilinx Virtex-5 XC5VTX240T). It is assumed that each PE runs on 160MHz. The horizontal line indicates 160MHz as a target frequency. This figure shows that when the number of PEs connected with a crossbar switch is more than 30, the frequencies are below the target frequency. Slice occupancy is below 6% to satisfy the frequenecy requirement (160MHz). As shown in this figure, 30 PEs can be connected with a crossbar switch in terms of slice utilization and frequency. Please note that since this evaluation does not include PEs' area, the number of PEs implemented would be lower than expected.

3.4 L2 Key-value Cache Architecture

In [16], a key-value data store that uses a host memory as storage is implemented in Linux kernel space. Key-value queries received by the kernel are hooked so that a customized handler is called in order to process the key-value queries inside the kernel. Such in-kernel processing can improve the key-value performance compared to the original user space implementation, because the network protocol processing and related system calls can be eliminated. As a result, about 3.3Mops (operation per second) performance in USR trace is achieved as reported in [16]. In the proposed multi-layer key-value cache architecture, as an L2 key-value cache, a similar in-kernel cache is implemented in Linux kernel using Netfilter framework.

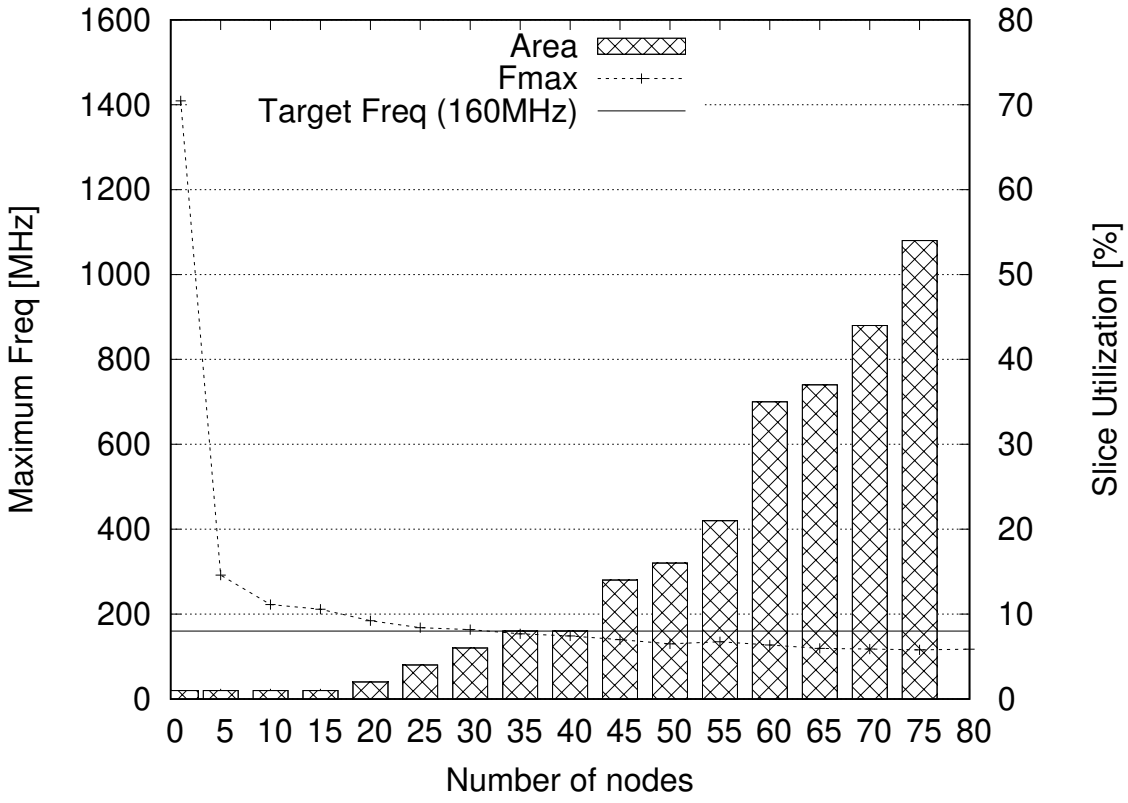


Figure 3.6: Maximum operating frequency and slice utilization vs. number of PEs connected to crossbar switch.

3.5 System Implementation

In this section, a prototype implementation of the proposed multi-layer key-value cache is illustrated. Only the necessary functions of the L1 key-value cache are implemented to explore the design choices. More specifically, multiple heterogeneous key-value cache PEs introduced in Section 3.3 are implemented in the L1 key-value cache.

3.5.1 Design Environment

Table 3.3 lists the design environment. The proposed L1 key-value cache is implemented on NetFPGA-10G board by partially using the reference NIC design provided by NetFPGA Project [9]. Table 3.4 shows the hardware specification. FPGA device used is Xilinx Virtex-5 XC5VTX240T. A 10GbE network interface is used for communication. hash table and value store in the L1 key-value cache are implemented on a 288MB RLDRAM-II memory on the FPGA board [1]. Design tool used is Xilinx ISE 13.4.

[1] The memory capacity of NetFPGA-10G board is very small, but newer FPGA boards have more capacity (e.g., 8GB DDR3 SDRAM for NetFPGA-SUME).

Table 3.3: L1 key-value cache design environment.

CPU	Intel(R) Core(TM) i5-4460
Host memory	4GB
OS	CentOS release 6.7
Kernel	Linux kernel 2.6.32-504
NIC (FPGA)	NetFPGA-10G

Table 3.4: Target FPGA board for L1 key-value cache.

Board	NetFPGA-10G
FPGA	Virtex-5 XC5VTX240T
DRAM	288MB RLDRAM-II
SRAM	27MB QDRII SRAM
PCIe	PCIe Gen2 x8
Network I/O	SFP+ x4

3.5.2 Implementation of L1 and L2 Key-value Caches

Slab Allocator is implemented as a microcode running on a MIPS R3000 compatible soft processor. Free List is a list structure that manages unused memory chunks. It is initially implemented as Block RAMs in the FPGA for simplicity, but it can be implemented using on-board SRAMs. When the key-value cache PE executes a SET operation that requires a free memory chunk, it interrupts the soft processor so that the Slab Allocator returns an unused chunk from Free List. In the case of a DELETE operation, Slab Allocator seeks the corresponding chunk and appends it to Free List.

As L2 key-value cache, an in-kernel key-value cache is implemented as a loadable kernel module. Netfilter framework is used to process key-value store packets in the kernel. That is, a customized handler is called when the kernel receives the key-value queries, as illustrated in Section 3.4.

3.5.3 Area Evaluation

This section evaluates the FPGA area utilization of key-value cache PEs, which are used in L1 key-value cache. Horizontal scalability that allows us to add more PEs depending on required performance is an advantage of our heterogenous multi-PE design, where each PE is optimized for specific data types as illustrated in Section 3.3. The target device is Xilinx Virtex-5 XC5VTX240T on NetFPGA-10G. Xilinx ISE 14.7 is used for design synthesis and implementation. SPEED mode is selected as a synthesis option.

Figure 3.7 shows the slice utilization. “Reference NIC” shows the slice utilization for the stan-

Table 3.5: L1 key-value cache throughput for four query types.

Query type	Average throughput [Mops]
GET (HIT)	1.42354
GET (MISS)	Determined by L2 key-value cache
SET (HIT)	2.20104
SET (MISS)	0.71049

dard 10GbE NIC functions that include four 10G MAC cores and a DMA controller for PCI-Express Gen2 x8. "Key-value cache PE + Reference NIC" shows that for up to eleven PEs and an RLDRAM/DDR3 SDRAM controller in addition to Reference NIC. Each key-value cache PE has a MIPS R3000 compatible processor and registers for storing key-value pairs. Up to eleven PEs can be implemented on the Virtex-5 device.

3.5.4 Throughput

This section evaluates the query processing throughput of L1 key-value cache. On the client machine, netmap-based [37] query injector that can fully utilize 10GbE bandwidth is used to generate queries. The following four query types are used for the throughput evaluation. Each type has a key and a value. Their lengths are fixed to 64B.

- "SET (HIT)" generates SET queries which are always hit in L1 key-value cache and modify the cache.
- "SET (MISS)" generates SET queries which are always missed in L1 key-value cache [2]. In this case, memory allocation is performed for each query in L1 key-value cache and thus the performance will be degraded.
- "GET (HIT)" generates GET queries which are always hit in L1 key-value cache by caching the keys to be requested in L1 key-value beforehand. The key-value cache PE returns response packets that contain the key-value pair requested in the GET queries to the client.
- "GET (MISS)" generates GET queries that never hit in L1 key-value cache. Such queries are transferred to L2 key-value cache and processed. Thus, its throughput is determined by that of L2 key-value cache rather than L1 key-value cache.

Table 3.5 shows average throughputs of the four query types on L1 key-value cache. Throughput of "GET (HIT)" is 1.42Mops. Throughput of "SET (HIT)" is 2.20Mops, while that of "SET (MISS)" is 0.71Mops because memory allocation on value store is performed for each query. In "SET (MISS)"

[2] We modified the FPGA logic of L1 key-value cache not to cache anything.

case, a soft processor running on the PE executes a memory allocation which takes about 200 clock cycles. The PE is stalled during the memory allocation, and thus the throughput is degraded.

The expected throughputs when assuming multiple PEs can be calculated. NetFPGA-10G board is equipped with two RLDRAM-II modules and their aggregate throughput is 25.6Gbps[3]. Thus, the memory bandwidth including the arbitration for the DRAM controller is not a major performance bottleneck when a given workload is under 25.6Gbps. An expected aggregated throughput is calculated as

$$T_{Total} = min(T_{Mem}, T_{Net}, n \times T_{PE}), \tag{3.1}$$

where T_{Mem}, T_{Net}, T_{PE}, and n denote the memory bandwidth, network bandwidth, single PE performance, and the number of PEs, respectively. T_{Mem} and T_{Net} are set to 25.6Gbps and 10Gbps, respectively. T_{PE} is set based on Table 3.5. Figure 3.8 shows the calculation result. It shows that 10Gbps network bandwidth is a major source of the performance bottleneck on L1 key-value cache. It also shows that seven PEs and nine PEs are required to achieve the 10Gbps GET and SET throughput, respectively.

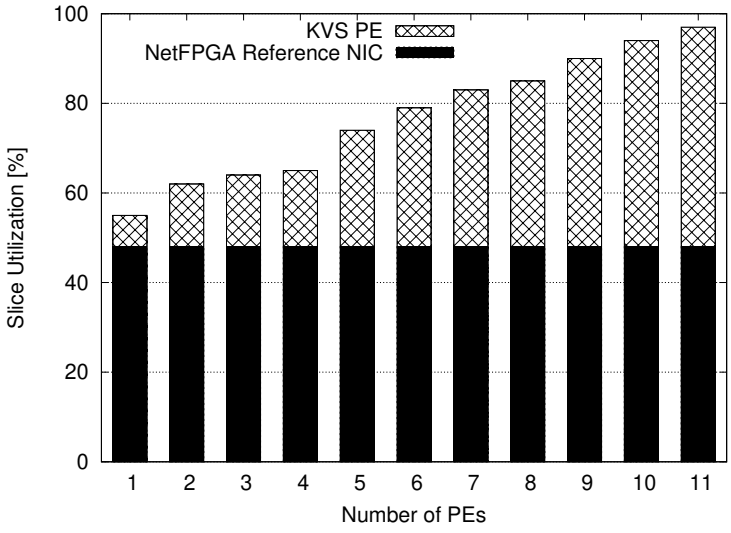


Figure 3.7: Area utilization on Virtex-5 XC5VTX240T.

3.6 Design Exploration on Key-value Cache Architecture

Various design options are available for the proposed multi-layer key-value cache architecture in terms of the multilevel organizations and cache policies, as proposed in Section 3.1. This section will quantitively explore the proposed design options by using a simulator with real memcached traces so

[3]Since we assume that RLDRAM controller runs with 160MHz as frequency and 128bit data bus, the throughput is up to 25.6Gbps.

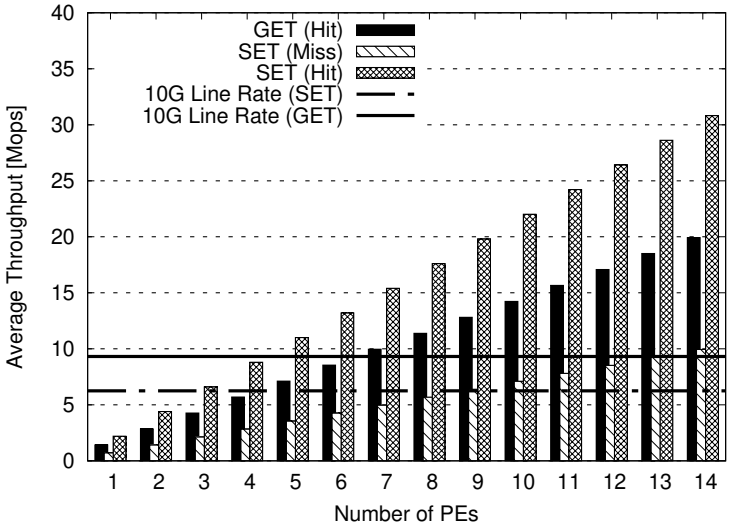


Figure 3.8: Aggregated throughput with multiple key-value cache PEs on NetFPGA-10G.

that this thesis would be the first guideline to build the multi-layer key-value cache architecture that utilizes the FPGA-based in-NIC cache and the in-kernel key-value cache.

3.6.1 Simulation Methodology

The memcached traces were generated based on a memcached workload analysis results [40]. In [40], the following five workload classes are analyzed in terms of operation types (e.g., GET, SET, and DELETE) ratio, key size distribution, value size distribution, key appearance, and so on. These three parameters were used to generate workload. To decide hit or miss in hash table, we referred only key appearance, i.e., identification number. Key length and value length were used for calculating throughput and slab allocation. Value contents were not utilized because the value contents are not considered in this simulation.

- USR : Key sizes are 16B and 21B. Value sizes are 2B.
- SYS : Most key sizes are less than 30B. 70% of value sizes are around 500B.
- APP : 90% of key sizes are 31B. Value sizes are around 270B.
- ETC : Most key sizes are 20-40B. Small amount of values are very large (e.g., 1MB).
- VAR : Key sizes are 32B. 80% of value sizes are 50B.

The above-mentioned values were measured by us from graphs in [40] by hand. Please note that these values may contain certain errors.

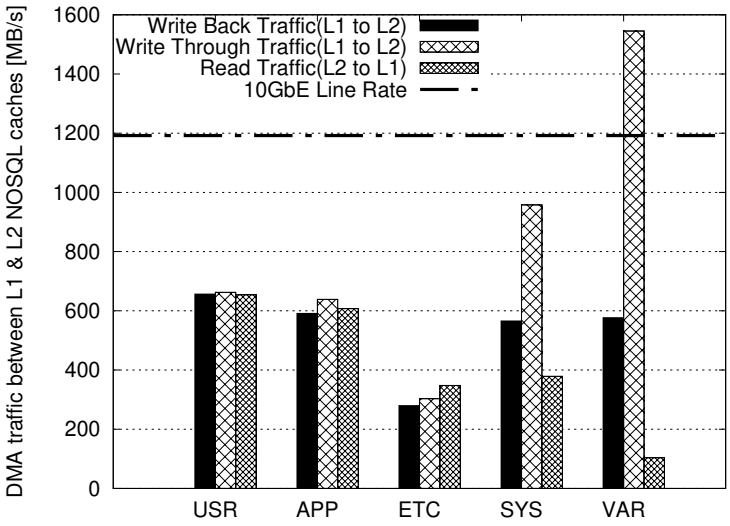


Figure 3.9: DMA traffic between L1 and L2 key-value caches (write-back vs. write-through).

3.6.2 Write-Through vs. Write-Back

There are two write policies between L1 and L2 key-value caches: write-through and write-back. Here the DMA controller between L1 and L2 key-value caches is simulated in addition to these caches. In the simulation, 10Gbps line rate queries based on the memcached traces are injected to L1 and L2 key-value caches.

Figure 3.9 shows the simulation result. In VAR and SYS traces, there is a large gap between the write-back and write-through policies. In VAR trace, more than 70% of queries are SET operations that update the L1 key-value cache. SYS trace also contains a lot of SET operations (i.e., more than 30% of whole queries). Thus, write-through policy achieves a quite high throughput in VAR and SYS traces, while the differences between these two policies are not significant in the other traces.

The 10GbE and PCI-Express interfaces would limit the write throughput between these caches. Theoretical DMA transfer capacity is 4GB/s and 7.69GB/s in PCI-Express Gen2 x8 and PCI-Express Gen3 x8, respectively. A PCI-Express Gen3 x8 interface achieves 7.06GB/s throughput [42]. 10GbE is assumed as a network I/O. In the graph, these 10GbE and PCI-Express bandwidth values are shown as horizontal lines.

Although a performance gain of write-back may not be significant in the read-intensive workload, write-back policy can reduce the DMA traffic in write-intensive workload. When assuming a 10GbE network bandwidth, the DMA bandwidth between L1 and L2 key-value caches is a bottleneck when write-through policy is used for VAR and SYS traces.

3.6.3 Cache Associativity

The multi-layer key-value caches were simulated by varying the set associativity N of the L1 key-value cache, where N = 1, 2, 4, and 8. In all the cases, the total L1 key-value cache size is fixed to

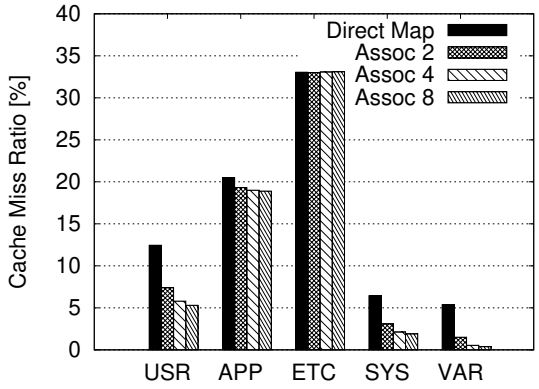


Figure 3.10: Cache miss ratio on set-associative L1 key-value cache.

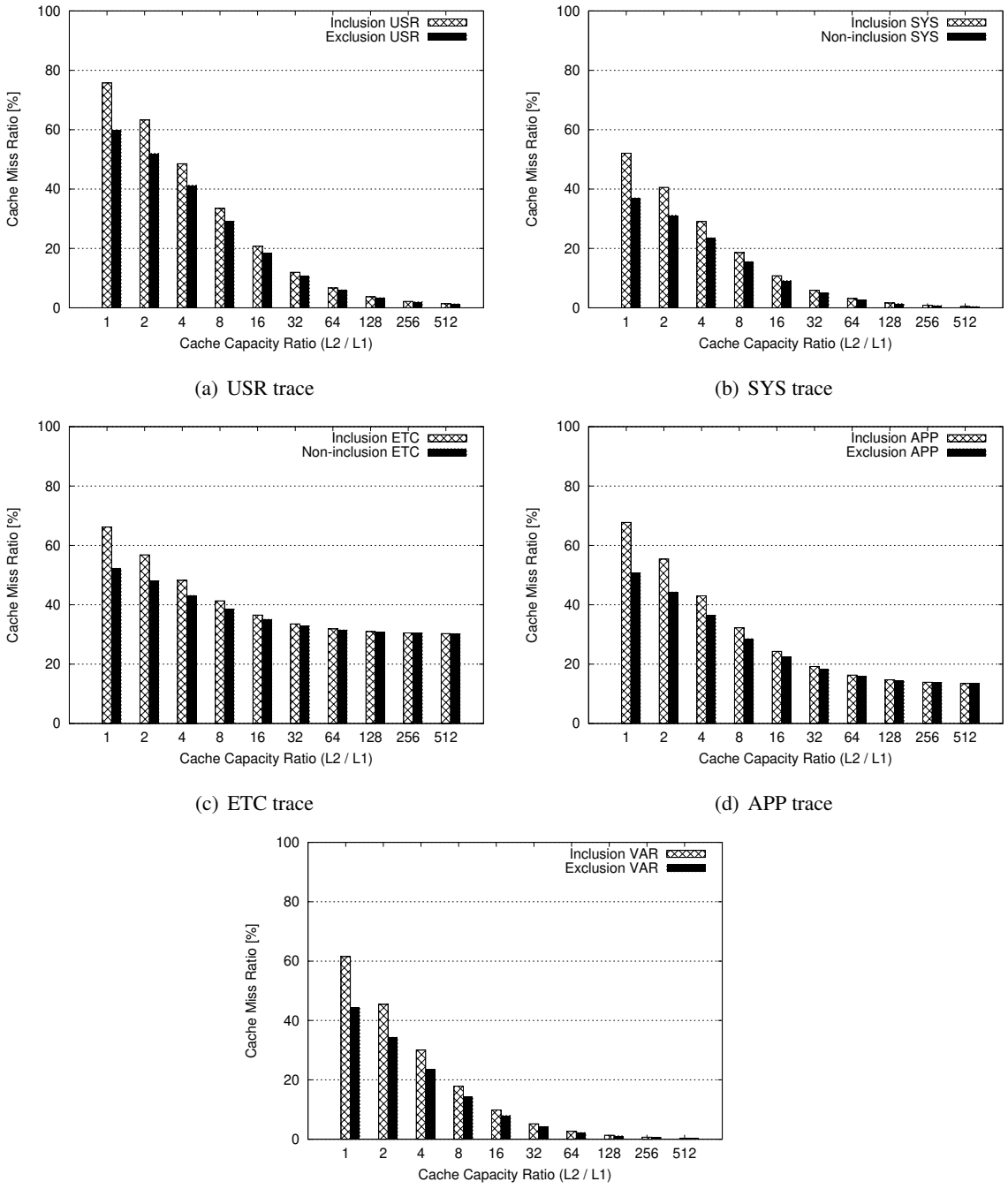
92MB.

Figure 3.10 shows the simulation results, where X-axis shows the memcached trace and Y-axis shows the L1 key-value cache miss ratio. In USR, SYS, and VAR traces, when N is varied from 2 to 8, the cache miss ratio is reduced by 2-7% compared to the direct mapped cache (i.e., $N = 1$). In ETC trace, the associativity does not affect the miss ratio due to less hash conflicts.

3.6.4 Inclusion vs. Non-inclusion

Here "L2/L1 capacity ratio" is defined as the ratio of the L2 key-value cache capacity against that of L1 key-value cache (e.g., the ratio is two when L2 is twice larger than L1). We will discuss inclusion and non-inclusion design options for L1 and L2 key-value caches when the L2/L1 capacity ratio is varied. Figure 3.11 shows the simulation results of inclusion and non-inclusion options, where X-axis shows the L2/L1 capacity ratio and Y-axis shows the cache miss ratio. When the capacity ratio is 1, the non-inclusion option reduces the cache miss ratio by up to 15% compared to the inclusion. Please note that inclusion option when the capacity ratio is 1 implicates the results with only L1 key-value cache (no L2 key-value cache). In this case, the cache miss ratio is quite high (e.g., 75.8% and 52.1% in USR and SYS, respectively), which demonstrates the necessity of our multi-layer key-value cache design. When the capacity ratio is over 16, differences between the inclusion and non-inclusion in terms of cache miss ratio become quite small. Assuming L1 key-value capacity is 288MB and 8GB, non-inclusion is an efficient option only when L2 key-value capacity is less than 4.6GB and 128GB, respectively.

DRAM capacity of the FPGA board used in the experiments is 8GB. When the capacity of a host main memory is less than 128GB, we can reduce the cache miss ratio by up to 10% by selecting non-inclusion option compared to the inclusion option. In other cases, the selection of inclusion and non-inclusion options does not affect the cache miss ratio significantly. Regarding the L1 key-value cache write policy, the inclusion policy assumes the write-through, while the non-inclusion policy assumes the write-back. non-inclusion can reduce DMA traffic in some traces. Although non-inclusion requires replacement of cache blocks, and so implementation cost is high. Thus in case



(a) USR trace

(b) SYS trace

(c) ETC trace

(d) APP trace

(e) VAR trace

Figure 3.11: Inclusive policy vs. non-inclusive policy.

of high bandwidth with PCI express, inclusion is recommended as a more straightforward design should be adopted when we consider implementation cost.

3.6.5 Eviction Policy

This section examines cache eviction policies in L1 key-value cache when the number of ways N is more than one. In the simulations, the capacity of L1 key-value cache is set to 2M entries, and eviction policies of Random and LRU are simulated. Figure 3.12 shows the simulation results of these eviction policies in terms of cache miss ratio for five traces when the number of ways N is varied from 2 to 8.

A visible improvement can be observed for each policy when N increases except for ETC trace which contains a lot of DELETE operations. The result of LRU is not better than that of RANDOM. Since hardware implementation of LRU is complicated, RANDOM is thus practical choice as the cache eviction policy for L1 key-value cache.

3.6.6 Slab Configurations for L1 key-value Cache

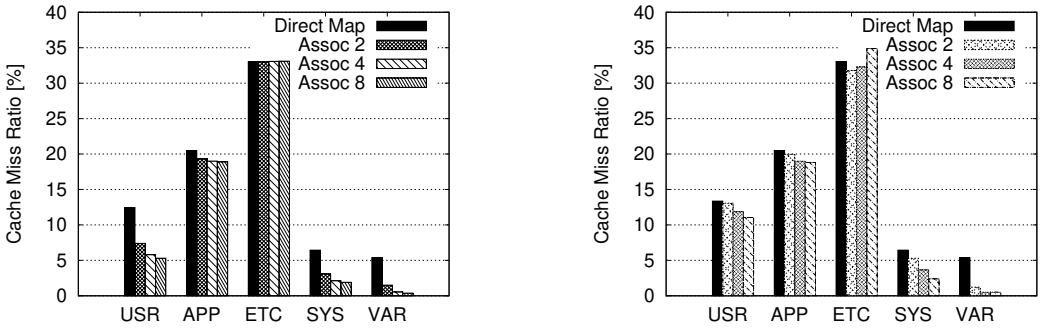
The proposed L1 key-value cache supports variable-length keys and values. Regarding the keys, since each hash table entry has a 64B memory space for key, keys larger than 64B are stored using multiple Hash Table entries. Since value sizes differ significantly, we employ Slab Allocator for the memory allocation. Up to six chunk sizes (a power of 2 from 64B) are supported by configuring the Slab Allocator. Table 3.6 shows various slab configurations used in this experiment. Each configuration type has a unique characteristic (e.g., uniform distribution, more small-sized slabs).

Figure 3.13 shows simulation results of these slab configurations. In USR, more than 90% of queries access small-sized values (e.g., 2B), so the slab configuration that has more small-sized chunks can reduce the cache miss ratio. In APP, the average value size is about 270B. When Type C configuration that does not have large-sized chunks is applied to APP, the cache miss ratio is more than 70% since most requested values cannot be cached in L1 key-value cache. In APP, we need 512B chunks to store most requested values; thus Type E that has many large-sized chunks can reduce the cache miss ratio to 40%.

The cache miss ratio can be improved by configuring the chunk sizes and their numbers in L1 key-value cache in response to value sizes in an expected workload. In the proposed design, soft-processor in L1 key-value cache can configure the chunk sizes and their numbers at a boot time.

Table 3.6: Chunk size configurations (In Type A, the number of 64B chunks is 70k).

Type	64B	128B	256B	512B	1024B	2048B	Note
A	70k	70k	70k	70k	70k	70k	Uniform
B	120k	100k	80k	60k	40k	20k	More small sizes
C	160k	140k	120k	0	0	0	No large sizes
D	140k	120k	100k	30k	20k	10k	More small sizes
E	20k	40k	60k	80k	100k	120k	More large sizes



(a) RANDOM policy

(b) LRU policy

Figure 3.12: Eviction policies on set-associative L1 key-value cache.

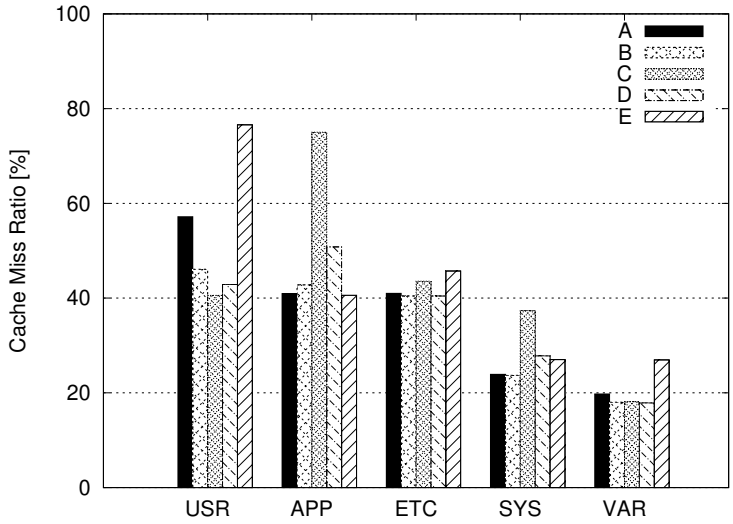
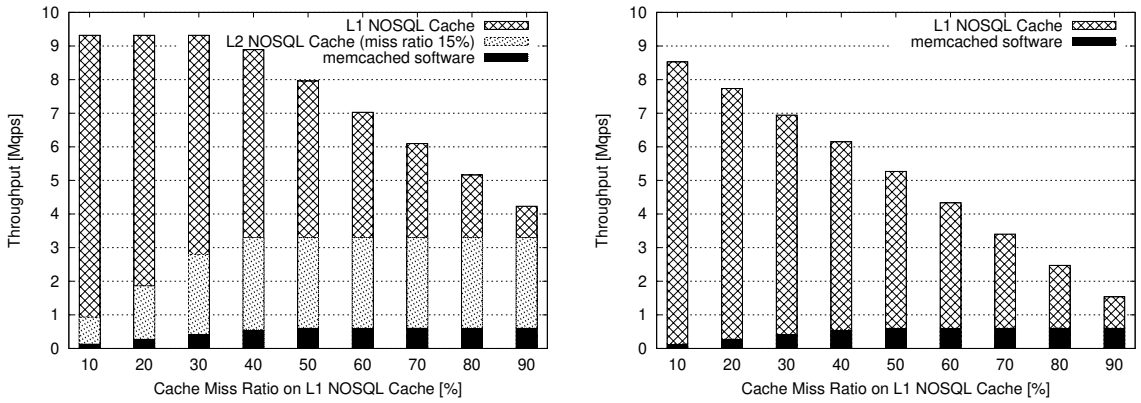


Figure 3.13: Five slab configurations on L1 key-value cache.

3.6.7 Cache Miss Ratio vs. Throughput

Finally, Figure 3.14 compares the proposed multi-layer key-value cache and L1 key-value only designs in terms of the total throughput when the L1 key-value cache miss ratio is varied from 10% to 90%. GET queries are injected in a 10Gbps line rate (i.e., 9.32Mops). In the multi-layer key-value cache case (Figure 3.14(a)), L2 key-value cache miss ratio is fixed at 15%, which is a conservative assumption. As L1 key-value cache miss ratio increases, L2 key-value cache and memcached software process more queries. Please note that the throughput decreases quite slowly in the multi-layer key-value cache case when L1 key-value cache miss ratio is less than 40%, while the throughput decreases linearly in the L1 key-value only case.



(a) Multi-layer key-value cache (L1 and L2)

(b) Only L1 key-value cache

Figure 3.14: Multi-layer key-value cache miss ratio vs. throughput.

3.7 Summary

This chapter introduced multi-layer key-value cache architecture combining in-NIC and in-kernel caches. Each component: in-NIC key-value store acceleration and in-kernel key-value store acceleration, has actively studied so far. But, each approach has disadvantages in terms of cache capacity and power efficiency. To complement these drawbacks, multi-layer key-value cache architecture was proposed to utilize them.

For the building block, there are a variety of design options to compose cache organization. Thus, we explore them with a simulator using realistic workload. This chapter introduced preferred design options, based on simulation results.

Chapter 4

Hierarchical In-NIC Key-value Cache Design

The previous chapter introduced concept based on simulation and basic proof of concept using a simple prototype. This chapter describes the practical implementation of in-NIC key-value cache design. This chapter focuses on inside in-NIC design and shows hierarchical in-NIC cache design combining on-chip RAM and on-board DRAM. As shown in Figure 4.1, this chapter introduces level 0 (L0) key-value cache, based on on-chip RAM to in-NIC cache of multi-layer key-value cache architecture, proposed in the previous chapter. Furthur, power efficient schemes are also introduced for scheduling software and hardware switching. Section 4.1 introduces the background behind this architecture. Section 4.2 provides a concrete architecture design of key-value cache integrated to NIC. Section 4.3 evaluates diffffrent design aspects. Section 4.4 provides design trade-off. Section 4.5 describes an on-demand controller. Lastly, Section 4.6 summarizes this chapter.

4.1 Background

The previous chapter introduced multi-layer key-value cache architecture. This chapter focuses on micro-level in-NIC design and introduces hierarchical in-NIC cache design for higher performance and an acceleration hardware as in-network computing.

In-network computing is an emerging area in computing, where applications natively running on the host are accelerated by running them on network devices. While hardware acceleration is typically done on stand-alone programmable platforms [10], in-network computing executes the applications on programmable network devices, such as network interface cards (NICs) or switches [43,44]. These network devices provide both the networking functionality and the execution of an application at the same time [45].

In-network computing has been shown to provide throughput and latency improvement of orders of magnitude [43,45]. Furthermore, the use cases are far from being limited to networking functions; examples include consensus [46], data processing [47], machine learning [48] and more. The most popular use case of in-network computing is for cache-based applications (e.g., [43]). The placement of the in-network computing device within the network saves traversals of the network by-design

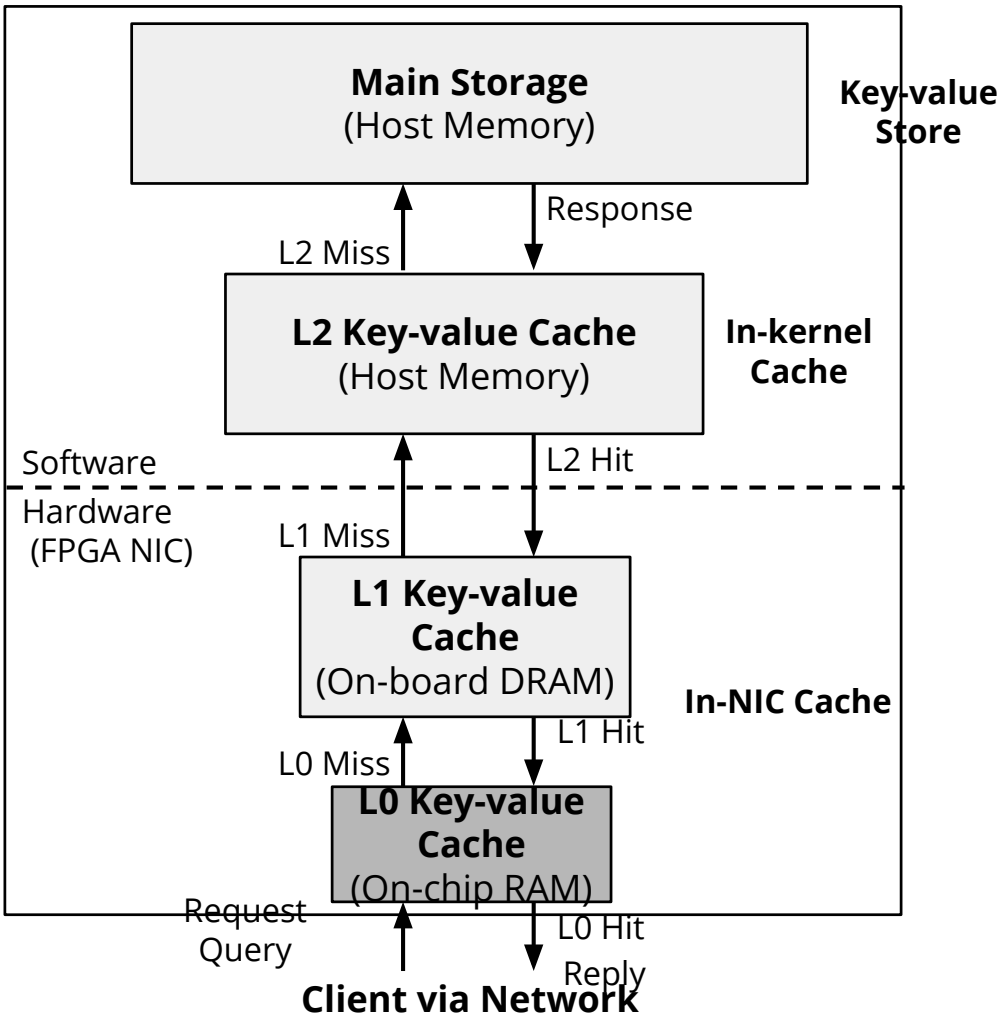


Figure 4.1: Chapter 4 introduces L0 key-value cache, based on on-chip RAM.

[45], and is ideal for handling frequently repeated requests for information. This work focuses on one class of caching applications, the caching of key-value store, to study design trade-offs in in-network computing.

Online services such as e-commerce and social networks, mostly running in the cloud [5], are commonly using key-value store. Key-value store deployments in datacenters are often scaled-out in order to increase performance [6], which leads in turn to increased power consumption. One of the limitations of key-value store is that it is very sensitive to latency, in the order of tens of microseconds, end-to-end [49]. Using in-network computing has the potential to significantly improve the performance of key-value store based applications.

While in-network computing has attracted a lot of attention over the last few years, most of the work has focused on ASIC-driven implementations [43–45, 47]. The design trade-offs in building in-network computing platforms, and in particular those implemented using FPGAs, have to the best

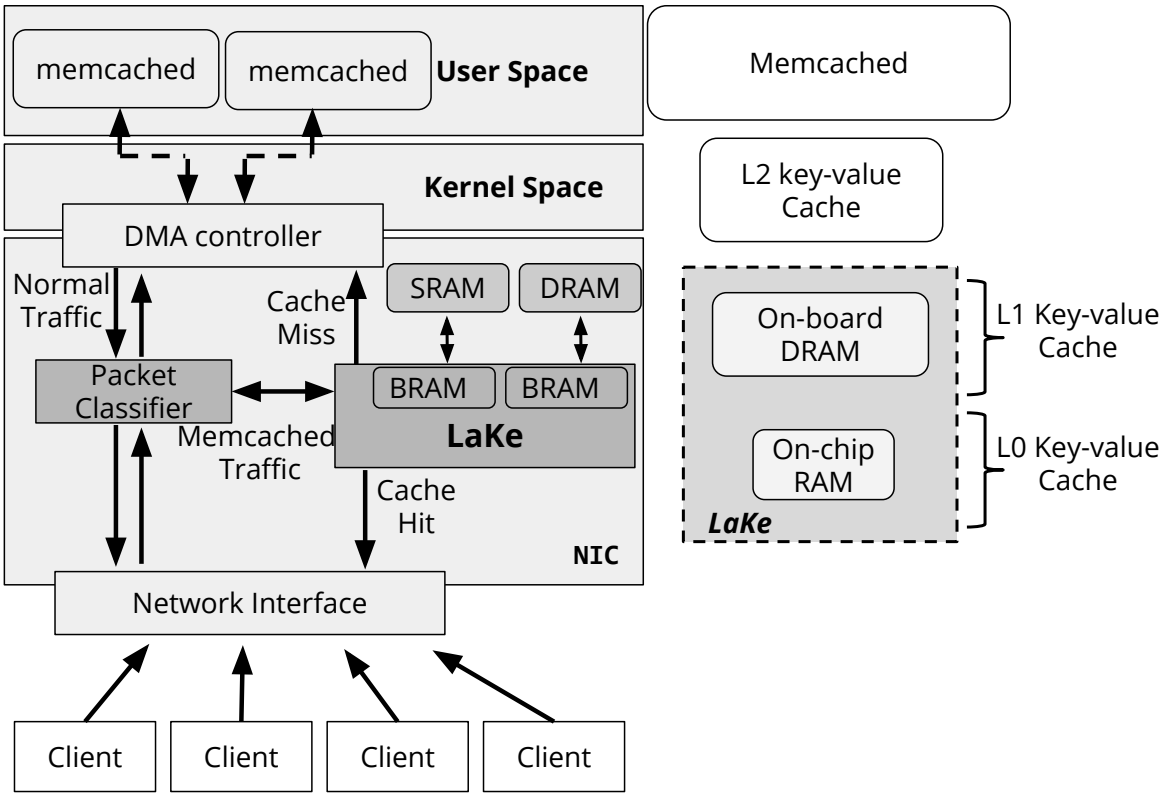


Figure 4.2: The high level architecture of LaKe.

of our knowledge, not been explored.

4.2 Architecture

Single-node memcached servers have been shown in the past to process queries at around 370kqps (query per second) on an Intel Xeon machine [19], with more modern servers achieving close to a million queries per second. Using consistent hashing algorithms has been shown to improve this throughput by an order of magnitude [50]. Large services, such as e-commerce or social networking services, therefore use tens to thousands of data center servers to sustain the query rate they require.

LaKe is an in-network computing **La**yered **Ke**y-value store architecture, focused on memcached with two layers: on-chip RAM and on-board DRAM. LaKe is FPGA-based and provides both network-switching functionality and key-value store acceleration. It achieves significant performance improvement by using multiple layers of cache. Each cache layer provides a trade-off between performance (latency, throughput) and memory size. LaKe runs on a platform that also acts, at the same time, as a NIC or a switch, therefore eliminating the need the cost of adding additional hardware. The performance achieved by LaKe reduces by an order of magnitude the number of servers required in the data center. This section explores its architecture, as shown in Figure 4.2.

L1 key-value cache architecture was shown in Section 3.3. In this chapter, to provide higher efficiency, hierarchy design with micro-level in-NIC cache is built: on-chip RAM and on-board off-chip DRAM as shown in Figure 4.1. Further, to optimize proof of concept, some parts of the design were upgraded, compared with the previous design in Section 3.3.

4.2.1 High Level Architecture

The LaKe architecture, a hardware component utilized with software components. It assumes that the software components are L2 key-value cache introduced in the previous section and the memcached host software, modified to support UDP binary protocol. Please note that we use only memcached host application as a software component in this section. The hardware component, which is the focus of this section, is a combined design of a networking device and a memcached accelerator running on a single platform.

The architecture of LaKe is shown in Figure 4.2. While LaKe can operate either as a switch or a network interface card (NIC), let us assume for clarity that it is used as a NIC. Traffic arrives to LaKe from multiple sources. A packet classifier is used to distinguish between memcached queries and any other type of traffic; general traffic will be sent to the host, as in a standard NIC, while memcached queries will be sent to the LaKe module. Queries that are a miss in LaKe's cache and memory, are sent to the host.

LaKe is implemented on the NetFPGA-SUME platform [27]. The network data plane is based on the NetFPGA Reference Switch project, which can also operate as a NIC, and we amend it with logic enabling memcached operation, as shown in Figure 4.3. Modules unique to LaKe are marked in dark gray. Incoming traffic from multiple ports is fed into the data plane using an arbitration module (Input Arbiter). A packet classifier, unique to the proposed design, identifies the type of the packet and sends memcached packets to the LaKe module, described later in this section. Non-memcached traffic continues in the pipeline, where it is merged (using a second arbiter) with packets returning from the memcached module: both reply packets, going back to clients, and missed queries, forwarded to the host. The destination of the packet is set in an output port lookup module, and packets wait in an Output Queues module to their turn to be transmitted.

4.2.2 LaKe Module

To improve performance bottlenecks and enable scalability across different platforms, LaKe adopts a multi-core processor approach for query processing, similar to the design described in the previous chapter. The architecture of the LaKe module is shown in Figure 4.4.

Incoming queries are evenly spread between a set of processing elements (PEs), using a multiplexing and demultiplexing PE-network. Each PE receives and processes queries. Once a query is processed, the PE accesses a shared memory network. Three types of memories are connected to the memory network: DRAM, containing the hash table bucket and data store chunks (Section 4.2.3,

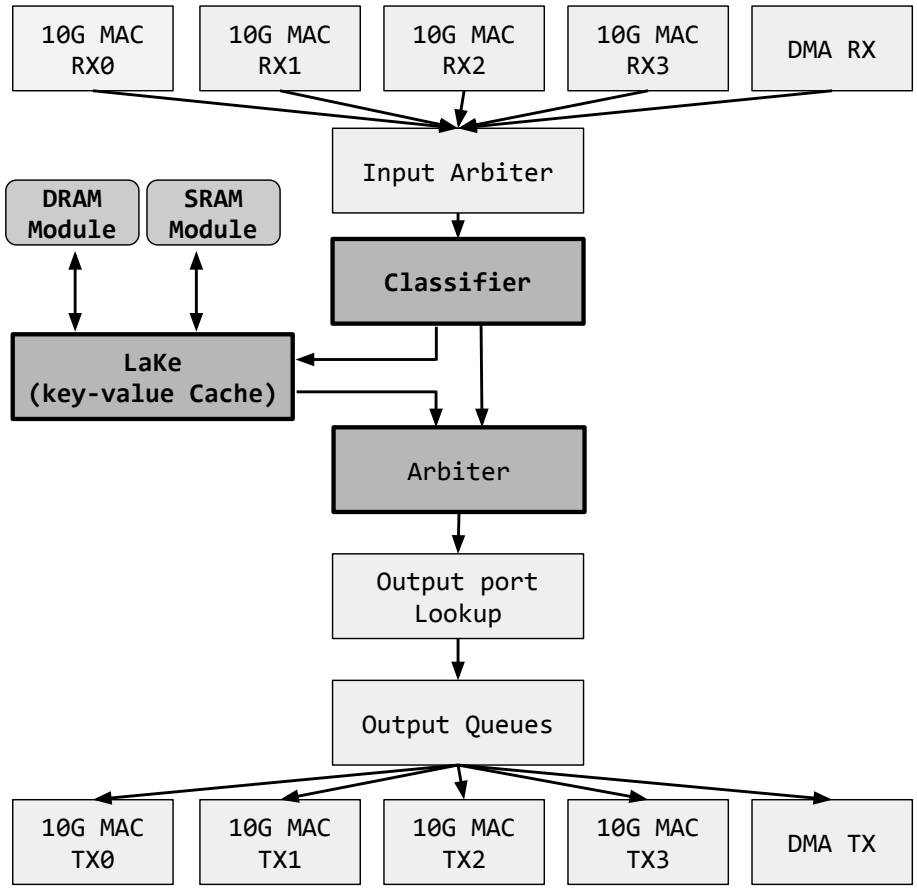


Figure 4.3: The block design of LaKe integrated with NetFPGA Reference Switch.

Section 4.2.4), SRAM, containing chunk information (Section 4.2.4), and CAM, serving as a look up table (LUT) for retrieving key-value pairs (Section 4.2.5).

Figure 4.5 illustrates the request-response process of a query in LaKe. As a new query arrives, the PE parses the packet and extracts the command, key and value. Next, the hash of the extracted key is calculated. In the proposed implementation, CRC32 is used as the hash function. The hash value serves as a pointer to an address in the DRAM, holding a descriptor (hash table bucket) pointing to the key-value pair in the memory. If a key exists in LaKe's memory, it is considered a *hit*, otherwise it is a *miss*. Upon a SET command that is a hit (Figure 4.5(a)), both the hash table and the key-value pair data are updated in the DRAM. If a SET command arrives with a new key (miss, Figure 4.5(b)), the PE assigns it to a chunk using a list of free descriptors stored in the SRAM and pointing to empty chunks. As a write-through policy is used, any SET query is also sent to the host's memcached server.

For a GET query that is a hit, a reply is prepared in the *Packet Deparser* and returned to the client. Otherwise, the request is forwarded to the host memcached server through the switch datapath (Figure 4.5(c)) and using a DMA engine [27]. The host then sends a reply to LaKe (Figure 4.5(d)),

which updates the key and value in the cache and DRAM before sending the reply to the user.

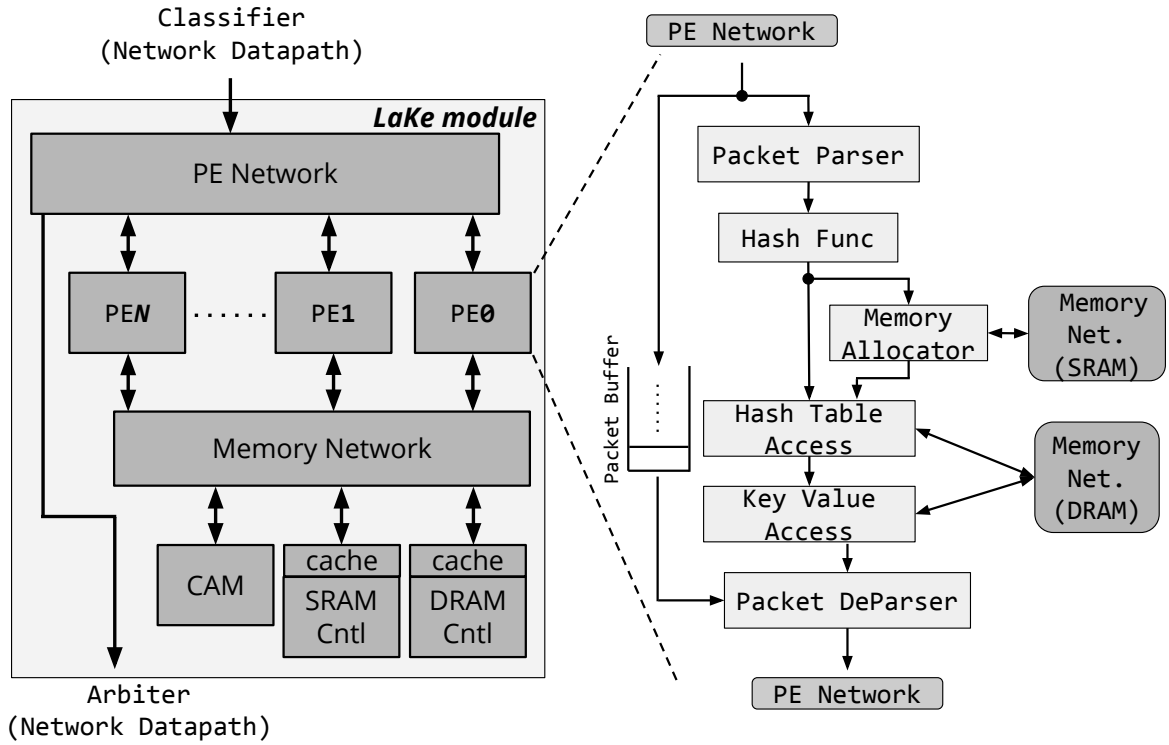


Figure 4.4: LaKe module architecture. The architecture of each PE is shown on the right.

4.2.3 Hash Table

The hash table is used to store descriptors pointing from a hashed key to the address in memory of the actual key-value pair. As such, it is a critical component in the design. The data structure of the descriptors in the hash table is shown in Figure 4.6. The descriptor size is 64bit, which is performance optimized: the DDR3 uses a burst size of 8 and data bus width of 64bit, which leads in turn to a bus width from the DDR3 controller of 512bit. This allows in a single access to read eight descriptor entries, enabling 8-associativity. To reduce the number of accesses to the DRAM, a key's length is compared to the key's length in the descriptor, and only if they match the PE attempts to access the DRAM and read the key-value chunk.

In the design previously described in Section 3.3, a hash table entry includes a maximum 64 bytes key. When we use more than 64 bytes key, multiple entries are assumed to be used. However, this is complicated logic and leads to in-efficiency in hash table, since duplicated addresses in hash table become invalid. Thus, a new hash table entry does not include a key, moving a key to data store managed by slab allocator. As a result, in order to retrieve a key, we need two memory access: hash table, and data store.

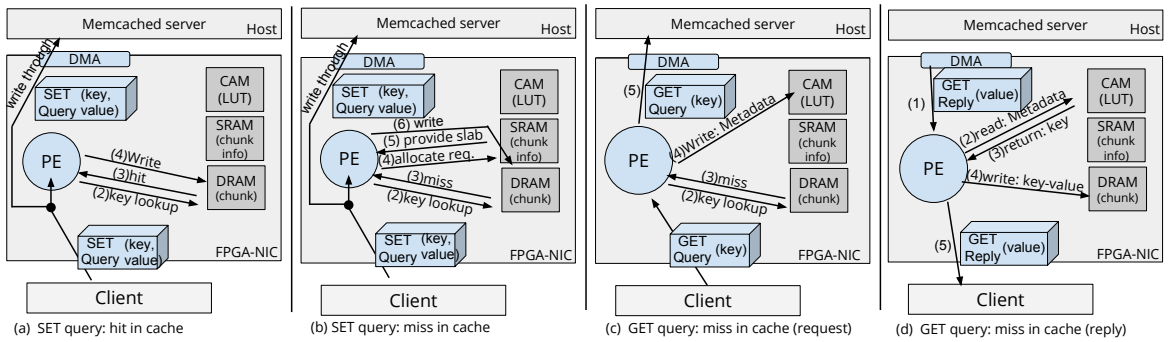


Figure 4.5: The request-response process of a query in LaKe.

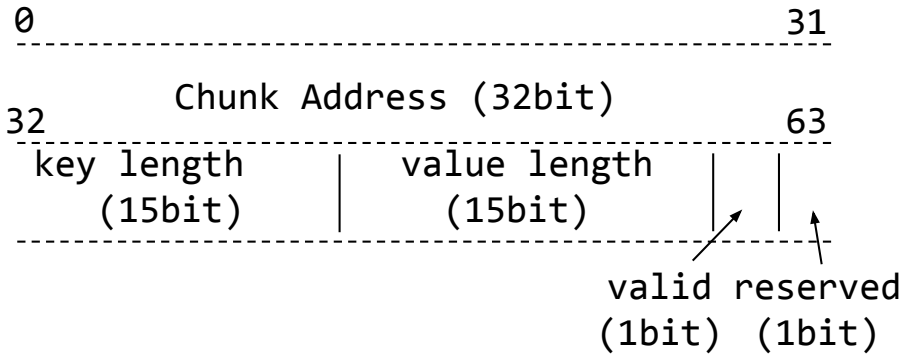


Figure 4.6: Hash table format. Fixed size entries are used.

4.2.4 Memory Management

Memcached builds upon a slab allocator to efficiently use the memory [6, 30]. This approach is also taken in hardware-based designs, as well as in LaKe, enabling to handle variable key- and value-length.

Similar to the design previously described in Section 3.3, a slab allocator is implemented using an SRAM-based memory, storing addresses of unused chunks. However, in this version. we do not use soft CPU for the processing slab allocation. To reduce access time to the SRAM, LaKe uses a small cache (implemented as a FIFO), which pre-loads the next available addresses from memory. The number of entries in the SRAM can be calculated using the following formula: $\sum_{k=i}^{n} S_k N_k \leq C_{mem}$, where S_k, N_k and C_{mem} denotes the size of chunk, the number of chunks and SRAM capacity, respectively. We use multiplications of 64B as slab size, and support 64B, 128B, 256B and 512B chunk sizes in our prototype. The minimum size slab is determined by the width of the memory network datapath: 512bit.

Shared cache To conceal DRAM access latency, LaKe uses a shared cache for each data context: hash table and data store. These caches are located in front of the DRAM controller, rather than inside

a PE, in order to enable PE scalability and avoid holding data context (such as CPU process) inside a PE. In this manner, frequent keys and values are referred from caches, immediately. Each cache has in our prototype 1024 entries, implemented using a BRAM. We employ write through as the update policy, thus cache coherency is maintained for both cache and DRAM.

DRAM access: Random access to the DRAM has a non-negligible and variable latency, which can stall PEs. To attend to this latency we integrate a small cache (e.g., 64B cache line, write through, direct-map, total capacity 64kB). Without the cache, we measure the DRAM controller's access latency (using Xilinx MIG, running at 933.33MHz) to be around 115ns in a zero load test, and to be up to 650ns under high load.

PE scalability: LaKe applies a modular, scalable approach to key-value store acceleration. The number of PEs supported by the design starts at one and scales up, with five PEs sufficient to support per-port full line rate. Beyond physically implementing a variable number of PEs, LaKe also allows to controlling on-the-fly the number of PEs used, balancing workload and power efficiency.

4.2.5 Memcache Protocol

Memcached systems [6] generally use the memcache protocol. There are two memcache protocol variants: ASCII based and binary based. Binary protocol over UDP/IP is used in the proposed cache system. The challenge using the memcache protocol is that key-value pairs cannot be identified in responses from the host. For example, a GET request missed in the hardware and sent to the host will have a query response returning with the value but without the key. Thus, cache systems cannot handle only response packets; it is required to learn and save request query's information.

To associate a key with a returning value, memcache protocol's opaque field and source UDP port number are used. Memcache protocol uses a 32-bit opaque field, and memcached systems use the same opaque value in both request and reply. A lookup module is used to match returned values from a host with their paired keys. The LUT is implemented using a CAM, where we query using the opaque value and the source UDP port, and the reply is the original query's key. The keys are updated every time a GET query is a miss in the hardware and forwarded to the host.

4.3 Evaluations

The evaluation of LaKe covers two aspects: absolute performance, and the exploration of design trade-offs. The evaluation results are summarized in Table 4.1.

4.3.1 Absolute Performance

The absolute performance of LaKe is evaluated, based on several performance metrics: throughput, latency and power efficiency. The performance is compared with memcached (v1.5.1), a software implementation, and Emu's memcached implementation [51], a hardware-acceleration of memcached

using the binary protocol. Emu is selected as it is comparable, being available as open-source on NetFPGA-SUME, yet it does not support networking functionality, only memcached-acceleration. Emu also supports only the on-chip cache, and cannot forward a missed query to a server.

4.3.1.1 Test Setup

The server uses Intel Core i7-4770 CPU, 64GB RAM, running Ubuntu 14.04 LTS (Linux kernel 3.19.0) and NetFPGA-SUME card. OSNT [52] is used for traffic injection. A 10GbE port is connected to LaKe-side card. GET requests including 4B key and 8B value are injected at 10Gbps. Throughput is measured on a second granularity. For comparison with software-based memcached, the memcached software was amended to support binary protocol over UDP.

4.3.1.2 Maximum Throughput

For maximum throughput, all three designs are compared using a warmed cache. LaKe achieves a throughput of 13.1Mqps (query per second) when all the queries are *hit* in the shared-cache, as shown in Table 4.1. This is ×6.7 improvement compared with Emu [51], and ×13.6 improvement compared with memcached running on the host. The throughput achieved is equivalent to 10GbE line rate, using the given query size, and requires only 5 PEs.

4.3.1.3 Latency

An Endace DAG card 10X2-S (4ns resolution) is used to measure queries' latency. A software-based client is used to generate queries, and the DAG measures the isolated latency of LaKe, client excluded. Despite supporting both memcached and networking functionality, as well as using the DRAM, LaKe's latency on a *hit* (1.16μs) is better than Emu (1.21μs), thanks to the small shared-cache (64kB) in front of the DRAM. When queries are *miss* in the shared-cache, and *hit* in the DRAM, the latency is 5.6μs. Emu does not support cache misses. LaKe's latency is ×205 better than a host-based memcached on a hit, and ×42 better on a miss in the cache and a hit in the DRAM. A miss in both cache and DRAM means LaKe and a host-based memcached will have about the same latency, as LaKe will forward the query to the host. The only penalty is the first lookup in the DRAM of the key.

4.3.2 Scalability

LaKe scales up both in throughput and resources.

Area and Resources: Up to six PEs were implemented while maintaining 200MHz core frequency. Each PE utilizes around 3% of chip slices and 2% BRAMs, as shown in Figure 4.7. These values include also the interconnection networks, as each PE is connected with both PE-network and memory switch. The small overhead in resources taken by each PE enables scaling the number of PEs used by LaKe with little effect on resource consumption.

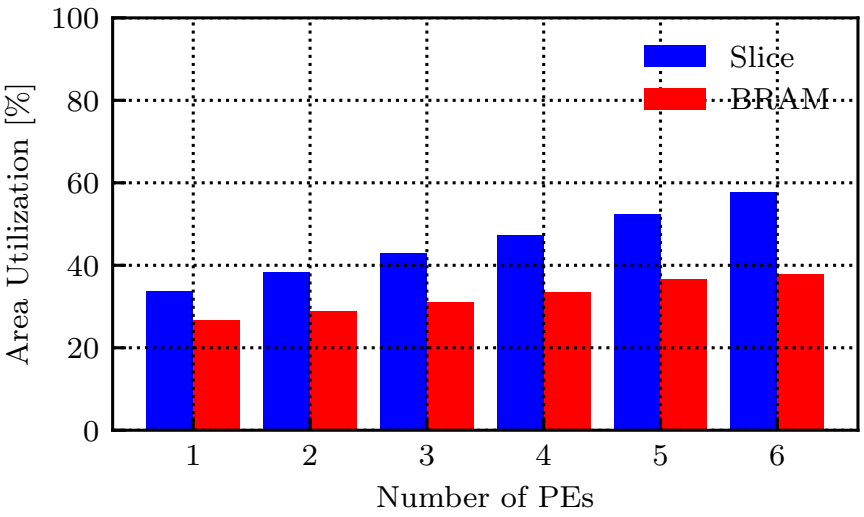


Figure 4.7: The area utilization of LaKe implemented on NetFPGA SUME.

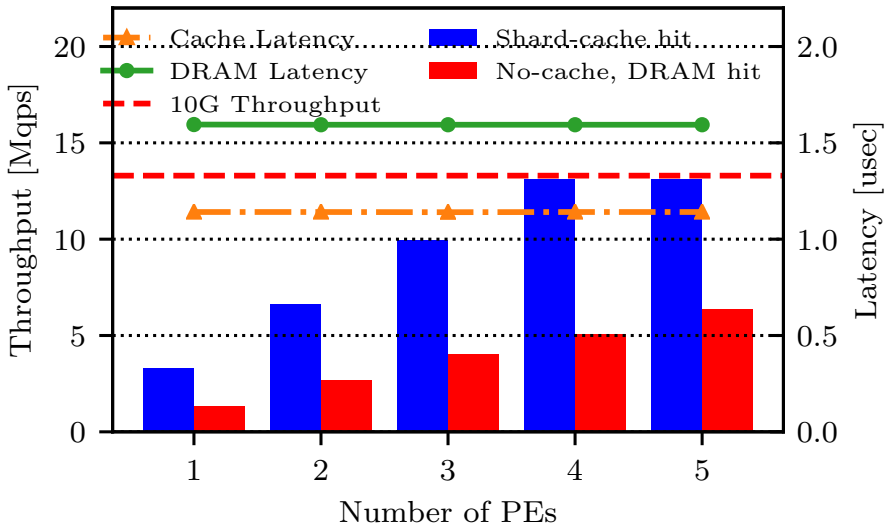


Figure 4.8: Throughput and latency as a function of number of PEs.

Throughput: The throughput scalability of LaKe is evaluated using OSNT [52]. First, the cache is warmed using a SET request. Next, OSNT generates GET requests, matching the warmed cache, using a 4B key, and returning an 8B value. The throughput scalability as a function of the number of PEs is shown in Figure 4.8. As the figure shows, LaKe can handle up to 13.1Mqps using five PEs, when the queries are *hit* in the shared-cache in front of the DRAM. Each PE processes up to 3.3Mqps. The bottleneck on throughput growth is the memory interconnect core and memory bandwidth. The throughput grows linearly with the number of PEs until reaching these bottlenecks. On a platform with more memory interfaces, or with a higher speed memory, a higher throughput can be achieved.

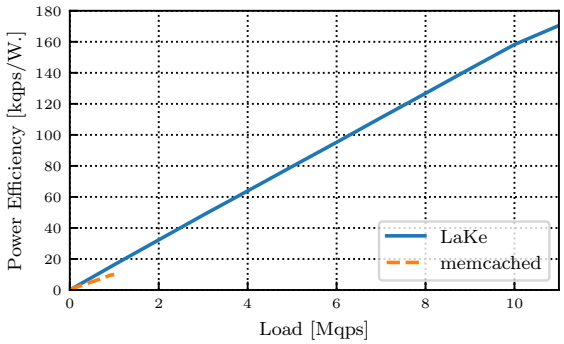


Figure 4.9: Power efficiency vs throughput, for LaKe and memcached (bottom left).

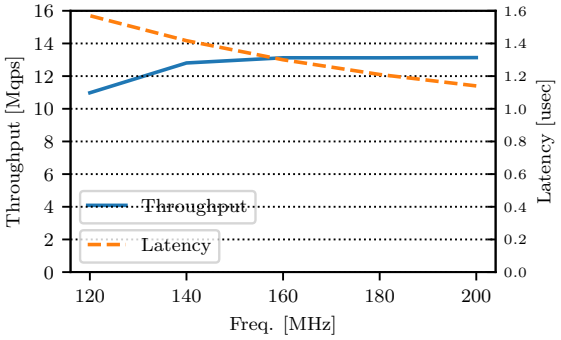


Figure 4.10: The throughput and latency of LaKe as a function of core frequency.

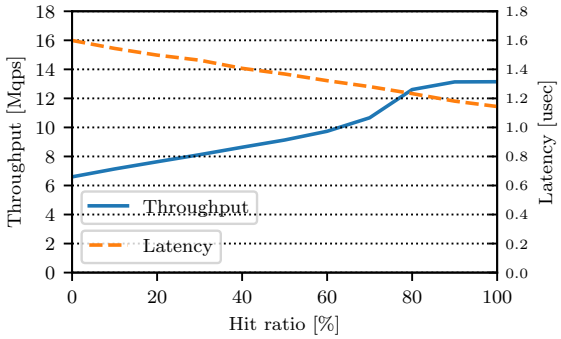


Figure 4.11: LaKe's throughput and latency under varying hit ratio in the on-chip cache. Queries missed in the cache are a hit in the DRAM.

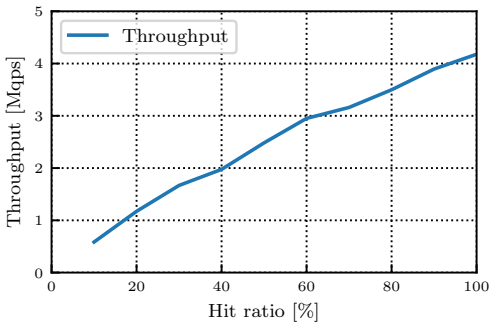


Figure 4.12: LaKe's throughput under varying hit ratio in the DRAM, with fixed 10% in the on-chip cache.

Core frequency: LaKe is a pipelined design. As such, its throughput depends on its packet processing rate. This packet processing rate, which is shared for the networking data plane and the LaKe memcached module, is fully achieved at a core frequency of 160MHz, as shown in Figure 4.10. Below this frequency, the NetFPGA platform has a performance limitation in its 10GbE ports [1]. Mean latency drops with core frequency increase: this is as the number of stages in the pipeline is maintained, but the duration of each clock cycle is reduced.

Hit ratio: The hit ratio in the cache plays a critical role in the performance on an in-network computing design. Figure 4.11 demonstrates the effect of the hit ratio on the performance of LaKe. The x-axis indicates the hit ratio of the keys in the on-chip cache. The y-axis indicates the maximum throughput (left) and mean latency (right). Mean latency is measured at a constant query rate of 10Kpps, for all hit ratios, since as shown in Section 4.4, the latency is subject to change under different query rates. The maximum latency, measured across all hit-ratios, is only 1.9μs. The effect of the hit ratio is mandatory to in-network computing solutions, as the size on the on-chip cache

[1] https://github.com/NetFPGA/NetFPGA-SUME-live/issues/36

Table 4.1: Performance comparison.

System	Average latency [μs]	Throughput [kqps]	Power efficiency [kqps/Watt.]
memcached(software)	238.84	962	9.938
Emu (hardware) [51]	1.21	1932	47.121
LaKe (shared-cache)	1.16	13120	242.962

directly affects the performance of the device. In devices where the memory capacity is in the order of megabytes to tens of megabytes [53], this becomes a crucial element.

Figure 4.12 continues the exploration of hit-ratio effect, by exploring the effect of hit ratio in the DRAM, and LaKe as a whole. The x-axis in Figure 4.12 indicates the overall hit ratio in both on-chip cache and DRAM. The hit ratio in the on-chip cache is fixed to 10%, and vary the hit ratio in the DRAM, with all queries missed in the DRAM being sent to the host. As the results show, throughput linearly increases with the hit ratio in the DRAM.

4.3.3 Power Efficiency

Next, we evaluate the power efficiency of LaKe. A wall power meter is used to measure power consumption. power efficiency is calculated as $E = T/W$, where E, T and W denote power efficiency, throughput, and power consumption, respectively. LaKe achieves 242.962 kqps/Watt using five PEs at full line rate. This is $\times$5.1 improvement compared with Emu.

Dynamic power consumption is investigated by measuring how power consumption varies as a function of throughput. Power consumption is normalized to 0W under zero load, i.e. the static power consumption. The dynamic power consumption takes up to 3.4W on LaKe modules, while software-based memcached consumes a maximum of 58.2W dynamically. Thus, LaKe reduces dynamic power consumption by more than an order of magnitude. Moreover, the power efficiency of LaKe scales linearly with throughput (as shown in Figure 4.9), much better than a host's power efficiency does. To put it in order words, LaKe's power consumption changes very little under load, which means that it is most efficient when the query rate is maximal. Even under low query rate, LaKe's power efficiency is better than running on a host.

4.4 Design Trade-Offs

The previous section has introduced the absolute performance of LaKe. This section focuses on trade-offs in the design of LaKe, and extrapolates from them to in-network computing designs at large.

In-network computing applications tend to implement cache using only on-chip memory [43, 45, 54]. For key-value store applications, this leads to a very small percentage of keys that can be cached: in the orders of thousands to tens of thousands on an FPGA, and in the order of hundreds

of thousands to a million on an ASIC. For example, NetChain [44] suggests that up to 10MB on a Tofino switch can be used as a cache. This number of cache entries is insufficient for large key-value store systems: in Facebook, between a billion and hundred billion unique keys are accessed every hour [40], with 18.4% to 74.7% of these keys accessed within 5 minutes (For Facebook's different workloads [40]). It is therefore important to understand the effect of using external memories on in-network computing performance.

So far the evaluation used a fully-featured LaKe: using BRAM, SRAM and DRAM. Next, we check the effect of each on the performance. Note that for this discussion the design employs a single DRAM module (4GB) which utilizes both hash table region (2GB: 268M entries) and a data store region (2GB: 33M entries as 64B chunk), and consumes 4W. When BRAM is used instead of DRAM, the number of hash table entries and data store entries are 4096 entries and 512 entries, respectively. Two SRAM modules (total 18MB) are also employed to manage free-list on slab allocation. When BRAM is used instead of SRAM, the number of free-list addresses stored is 144 entries.

When only the BRAM is used, and the SRAM and DRAM memory controllers are taken out, the maximum power consumption of LaKe is 16W including NetFPGA-SUME card — almost identical to a standalone switch, and the maximum throughput is 13.1Mqps. Under these circumstances we use a BRAM-based 1k entry cache as hash table and data store instead of a DRAM, and use BRAM-based FIFO as slab allocator instead of an SRAM.

Adding the SRAM adds 6W and holds 4.7M chunk addresses, which are updated when a DELETE operation moves a specific chunk to the free list. A BRAM-based FIFO placed in front of the SRAM is used to hide SRAM access latency, but is shallow in comparison with the SRAM. One can therefore trade the 6W SRAM power consumption with the number of available chunks on LaKe. Alternatively, one can use a DRAM to store chunk address: this solution is cheaper and more power efficient than using SRAM, but results in an increased latency and considerably lower throughput.

The use of DRAM as a first level cache increases the number of keys hit in LaKe. However, as can be expected, DRAM access does not provide the same performance as on-chip cache access. As shown in Figure 4.8, the maximum throughput using DRAM only is 6.3Mqps (using five PEs), lower than using the shared-cache. To understand the throughput of the DRAM, we isolate the DRAM from the LaKe module, and consider its latency under low and high utilization. As Figure 4.13 shows, while the latency is almost constant without a load, under high utilization the latency almost doubles. Memcached accesses to the memory are random and not sequential, as keys are not requested in a sorted order, this double-latency explains the 6.3Mqps throughput achieved using the DRAM.

While using the DRAM may seem as a disadvantage, it is in fact an advantage: access to the DRAM is only upon a miss in the cache, and replaces an access to the host memory (as in a device without a DRAM). In this manner, significant time (×42) and dynamic power (×17) are saved.

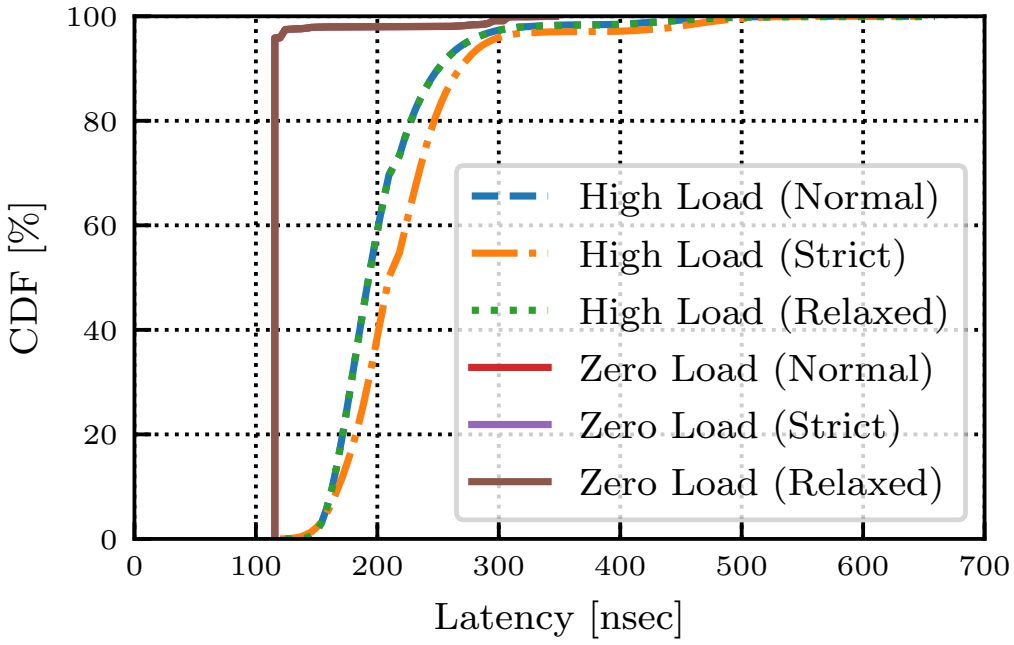


Figure 4.13: The cumulative distribution function of READ latency from the DRAM, for a Random access, under zero and high load. Strict, Normal and Relax are the three memory controller access modes [1].

4.5 On-demand Controller

This section discusses our LaKe, in-network computing device should be treated as one would treat other scheduled computing resources. Workloads can be assigned to network devices, and one should be able to reallocate the workloads to other computing resources. Section 4.5.1 provides an analysis describing when in-network computing can be optimally used, and next the how is discussed.. Section 4.5.2 proposes the on-demand scheme in terms of power consumption, CPU usage and traffic rate. Section 4.5.3 provides the details of host-controlled on-demand and network-controlled on-demand. Section 4.5.4 discusses the controllers.

4.5.1 Power and Performance Measurement

Here, we examine that in-NIC cache can be power hungry [55], by evaluating the power consumption of in-NIC cache and memcached. The power consumption is evaluated for both software- and hardware-based implementations, including overheads, e.g., power supply unit.

Experiment Setup In this experiment, an Intel Core i7-6770 4-cores server, running at 4GHz, equipped with 64GB RAM, 10GE Mellanox NIC (MCX311A-XCCT), and Ubuntu 14.04 LTS (Linux kernel 3.19.0) was used for software-based evaluation. For hardware-based evaluation, the NIC was

replaced by NetFPGA-SUME [27] card.

OSNT [52] was used to send traffic, in which the rates can be controlled at fine granularities and reproduce results. Average throughput was measured with a second granularity. An Endace DAG card 10X2-S was used to measure latency, measuring the isolated latency of the application under test, traffic source excluded. Power measurements were performed using SHW 3A power meter [56], connected to the device under test's power supply.

Results With LaKe, the role of the server software is not eliminated by shifting functionality to hardware. LaKe serves as a first and second level cache. In the event of cache misses at both levels, the software services the request. Memcached (v1.5.1) was used as both the host-side software replying to queries missed in LaKe's cache, and as the software implementation we benchmark against. The power consumption evaluation of LaKe, therefore, includes the combined power consumption of the NetFPGA board and the server. Note that the NIC is taken out of the server for LaKe's evaluation, as LaKe replaces it.

To measure the power consumption of LaKe and memcached, we start with the power consumption of an idle system, and then gradually increase the query rate until reaching peak performance. Peak performance is full line rate for LaKe and approximately 1Mpps for memcached. It is verified that the CPU reaches full utilization using all 4-cores.

Figure 4.14 shows the power to throughput trade-off for the key-value store. The x-axis indicates the number of queries sent to the server every second, while the y-axis indicates the power consumption of the server under such load. It shows the power consumption for memcached (software only), LaKe within a server, and LaKe as a standalone platform, i.e., working outside a server and without the power consumption contributed by the hosting server. As the figure shows, the power consumption of the server while idle or under low utilization is just 39W, while LaKe draws 59W even when idle. However, the picture changes quickly as query rate increases. At less than 100Kpps, LaKe is already more power efficient than the software-based key-value store, with the crossing point occurring around 80Kpps.

LaKe has a high base power consumption, but the consumption does not increase significantly under load. Figure 4.14 shows the throughput up to 2Mpps. But, please note that LaKe reached full line rate performance, supporting over 13Mpps for the same power consumption.

4.5.2 Scheduling Scheme

The experiments have shown that there is no silver bullet for using in-network computing while maintaining power efficiency at all times. Not only do in-network computing designs vary so much between platforms, but also on the same platform there are many components that can affect power efficiency, as shown by the example of libpaxos and DPDK. This is, however, not unlike standard network switching where seemingly "similar" switches from different vendors have up to $\times 2$ difference in power consumption [57].

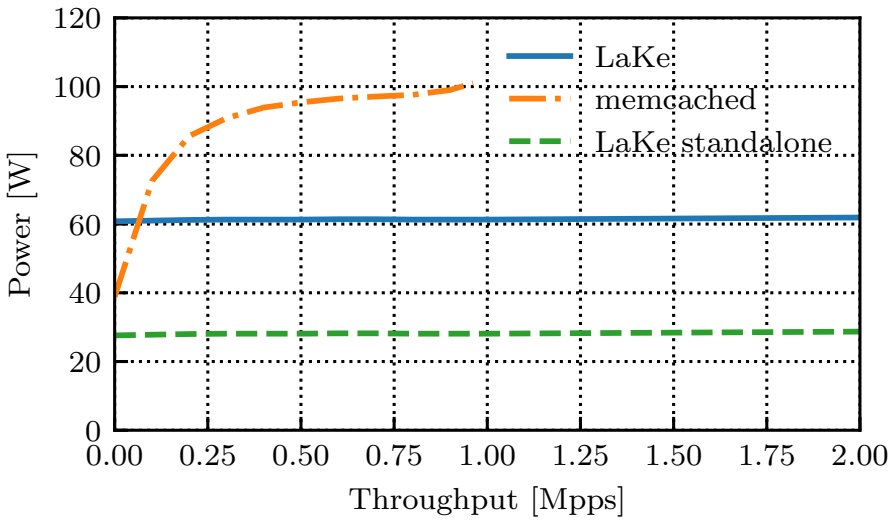


Figure 4.14: Power vs throughput comparison of key-value store.

As there is no doubt that in-network computing offers significant performance benefits (Section 4.5.1), it is essential to get the performance benefit of in-network computing, without losing power efficiency.

Thus, this section proposes *in-network computing on demand*, a scheme to dynamically shift computing between servers and the network, according to load and power consumption. This scheme is useful where identical applications run on the server and in the network, as in our examples. It can be applied to a wide range of applications, though possibly not all (as discussed later). It is also not applicable to bespoke in-network computing applications, which have no server-side equivalent.

The power consumption using in-network computing on demand is illustrated in Figure 4.15. As the figure shows, at low utilization power consumption is derived from the properties of the software-based system. As utilization increases, processing is shifted to the network, and the power consumption changes little with utilization.

Here, the communication cost associated with in-network computing on demand is considered. Stateless applications will require no additional communication cost to run, whereas stateful application will have a communication cost that is bounded by the communication cost of shifting the application from one server to another. The networking device providing in-network computing services is expected to be en-route to a server running the application (otherwise it is not in-network computing, but standard offloading), meaning that no additional latency is introduced.

Two components are required to support in-network computing on demand. The first is a controller, deciding where the processing should be done and when the processing should be shifted between a server and the network. The second is an application-specific task, which may be null, in charge of the actual transition of an application.

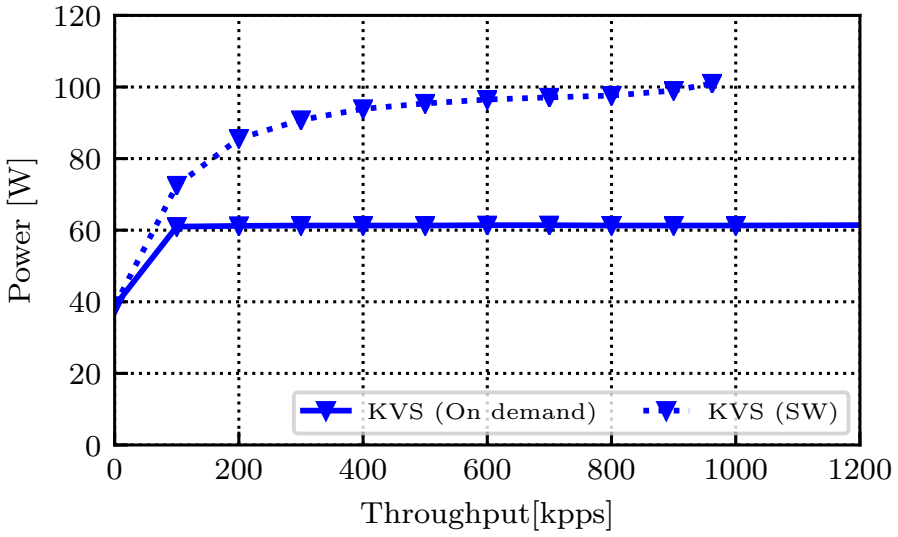


Figure 4.15: Power consumption of key-value store using in-network computing on demand. Solid lines indicate in-network computing on demand, and dashed lines indicate software-based solutions.

4.5.3 In-network Computing On Demand Controller

Two types of in-network computing on demand controllers are considered: host-controlled and network-controlled. This section shows proofs-of-concept for both approaches and evaluates them. There are trade-offs between the two approaches. The network-controlled approach typically reacts faster, but must make its choices based on fewer parameters. On the other hand, host-controlled takes time due to measure power consumption and CPU usage.

Network-controlled in-network computing. The first controller design makes offloading decisions in the network hardware, based on the traffic load. The goal is to reduce load on the host as early as possible, to take out bottlenecks, and provide another layer of offloading (rather than encumbering the host with an additional controller). The control is not entirely automatic: all of its parameters are configurable.

Figure 4.16 shows network-controlled in-network computing scheme. The controller utilizes a pair of parameters to shift a workload from the host to the network. The first parameter is the average packet rate that would trigger the transition, and the second is the averaging period (implemented as a sliding window). If the average packet rate of the accelerated application exceeds the packet rate threshold over the averaging period, the device transitions the workload to the network. A mirror pair of parameters is used to shift workloads from the network back to the host. Using two sets of parameters provides hysteresis, and attends to concerns of rapidly shifting workloads back-and-forth between the host and the network.

The values used to configure the controller are taken from the evaluation of the target application, such as introduced in Section 4.5.1. This is, however, a downside of this approach, as it requires some

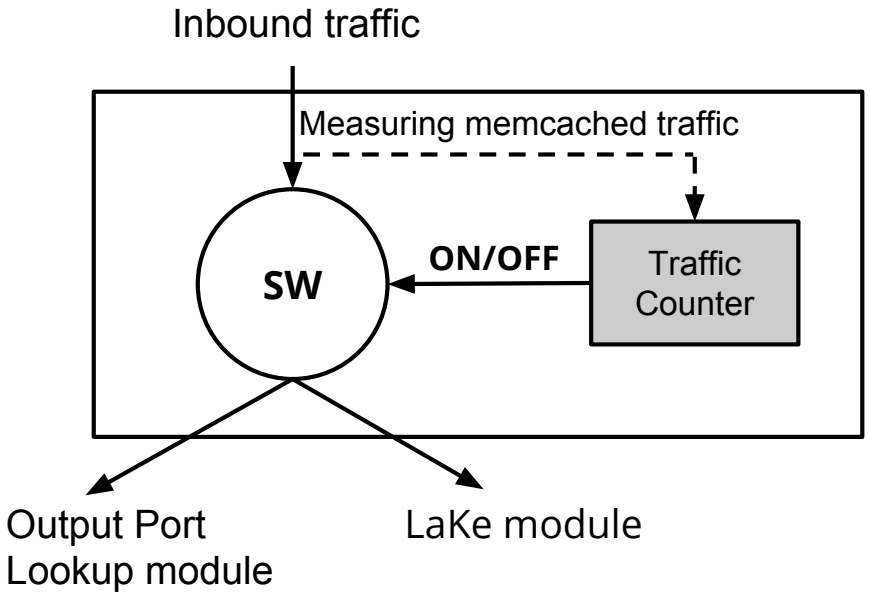


Figure 4.16: Network side in-network computing controller

knowledge of the observed application. A second drawback is that this approach does not take into account the actual power consumption of the host. A drawback of this approach is that it does not take into account the actual power consumption of the host. It only has access to the packet rate. Different applications have very different power profiles [58], and there is no suitable heuristic that can be applied to the shifting thresholds. The proposed controller is implemented in 40 lines of code within the FPGA's classifier module, and consumes negligible resources (order of 0.1%).

Host-controlled In-network Computing. The second controller design makes offloading decisions at the host, using information such as the CPU usage and power consumption. A shift occurs when there is a clear power consumption benefit, and the offloading leading leads to a performance gain. A shift may also happen when computing demands exceed available resources, and the network provides extra computing capacity.

Similar to the network-based controller, the host-based controller maintains two sets of parameters: one for shifting the workload to the network, and one for shifting the workload back. As long as the application is running, the controller monitors its CPU usage. The end-host's power consumption is also monitored using running average power limit (RAPL). If the application exceeds a given power threshold set for offloading, and CPU usage is high, the controller shifts the workload to the network. Monitoring the power consumption alone is not sufficient, as high power consumption can be triggered by multiple applications running on the same host. As before, the information is inspected over time, avoiding harsh decisions based on spikes and outliers. In order to shift back to the host from the network, the controller needs information from the network (e.g., packet rate processed using in-network computing). Otherwise, the shift may be inefficient, or cause a workload to bounce back and forth. The proposed controller is implemented in 204 lines of code, and consumes only

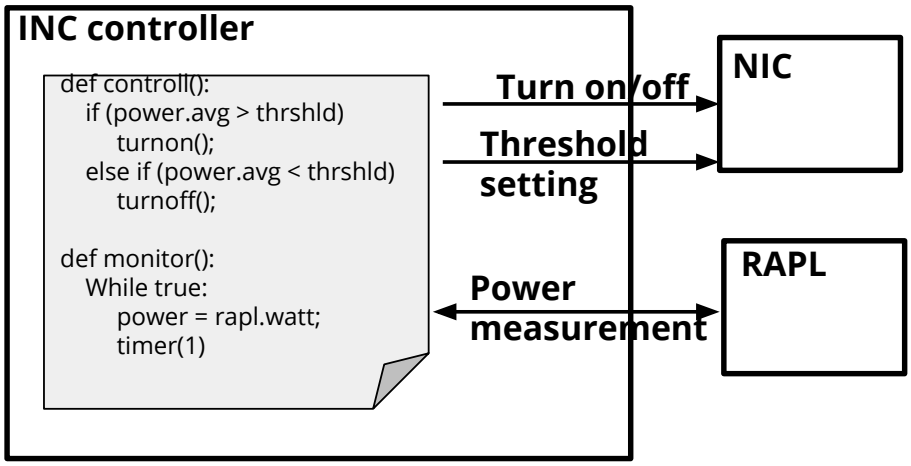


Figure 4.17: Host side in-network computing controller

0.3% CPU usage, mainly for performing RAPL reads.

The host-controlled approach provides better control and flexibility to the user. Yet, care needs to be taken when benchmarking a workload [59]. An advantage of the host-controlled approach is that we do not need application knowledge. Unlike the network-driven approach, one needs to know only CPU usage and target power consumption. This does not mean that any application can be offloaded: the network device must support the application. However, there is no need for a full power/performance characterization. The host-based controller is not perfect: it does incur a (small) processing overhead and it does require longer reaction cycles than a network-hardware controller. The algorithms used are naive, providing a proof of concept. They can be enhanced by more sophisticated algorithms. In energy proportional servers, energy efficiency is not linear, though power consumption still grows linearly with utilization [60], and algorithms such as those based on PEAS [61] may improve energy consumption. These algorithms are beyond the scope of this section and remain part of future work.

4.5.4 Discussion

4.5.4.1 Managing in-network computing on demand.

Power consumption differs between use-cases, and each application has a different optimal threshold for switching between software and hardware. While we demonstrated a transition based on throughput, a different approach would be to monitor the utilization of CPUs or VMs. While monitoring power consumption sounds like a palatable idea, it would be hard to apply it in a machine serving multiple users and running multiple services. While there is a value in characterizing the power consumption-to-throughput of each application, a more practical approach would be to apply a rule of thumb to migration, based on throughput or CPU utilization. Such a value would need to

apply for hysteresis and ignore short bursts of activity.

Generality of in-network computing on demand. in-network computing is not the magic cure-all for data center's problems. Not all applications are suitable to be shifted to the network, and the gain will not be the same for all. In-network computing is best suited for applications that are network-intensive, i.e., where the communication between hosts has a high toll on the CPU. Latency sensitive applications are also well suited for in-network computing. It is no coincidence that the most popular in-network computing applications to date are caching related [29, 43, 44]. Caching provides a large benefit in the common case, and a way to handle tail events. Other applications may find in-network computing on demand to be hard. For example, using Paxos in the network is hard, and doing it on demand is even harder. The effort of implementing an in-network computing solution may just be too high for some applications. Furthermore, each application may have a different power consumption gain, as shown in Figure 4.15.

In-network computing alternatives. Readers may wonder if there are no simpler solutions to increase application performance, rather than in-network computing. One solution, for example, is using multiple standard NICs in a server to achieve higher bandwidth [62, 63]. However, this approach comes at the cost of more NICs, increasing power and price. Alternatively, one may use multiple servers, or opt for a multi-socket or multi-node architecture [64]. These may be cost and power equivalent to an FPGA, a smartNIC, or an ASIC based design. But, their performance per watt is unlikely to match the ASIC-based solution. GPUs are efficient for offloading computation-heavy applications, but as they are not directly connected to the network, they are less suitable for network-intensive applications.

FPGA, SmartNIC or Switch? "Where should I place my in-network computing application?" one may wonder. The answer is not conclusive. Today, a switch ASIC can provide both the highest performance and the highest performance per Watt. It may not be, however, the cheapest solution, with a price tag of $\times 10$ or more compared to other solutions[2]. Use a switch as the place to implement in-network computing leads to other questions. What is the topology of the network? Can and will all messages travel through a specific (non addressed) switch? What are the implications of a switch failure (as opposed to a smartNIC/FPGA next to the end-host)? The answers are all application and data center dependent.

SmartNICs maintain the same power consumption as NICs, typically limiting their power consumption to 25W supplied through the PCI express slot, while achieving millions of operations per Watt, including external memories access [65, 66]. Many smartNICs are, in-fact, FPGA based [67–69].

Of the three, FPGA (and FPGA-based smartNICs) is likely to provide the poorest performance and performance per Watt, due to its general purpose nature. Yet, FPGA performance per Watt in real data centers is not significantly below ASIC. Azure's FPGA-based AccelNet SmartNIC [67] consumes 17W-19W (standalone) on a board supporting 40GE, providing close to 4Mpps/W for some

[2] List prices, obtained from `https://colfaxdirect.com`

use cases. This is slightly better, but on a par with, the FPGA-based power consumption reported in this work. The big advantage of FPGA, and FPGA-based platforms, is their flexibility—the ability to implement almost every application and to use (on a bespoke board) any interface, memory or storage device. ASIC-based smartNICs may not be suitable for every in-network function, but for many applications, they will provide a good trade-off of programmability, cost, maturity and power consumption.

4.6 Summary

This chapter introduced in-NIC key-value cache design for a practical case. While L1 key-value cache consists of only on-board DRAM in the previous chapter, this chapter introduced L1 key-value cache composed of on-chip RAM and on-board DRAM to hide latency and to improve performance. LaKe was introduced as a new architecture for energy efficient in-NIC key-value cache. LaKe can serve as a switch or a NIC, while presenting a multi-core, multi-level cache architecture, that balances throughput, latency and power efficiency. LaKe achieves $\times$17 better energy efficiency than running on a host, with $\times$6.7 to $\times$13.6 higher throughput, maintaining two orders of magnitude better latency. LaKe does all that without giving up memcached functionality and while supporting a large and scalable number of keys. Further, this chapter introduced optimized scheduling schemes for power efficiency.

Chapter 5

The Case for DDoS Amplification Attack

This chapter introduces the applied case to DDoS mitigation, the important Internet security issue. Section 5.1 introduces background of DDoS amplificaiton attack. Section 5.2 introduces related work regarding DDoS detection and prevention. Section 5.3 then proposes architecture and detection scheme. Section 5.4 shows evaluations for absolute performance and capability of our system. Section 5.5 discusses further research and then Section 5.6 remarks summary.

5.1 Introduction

Distributed Denial of Service (DDoS) attack is a significant problem of the Internet. Major DDoS attacks exhaust network or server resources of victims. Amplification attacks using UDP-based protocols such as DNS and NTP try to exhaust link capacity on victim networks. TCP SYN flooding attacks lead to victim servers consuming CPU resources. With the growth of importance of services on the Internet, DDoS attacks are not negligible from the viewpoint of service availability. Hence, mitigating DDoS attack has become increasingly important.

The volume of DDoS attacks to exhaust network resources is increasing due to the growth of link speed and server performance. Recently, the DDoS using Mirai botnet is reported that the most massive traffic volume of DDoS attack becomes over 600Gbps with infected 600k devices [70]. In addition, as of Feburary, 2018, memcached is also used as an amplifier and the attack becomes more than 1.35TBps, targeted to Cloudflare and Github. Akamai reported that Github servers receieved 1.35TB traffic [71]. Also, 66% of DDoS attack's targets are customers, such as end users, financial and hosting services. The customer Autonomous Systems (AS) purchase transit connectivities that are 10, 40 and up to 100Gbps links from transit network providers in general [72]. Therefore, today's DDoS attacks exhaust transit links effortlessly; and thus DDoS attacks should be mitigated in transit provider sides to protect customer networks.

On the other hand, a current dominant type of DDoS attack is DNS-based amplification attack. The DNS protocol has a high traffic amplification rate, and there are over 12 millions of open recursive resolvers around the world that can be used for amplification attack [73]. Furthermore, Arbor

Networks reports that DNS occupies 85% of protocols used for amplification attacks [74]. Thus, if DNS-based amplification attack is wholly prevented, most of DDoS attacks can be eliminated. It is a critical issue to prevent DDoS traffic at a transit link since an Autonomous System needs to pay the cost on communication fee with a transit AS depending on the volume of traffic.

This section focuses on the DDoS mitigation on a transit AS's link. To address the increasing volume of DDoS attack, detection, and prevention at the AS-level link bandwidth are required. In the manner of software mitigation scheme, it is insufficient to forward the packets on data-plane to satisfy over 10Gbps links. To build a middlebox for line rate mitigation, we need to choose a hardware-based solution to achieve high forwarding throughput. This section proposes a novel hardware-based DDoS mitigation system, called mitiKV, that focuses on DNS amplification attack. The mitiKV works as an inline middlebox on a transit link between a transit provider and a customer network. mitiKV detects and discards DDoS packets toward customer network without fail by using ICMP port unreachable message. In addition, detecting and filtering the attacks are all processed in hardware-based key-value store on Field Programmable Gate Array (FPGA), so that mitiKV achieves high throughput up to 100Gbps. A prototype system was developed on a NetFPGA-SUME board [27] and was demonstrated to prevent DNS-based amplification attacks as a proof of concept.

The evaluation shows that mitiKV has a capability of mitigation in 10Gbps line rate. Besides, it was tested with the Internet traffic combined with DDoS traffic. mitiKV can prevent only DDoS traffic without affecting the normal traffic of the Internet traffic. This methodology can apply to the other types of UDP-based amplification attacks.

5.2 Related Work on DDoS Mitigation

This section will show related work in terms of detection methodology and storage design for detected rules on DDoS mitigation.

5.2.1 Detection Methodology

DDoS detection mechanisms can be classified into two categories: anomaly detection and pattern matching [75]. Anomaly detection based methods collect normal system or network behavior regularly and compare present state with the normal state to detect anomalies. PacketScore [76] based on anomaly detection provides a score, called Conditional Legitimate Probability, that can be used to decide a packet is malicious or not. An advantage of pattern matching over anomaly detection is that it can detect several known attacks without fail. Snort [77], a popular open-source intrusion detection system using pattern matching, has wide usage. The proposed detection method is also one of pattern matching methods focusing on DNS-based amplification attack.

In addition to DDoS detection mechanisms, data-plane systems for packet forwarding and filtering are also a fundamental part of DDoS mitigators. In such data-plane systems, high throughput is a

technical issue to overcome the growth of attack traffic volume. CYSEP [78] is a hardware architecture for firewall, encryption/decryption, message authentication, and DDoS mitigation. The DDoS detection system of CYSEP is PacketScore [76], and it is designed to be implemented in ASIC. The proposed mitiKV and CYSEP mitigation module prevent link congestion attacks from exhausting on high-speed networks by hardware implementation. On the other hand, other hardware-based mitigation approaches, Sentinel [79] and SQL DDoS Mitigator [80], mitigate end host CPU resource exhaustion attacks by generating and sending CAPTCHA [81]. While these hardware-based solutions focus on conventional anomaly detection to accelerate statistical processing and prevention system, this chapter proposes a novel detection scheme and optimal hardware-based key-value store to hold the rules generated by this scheme. FPGA-based DDoS detection by using an effective correlation measuring is proposed in [82]. It needs 354ns to distinguish DDoS attacks. In this chapter, we are assuming the case of the Internet backbone, and DDoS mitigation requires line rate processing on the link. Although the sophisticated algorithm to detect anomalies takes a certain time, once rule is added, simple pattern matching can detect DDoS without statistical processing, which is low latency.

As opposed to hardware-based technique, software-based packet inspection implementations such as Snort [77] have plenty of flexibility to inspect and filter packets from the network. Although software-based technique historically has performance drawbacks, many proposals were addressed to alleviate the drawbacks to take both advantages of high-performance and flexible packet processing simultaneously. Hardware-based approaches can achieve high throughput keeping low latency, compared with software-based solutions. GASPP [83] is a network traffic processing framework which integrates Graphic Processing Units (GPU) for their packet processing purposes. While taking advantage of flexible packet processing such as filtering with a regular expression specified by users, it optimizes memory usage as well as packet scheduling for packet processing to speed up its processing. It also integrates Snort to accelerate the performance to mitigate traffic from attacks. The proposed method does not require such a flexible method for packet processing as it only needs to look at a specific portion of a packet. This simplified detection with protocol behavior gives us an opportunity to implement the mitigation middlebox as hardware.

5.2.2 Storage Design for Detected Rules

Hardware-based key-value store [10, 14, 84] has been studied last five years (e.g., FPGA and ASIC). It is a suitable platform for networked systems such as DDoS filtering, since hardware-based solution utilizes an external memory for large capacity to manage a key-value pair with low latency, compared with software-based key-value stores.

Content addressable memory (CAM) is utilized on a commercial network switch and a router to retrieve the destination of an incoming packet and to refer rules against a packet flow (e.g., ACL). While ASIC-based CAM has a 40M-80M bits capacity [85], mitiKV prototype has 262k entries (34M bits) on BRAM; it can be expanded with an external memory. For the defense of recent

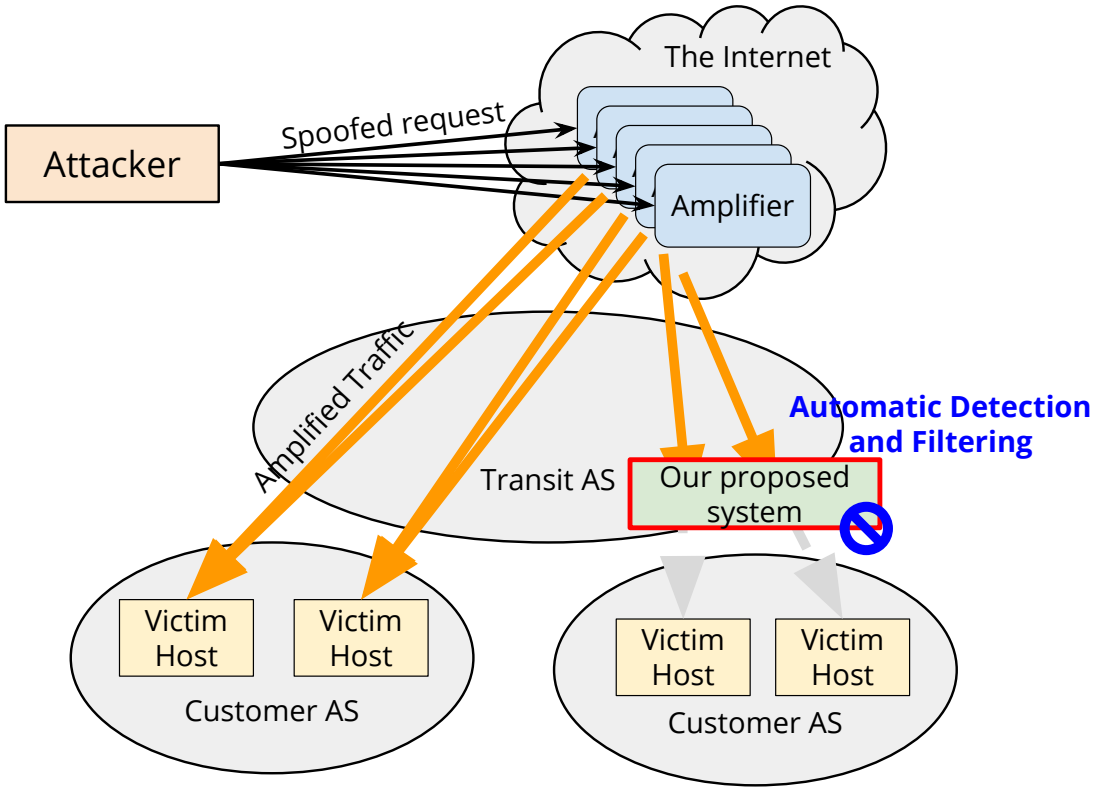


Figure 5.1: The overview of mitiKV installation on a provider network. mitiKV box is installed on a provider side of a customer link and works as an inline middlebox.

DDoS attack, it includes a number of flows because IoT devices are targeted, so DRAM can hold the capacity to manage the flows rather than CAM. To satisfy this characteristic, the key-value store is used to manage IP-based 4-tuple on an FPGA, while ASIC-based CAM can hold the same capacity with FPGA's internal RAM but it is expensive.

5.3 mitiKV Architecture

The mitiKV is a hardware-based, autonomous, inline DDoS mitigation system, which generates DDoS rule automatically and prevents them. Figure 5.1 shows the overview of mitiKV on a provider network. The attacker sends DNS queries to amplify the response packet size, then DNS open resolvers reply to victim servers. Since an AS pays the connection fee with the peered AS according to the volume of traffic, it is a critical challenge to reduce DDoS traffic. Thus, mitiKV is located in front of the customer AS to prevent DDoS packets and identifies the DDoS traffic and generate 4-tuple rules for the filtering automatically. Section 5.3.1 proposes a novel DDoS detection scheme with protocol behavior by using an ICMP error message, and Section 5.3.4 then provides hardware

design. DDoS defense mechanisms can be characterized with three features: activity level, cooperation degree, and deployment location [75]. The proposed mitiKV is discussed in terms of these features below.

- **Activity level:** mitiKV is reactive and pattern matching based detection mechanism. mitiKV achieves high speed detection and prevention by focusing on DNS amplification attack.
- **Cooperation degree:** mitiKV does not need any other traffic measurement or filtering systems. mitiKV performs DDoS attack detection and filtering with dedicated hardware. Besides, multiple mitiKV boxes installed on multiple customers' links work autonomously.
- **Deployment location:** mitiKV is located on a transit link which is connected to other networks. It protects a link connected to a customer network against DDoS attack.

In this manner, mitiKV works as an inline DDoS mitigator on customers' links to protect customer servers against DNS-based amplification attack.

5.3.1 DDoS Detection by ICMP Behavior

The key idea of the DNS-based amplification attack detection mechanism of mitiKV is leveraging one of the most basic mechanisms of the Internet, **ICMP Port Unreachable** message. Since ICMP port unreachable packet has 64 bytes of the original datagram's data [86], DNS-based amplification attack can be identified by checking ICMP port unreachable message on customers' links.

The detection sequence is shown in Figure 5.2 and is explained below. Figure 5.3 shows the state machine to manage 4-tuple state discussed below.

1. DNS Query: An attacker sends a DNS query to a DNS open resolver. The attacker spoofs its source IP address as victim host's IP address.
2. DNS Response: The DNS open resolver receives the query and replies to the spoofed source IP address to the victim host via a transit link. mitiKV learns 4-tuple that consists of source IP address, destination IP address, source UDP port number and destination UDP port number. mitiKV updates the state of flow into **Suspected**.
3. ICMP Port Unreachable Meassage: The victim host receives the unexpected DNS response packet and replies ICMP port unreachable message including the DNS response packet on its payload. mitiKV updates the state of 4-tuple into **Filtered** and decides this as DDoS attack.
4. DNS Query: Repeating steps 1 and 2.
5. DNS Response: The DNS open resolver replies against spoofed queries, but mitiKV already learned the 4-tuple and recognized that these packets were related to the DDoS attack. Thus, mitiKV discards the packets.

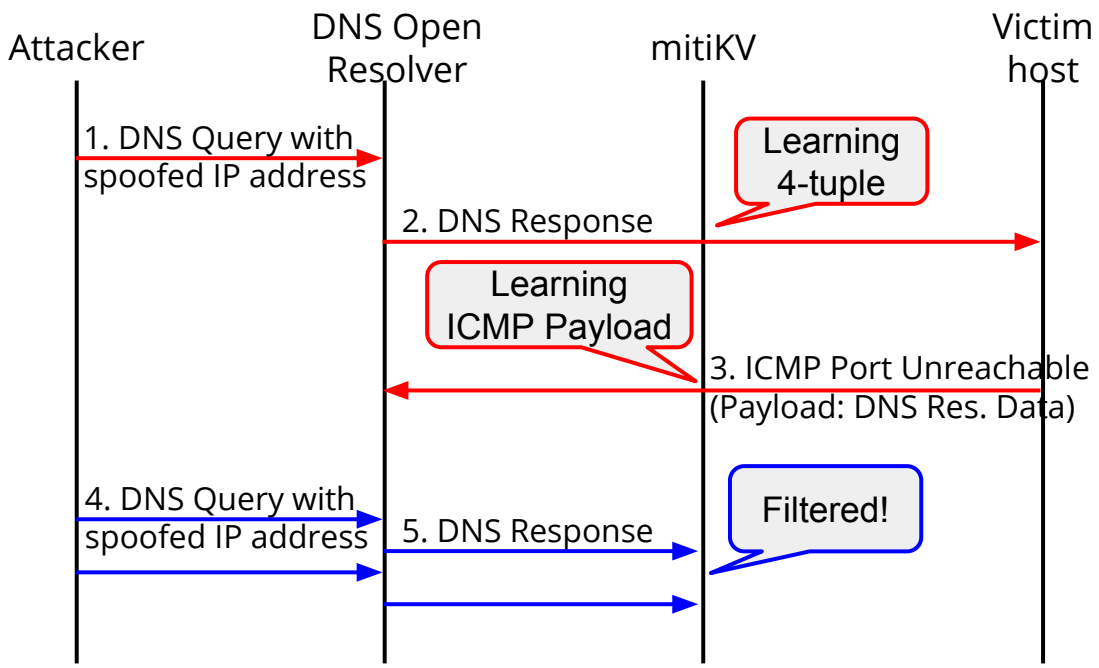


Figure 5.2: The detection sequence of mitiKV.

By this method, mitiKV can detect certain DNS-based amplification attack and prevent them immediately. Because this method relies on ICMP messages, if they are perfectly filtered by firewalls between any victim and amplifier hosts, the proposed method does not work. Please note that it is assumed that amplifiers are distributed across ASes. In this case, the proposed method can reduce DDoS traffic if network paths where ICMP port unreachable message can go through exist between ASes.

5.3.2 Hash Table Size Managed on mitiKV

Hash table size is the critical factor in terms of mitigating the DDoS attack. DDoS attackers may spoof multiple source IP addresses. In this case, the number of combinations of source IP addresses and source UDP port numbers that query packets include is important to mitigate the DDoS attack. Thus, hash table size whether how many key-value pairs to be stored should be determined depending on the expected number of flows, and key conflicts should be handled.

The hash table size is limited by a physical hardware capacity in which an FPGA has limitations of an external memory module [25]. A simulator, written in C, of mitiKV behavior was built to evaluate hash table sizes and the number of mitigated packets. This simulator performs hash calculation by the combination of source IP address, destination IP address, source UDP port number, and destination UDP port number. Then, a hash table entry indexed by the calculated hash value is accessed. Finally, hit ratio is calculated from this procedure. Note that *hit* means that the pointed hash

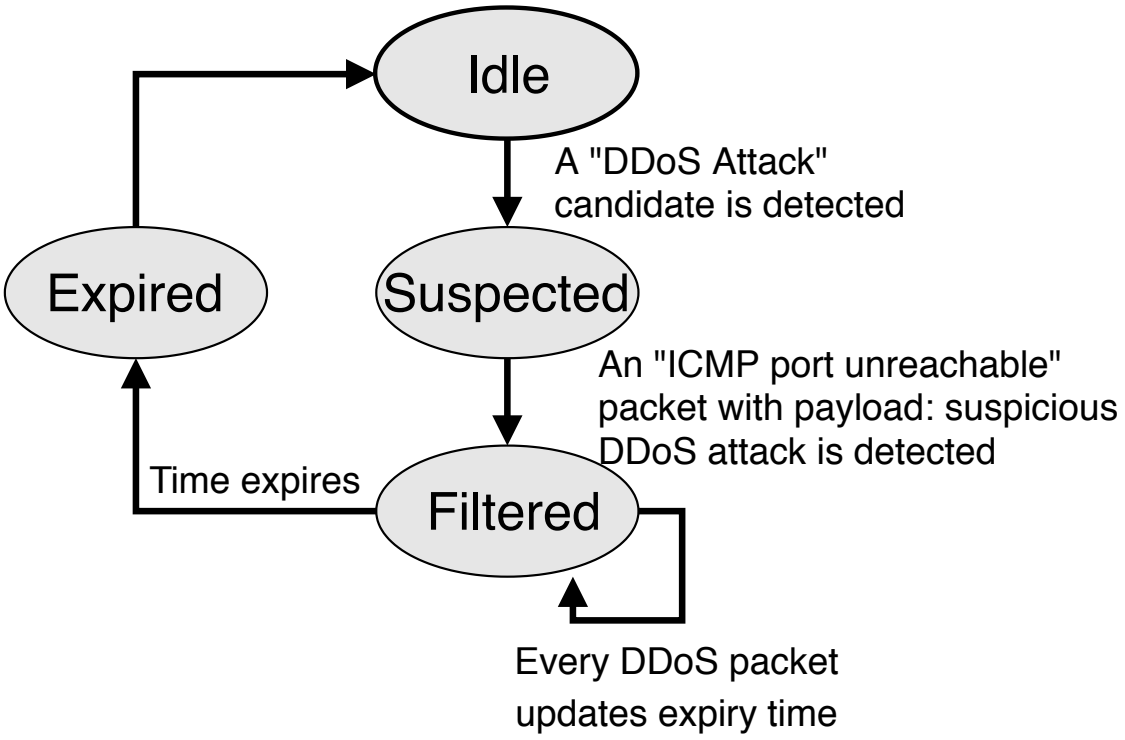


Figure 5.3: State machine of mitiKV that manages 4-tuple.

table entry can be inserted without overwriting a key-value pair which has already been registered. A compulsory miss is counted as *hit*.

Here, a simple incremental pattern was used as data pattern, where combinations of destination UDP port and source IP address are incremented for each query. Hash functions used are CRC32 and lookup3 of Jenkins hash. CRC32 is often implemented as dedicated hardware for calculating frame check sequence on Ethernet MAC. Jenkins hash is often used in a hardware design of key-value store [10].

Figure 5.4 shows simulation results when hash table size is 1k and 262k entries using CRC32 and Jenkins hash to calculate hash table index, respectively. The x-axis denotes the number of combinations of amplifiers' source IP addresses and destination UDP port numbers. Y-axis denotes the hit ratio of packets which are mitigated by mitiKV. In 1k table size, mitiKV cannot mitigate these packets related to DDoS attacks when the number of combinations is over 600 on CRC32. In 262k hash table size, mitiKV cannot mitigate packets when the number of combinations is over 500,000. In Jenkins hash, higher hit ratio than the ratio of CRC32 is observed. When the number of combinations is 1,000, the hit ratio is 38%. This result implies that hash table is used efficiently when Jenkins hash is adopted, compared with CRC32. Thus, we have to choose hash table size carefully taking expected traffic amounts and network interface of transit link into consideration. It is required to further investigate the hashing system including the hash function to obtain the higher hit ratio on mitiKV.

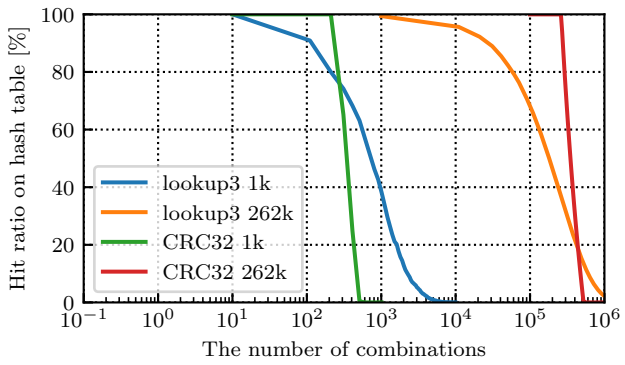


Figure 5.4: Simulation result on CRC32 and lookup3.

5.3.3 Rule Management on Hardware-based Key-value Store

Hardware-based key-value store is used as rule management for decision of all packets to pass-through or to be discarded.

The key-value store was optimized to meet the required network middlebox as described in the following. The assumed data structure is the 4-tuple flow label as the key, which is comprised of source IP address, destination IP address, source UDP port number and destination UDP port number and an expiry time as the value as shown in Figure 5.5. In this context for a network middlebox, since key length and value length are fixed size, we can reduce the number of lookup to tables (e.g., hash table and bucket list).

Key-value store and hash table are looked up at the expense of the wildcard. We can also think the number of tuples for each flow label n can be smaller than 4 to achieve efficiency of table size. We introduce trade-offs from a real Internet traffic, which lead to errors when adopting n-tuple-base. We explain each tuple attributes at the receiver of DDoS traffic as described in the following.

- **4-tuple {Source IP address, Destination IP address, Source UDP port number, Destination UDP port number}**
  Although the 4-tuple flow level attributes can specify a fine-grained UDP-based flows, the attack with randomized port is not absorbed due to limitation of rules generated by 4-tuple. It takes time until the mitigator learns all the flows. On the other hand, mitigator decides the action by 4-tuple flow, so false positives are reduced.

- **3-tuple {Source IP address, Destination IP address, Source UDP port number}**
  Some attackers use randomized destination port numbers against victim hosts. Since the number of tuples can be aggregated due to the lack of destination UDP port, the number of stored entries can be reduced and mitigator uses memory more efficiently. The attack by concerned protocol is mitigated between specified hosts regardless of randomized ports.

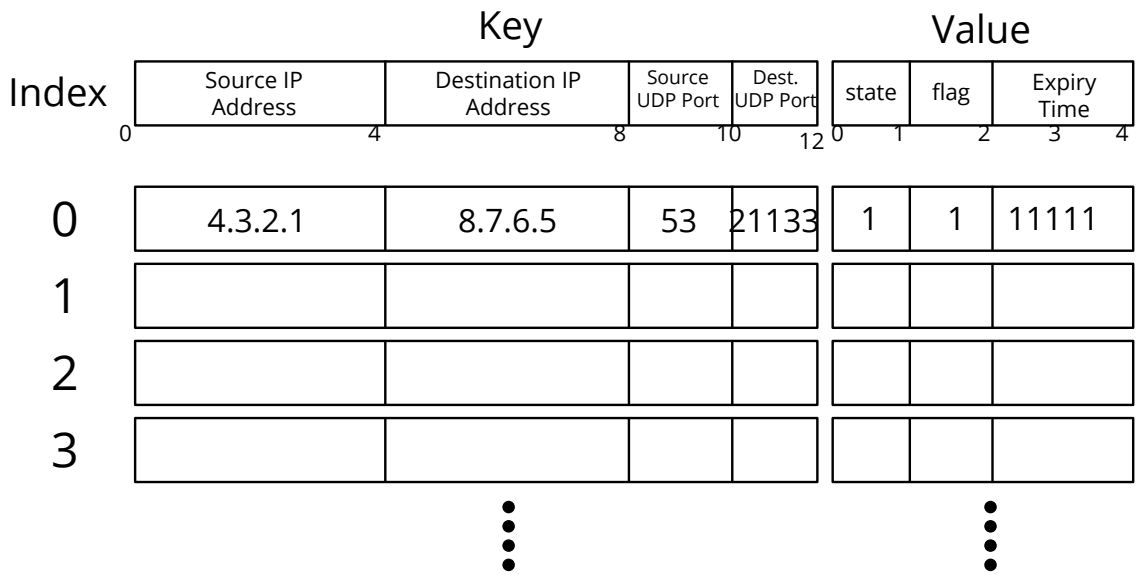


Figure 5.5: Hash table design on mitiKV.

- **2-tuple {Source IP address, Destination IP address}**
  The mitigator recognizes that the traffic from source IP host to destination IP host is perfectly malicious and mitigated traffic. Instead, false positives increase.

- **1-tuple {Source IP address}**
  The mitigator recognizes that the traffic from the host with the concerned source IP address is perfectly malicious. The key size is only 4B, so store space is effectively used and has more entries than any n-tuple key. However, the detection by a single ICMP packet leads filtering traffic from the host even though the host sends legitimate traffic.

Since the constraints of memory capacity affect the hit ratio, the efficiency of memory capacity is critical. When hash collision occurs, the entry is updated to a new key if an expired time is invalidated. Otherwise a new key is ignored.

This chapter focuses only on 4-tuple base data structure. Since key-value store uses a hash function to look up the hash table, the number of the rule entries and the hit ratio of hash collision are important design parameters.

5.3.4 Hardware Design

Figure 5.6 shows the proposed mitiKV architecture overveiw. mitiKV consists of the following modules.

- **DDoS Attack Filter module** : This module monitors all packets to filter the packets related

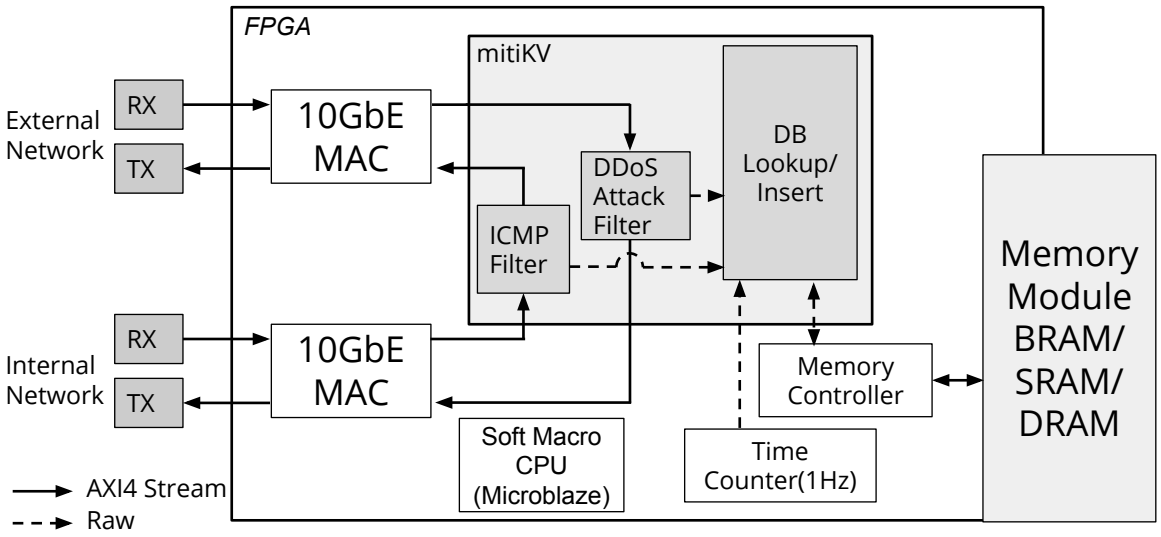


Figure 5.6: mitiKV architecture.

to suspected DDoS attacks. When DDoS attack packets are filtered, the module creates a tuple which will be processed in database Lookup/Insert module. In the prototype FPGA implementation, we assume DNS-based amplification attack as attack traffic in Section 5.3.5. In this case, DNS response packets whose source UDP port number is 53 are filtered.

- **ICMP Filter module** : This module monitors all packets whether packets with ICMP port unreachable and an error packet over the ICMP matched with DDoS suspected packet as mentioned in Section 5.3.1. When packets are filtered, this module sends a query to database to lookup state and then checks whether the packet is suspected DDoS attack or not.

- **Database Lookup/Insert module** : This module provides two interfaces, READ and WRITE interfaces to manage key-value store for flows and their states. The DDoS Attack Filter module and ICMP Filter module access this module to check the 4-tuple flow whether DDoS attack or to update the state. This database is managed by hardware-based key-value store on the memory module.

- **System Counter** : This module is a counter module generating timestamp for expiring logic. The timestamp in value field is checked if time is expired when database module accesses each key-value entry.

An external memory is used as the hash table as shown in Figure 5.6. The hash table can be implemented on DRAM, SRAM, and internal FPGA memory. While SRAM and internal FPGA memory modules has constant access latency, DRAM has dynamic access latency. The difference of these access latencies is hidden by implementing a pipeline deeply in which the number of stages is equivalent to the access latency. On the other hand, the memory size will be a crucial factor to

mitigate attack traffic. To mitigate abusing traffic, the hash table needs to be larger than the size which is capable of storing the number of abusing flows in a period. The hash table size and hit ratio on the hash table is the key component of this approach. They were described in Section 5.3.2.

Figure 5.5 shows hash table design on mitiKV. Each entry consists of key and value. Key is the 12B fixed length and consists of IPv4 source address, IPv4 destination address, source UDP port number and destination UDP port number. Value is 4B fixed length and consists of status, flag and expiring time. The index is calculated by hash function to retrieve the key. The key retrieved from the hash table is compared with the requested key to see if both the requested key and the key from the hash table are identical. Status field is used to identify a suspicious or a filtered flow. Flag field is used for validation of data, which utilizes 1 bit and the other bits are reserved. This validation bit implies that data has been stored. If validation bit is set to 1 and expiry time is expired, this entry becomes invalidation and can store a coming flow.

The expiry time is used to validate filtering rules. According to the investigation of UDP-based amplification DDoS attack [87], the 99-percentile duration is 130.52 minutes. Thus, it is sufficient for 16bit time counter with a second resolution in mitiKV as shown in Figure 5.5. Expired key-value pairs are deleted when the hash table entry is read. Expiry time can be set an arbitrary value by an operator. Operators can use 130 minutes or more based on past rule of thumb above. Since DDoS trend and attack condition would change, network operators carefully choose the expiry time, which affects the communication after recovering from hijacking.

When hash table is fully utilized, a new entry will be added, depending on status field and expiry field. When the status field of an entry is suspicious, a new entry can be replaced. When the status is filtered, adding a new entry is depending on the expiry field. If the expiry time exceeds, a new entry can be replaced.

5.3.5 An FPGA Implementation

This section illustrates a prototype of the proposed mitiKV. The proposed mitiKV is implemented on Digilent NetFPGA-SUME board [9, 27]. An FPGA device used is Xilinx Virtex-7 XC7V690T FFG1761-3. The board has four 10GbE interfaces for communication. Design tool used is Xilinx Vivado 2015.4.

The proposed mitiKV module uses Advanced eXtensible Interface (AXI) Stream [1] as data bus interface. 10G Ethernet Subsystem IP is used as a 10G MAC. The data width is 64bit. This implementation is based on AXI Stream data bus interface.

Integrating CPU into our implementation can provide configurations, statistical information and manual managing of rules on key-value store via a user interface on UART. A soft-macro CPU was not implemented for system configuration for the sake of simplicity.

The key-value store was implemented on BRAMs. The number of hash table entries is 1k and

[1] AXI is a family of microcontroller buses by ARM AMBA.

Table 5.1: Synthesis results.

Hash Table Entry Size	1k	262k
Slice Utilization [%]	4.17	6.21
BRAM Utilization [%]	1.63	64.73

262k to store filtering rules. From the viewpoint of hardware implementation, while CRC32 can be calculated by a single clock cycle, Jenkins hash can be calculated in 6 clock cycles [10], which need 6 steps data pipeline to conceal the latency. Hence, as a hash function, CRC32 was chosen for simplicity. If more capacity for storing rules is required, an external memory such as SRAM or DRAM is also available to store them. As mentioned above, constant access latency for SRAM and dynamic access latency for DRAM are hidden by implementing a deep pipeline. In the evaluation, the BRAM-based mitiKV is used.

Specifically, 10GbE requires running at more than 156.25MHz when AXI stream data width is 64bit. Table 5.1 shows the synthesis report of mitiKV per table size. The hash table sizes we implemented are 1k and 262k sizes. We implemented 262k size hash table, which is the maximum size of single Dual Port RAM IP in the targeted FPGA. The BRAM utilization is 64.73%. In case more hash table space is needed, an external RAM such as SRAM and DRAM can be replaced with BRAMs. Since the slice utilization indicates that mitiKV core is so small compared to the overall FPGA's area, the mitiKV core can be also embedded in other hardware-based network appliances.

5.4 Evaluations

This section provides feasibility in terms of two aspects: *Can ICMP port unreachable message be used for DDoS attack detection?* and *Can mitiKV defense DDoS attack even on the Internet backbone?* Section 5.4.1 analyzes the Internet traffic to provide the answer to the first question. To answer the second question, Section 5.4.2 provides the hardware test to show mitigation performance under the simple environment, and then Section 5.4.3 provides mitigation capability under the Internet traffic. Section 5.4.4 provides results of completion time for DDoS attack.

5.4.1 ICMP Port Unreachable Message

The proposed method in this section, heavily relies on the message encoded by the network stack of the victim hosts. Therefore understanding how the ICMP destination unreachable message is observed at a transit link is important since some implementations may not embed the original packet information which triggers port unreachable message, some middleboxes may strip the packet or just simply filtered out. The standard says *If a higher level protocol uses port numbers, they are assumed to be in the first 64 data bits of the original datagram's data.* [86], which is interpreted as 28 bytes

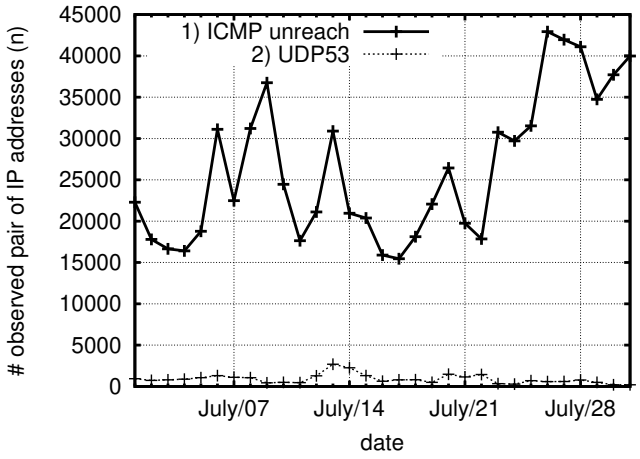


Figure 5.7: Number of IP address pairs among 1) all of observed ICMP port unreachable messages, and 2) 1) with DNS packet in the payload.

in IPv4 (i.e., 20 bytes IP header and 8 bytes higher protocol datagram) of the original packet is in the payload. In the IPv6 standard [88] it specifies differently with 1,280 bytes payload at the maximum size.

This section investigates the availability of the key information for our proposed method by analyzing the public packet trace at educational backbone network. In summary, the ICMP port unreachable messages we observed fulfill our expectation to detect UDP-based amplification attack. The detail will be discussed in the following description.

5.4.1.1 Dataset

We used traffic traces from MAWI (Measurement and Analysis on the WIDE Internet) archive samplepoints F [89] in July 2016 (i.e., 31 days). The archive includes 15 minutes daily packet trace at the transit link of the backbone network, with anonymized IP addresses in the traces.

Although the traffic trace only recorded a short duration in a day as well as with partial length (first 96 octets) of packets, it contains enough information of what we are investigating here —we only need the payload of IPv4 ICMP destination unreachable message and the trend of ICMP message, not the full number of messages exchanged, to justify how the proposed method is practical in the wild.

We focus on the number of source and destination pairs of IP addresses in the trace which 1) have ICMP destination port unreachable (type 3 code 3) messages, 2) 1) with DNS packet (port 53 of UDP packet) in the payload. In addition to that, we counted the distribution of packet size which each packet contains ICMP destination unreachable message to study how the network stack implementation encodes the original packet when it sends back the error.

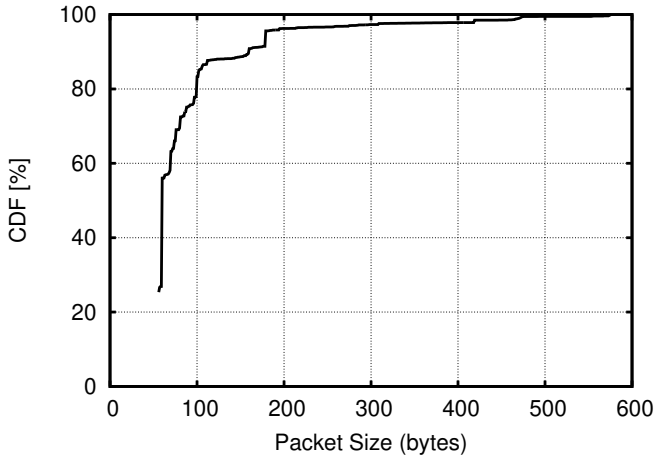


Figure 5.8: Cumulative distribution of the packet size in the ICMP destination error message, which indicates the existence of the original packet to identify it is DNS packet or not.

5.4.1.2 Results

Figure 5.7 shows the number of observed flows in the packet traces which contains the ICMP destination unreachable messages and payloads which the original packets were DNS packets. Figure 5.8 plots the distribution of packet size among all of ICMP port unreachable messages.

As shown in Figure 5.7, there are a number of ICMP port unreachable messages which the original packets are DNS-related (query or response). Also Figure 5.8 represents that the minimum packet size of the original packet is 28 bytes[2], which can contain the port number of UDP to identify if the packet is DNS or not.

Above information confirms that there are sufficient information to identify the packet under suspicious by our proposed method based on the ICMP port unreachable messages.

5.4.2 Mitigation Test under DDoS Traffic

This section shows the capability of DDoS prevention. mitiKV is tested to see whether it can detect and prevent DDoS traffic. We use the implementation with 262k hash table entries as described in Section 5.3.5. Hash table hit ratio in Section 5.3.2 was simulated, so that this hardware should be able to prevent up to around 200k without misses on the hash table.

5.4.2.1 Hardware Environment

Figure 5.9 shows an evaluation environment. Each component is explained as follows, except mkv. Table 5.2 shows specification of each component on measurement environment.

[2]The minimum packet size 56 bytes of ICMP unreachable message can be interpreted as 28 bytes the original packet since the size of IPv4 header is 20 bytes and the size of ICMP Destination Unreachable Message without the Original datagram is 8 bytes.

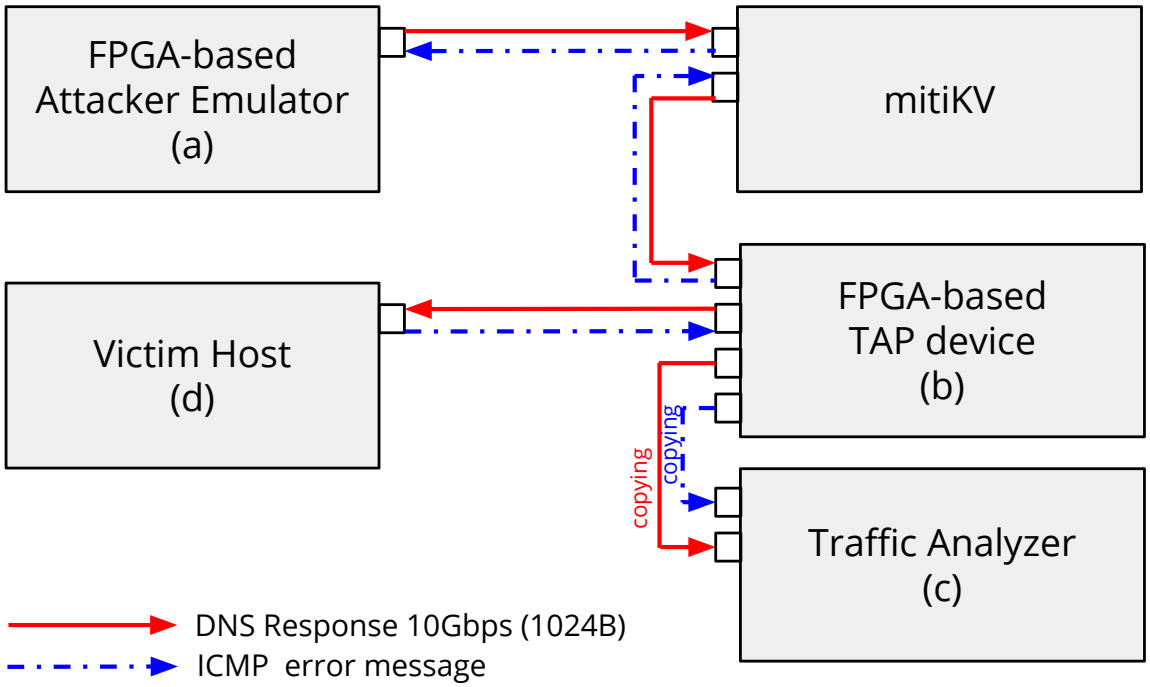


Figure 5.9: Evaluation envrionment.

(a) **FPGA-based Attacker Emulator** : Attacker emulator generates packets related to DNS amplifier attack. More specifically, it generates DNS response packets with response bit by changing the combinations of source IP address and destination UDP port number and sends them to the victim host via mitiKV in 10Gbps line rate. Therefore, packets are generated against multiple destination UDP port numbers from UDP port number 53. In Section 5.4.3, OSNT [90] is used on NetFPGA-SUME card instead of VC709 card.

(b) **FPGA-based TAP device** : In order to measure packets per second (pps) between mitiKV and the victim host, the FPGA-based TAP device is installed. The TAP device has four Ethernet ports. Two ports are bridging: port0 is connected to mitiKV, and port1 is connected to the victim host. Remaining two ports are connected to (c) traffic analyzer for mirroring of inbound and outbound traffic.

(c) **Traffic Analyzer** : The traffic analyzer measures packets per second from mitigator's outbound port and victim host's inbound port. This analyzer performs tcpdump for capturing a large number of packets to measure packets per second in two 10GbE ports: victim host's RX and TX ports. A RAM disk is used as storage to store captured pcap files in consideration of storage throughput. The number of packets per 10 ms from the captured files are analyzed.

(d) **Victim Host** : It is assumed that victim host is a general Linux machine.

Table 5.2: Components of measurement environment.

	Hardware	OS	NIC + Driver	Main memory
(a)	Xilinx VC709	—	—	—
(b)	Xilinx VC709	—	—	—
(c)	Intel Core i5-4590	FreeBSD 10.2R	Intel X520-DA2 + ixgbe 2.8.3	4GB
(d)	Intel Core i7-4770	Fedora 24 (Linux 4.6.6)	Intel X520-DA2 + ixgbe 4.2.1	32GB

5.4.2.2 ICMP Kernel Tuning

A general Linux machine is set as rate limit parameter that defines a packet per one second against one host. Since the mitiKV is located at transit link of network service provider, mitiKV needs to observe all packets in the located network and the connected network. To evaluate mitiKV hardware, the Linux machine denoted in (d) is required to emulate multiple victim hosts located in the connected network. Kernel parameters for ICMP on the victim host were configured for emulating multiple victim hosts temporarily. A Linux host returns an ICMP packet with a port unreachable message per one second against one remote host in an environment with default kernel parameters. Here, the following parameters were tuned as returned packets per one second.

- /proc/sys/net/ipv4/icmp_ratelimit : 0
- /proc/sys/net/ipv4/icmp_msgs_per_sec : 14880000
- /proc/sys/net/ipv4/icmp_msgs_burst : 25600

In default, Linux host has parameters of rate limiting on ICMP packets. The parameter *icmp_ratelimit* controls the maximal rate of sending ICMP packets and can be disabled by setting 0. The parameter *icmp_msgs_per_sec* defines the maximal number of ICMP packets from a host. To emulate multiple victims at a single host in the experiments, the machine was tuned to return the ICMP packets at 10GbE line rate because the rate of short packet (64B) is 14.88Mpps. The parameter *icmp_msgs_burst* can set the burst size of ICMP packets.

5.4.2.3 Results

Figure 5.10 shows the result of DDoS attack to a victim host as shown in Figure 5.9. FPGA-based Attacker Emulator generates DDoS attack traffic which includes 1,000 flows — amplified traffic from DNS server to a victim host (incremental 1,000 destination UDP port numbers, which are equivalent to 1,000 source UDP port numbers from an attacker). This measurement is performed on (c) traffic analyzer. Received packets from two network interfaces on the traffic analyzer machine represent the

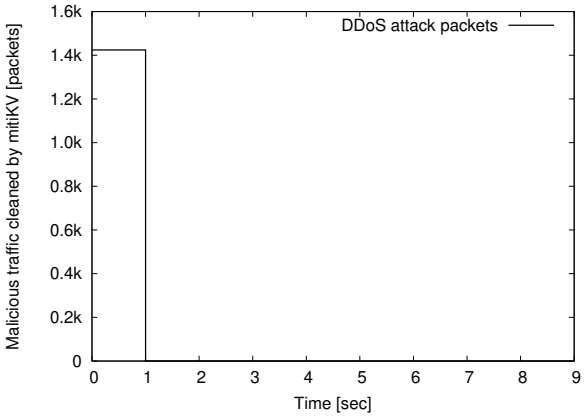


Figure 5.10: DDoS attacking test against a victim host with incremental 1,000 destination UDP port numbers.

Table 5.3: Components of measurement environment.

	Hardware	OS	NIC + Driver	Main memory
(a)	Intel Xeon E5-2637	Ubuntu 16.04 LTS	NetFPGA-SUME + sume_riffa 1.34	512GB
(b)	NetFPGA-SUME	—	—	—
(c)	Intel Core i7-6700K	Ubuntu 16.04 LTS	Intel X520 + ixgbe 5.3.5	64GB
(d)	Intel Core i5-3450S	Fedora 24	Intel X520 + ixgbe 4.2.1	8GB

number of DDoS attack packets and ICMP port unreachable messages, respectively. The figure indicates that the proposed detection method takes time to learn all DDoS flows. Specifically, we need 1,000 DNS packets and 1,000 ICMP port unreachable messages to generate rules for 1,000 flows, but we need to consider round trip time between mitiKV and a victim host. In this time, a victim host receives DDoS packets because mitiKV does not complete generating rules until receiving ICMP port unreachable messages on mitiKV. Although the latency between victim host and mitiKV is small due to the back-to-back connection in this evaluation, the millisecond level latency to achieve ICMP port unreachble messages to the mitiKV occurs due to network devices in the practical case. Figure 5.10 indicates that dropping packets in a first second mean packets to pass through mitiKV until filetering rules are generated. After completing to generate rules, packets are perfectly filtered by mitiKV. The learning time is depending on the round trip time between mitiKV and a victim host. In this case, the victim may receive more attacks in a period of learning flows on the mitiKV because it takes more time to learn DDoS flows.

5.4.3 Mitigation Test under DDoS Traffic with the Internet Backbone Traffic

In this section, we try to understand the behavior of mitigation against the real world traffic. To examine it, a trace was created by combining traffic data and pseudo-DDoS traffic, which is DNS amplified packets with 1,036 bytes as packet size, including 1,000 flows. We use SINET5 [91] backbone traffic as of 18th November 2015 and extracted 1M packets and concatenated it with DDoS amplified traffic which we prepared. In the extracted traffic, average packet size is 688.74 bytes and average throughput is set to 2,038 kpps.

For this experiment, almost the same environment in Figure 5.9 is used, but replaced FPGA-based Attacker Emulator with OSNT [90] on NetFPGA-SUME to replay the traffic file we created. Table 5.3 shows the environment used in this experiment. In traffic analyzer, tcpdump was performed to capture packets on RAM disk. Note that, in this experiment, since the purpose is not throughput measurement but confirming protocol behavior and mitigation test under the Internet backbone traffic, the inter packet delay was set to 10μs on OSNT to capture them without packet loss due to capture machine's performance.

To confirm the effectiveness of mitiKV, two trials were examined: 1) traffic replaying with mitiKV and 2) traffic replaying without mitiKV. In Figure 5.11, while the traffic with mitiKV is cleaned up after mitigation by mitiKV, the traffic without mitiKV shows the traffic including DDoS attacks. We used pcap file including 1,010,000 packets and repeated the traffic file 10 times with OSNT, so we generated 10,100,000 packets in total: 10,000,000 packets as normal and 100,000 packets as DDoS. In the traffic without mitiKV, 10,100,000 packets were captured. When we used mitiKV and tried this experiment five times, 1145.2 DDoS packets on average (standard deviation is 15.6) were passed through mitiKV for learning the DDoS flows by detecting ICMP port unreachable messages. In this case, 98,854.8 DDoS packets of 100,000 DDoS packets were reduced on average. That is, in this setup, the mitigator reduced 98.8% DDoS packets in total when mitiKV is used with the real Internet traffic. Traffic before cleaning up is the almost constant packet rate at 100kpps. In contrast, mitigated traffic shows a periodical wave because we added DDoS traffic in the tail of the regular backbone traffic and repeated replaying the concatenated traffic file.

Thus, mitiKV can prevent DDoS attacks drastically under the Internet backbone traffic. In this case, 100,000 DDoS packets which include 1,000 flows were used. Current BRAM-based mitiKV can hold 262k flows, which can mitigate less than 262k flow attacks. For the recent trending attacks, we can expect a longer duration for DDoS attack and a number of flows for DDoS. In such cases, mitiKV can also utilize external memory as the key-value store, sacrificing its access latency.

5.4.4 Mitigation Completion Time and Rate

DDoS traffic is generated with N_{amp} = 512, 1k, 2k, 4k, 8k, 16k and 32k. The completion time depends on Round Trip Time (RTT) between a victim host and amplifiers, and detection scheme, which we use ICMP-based detection in this section. The number of targeted victims also affects

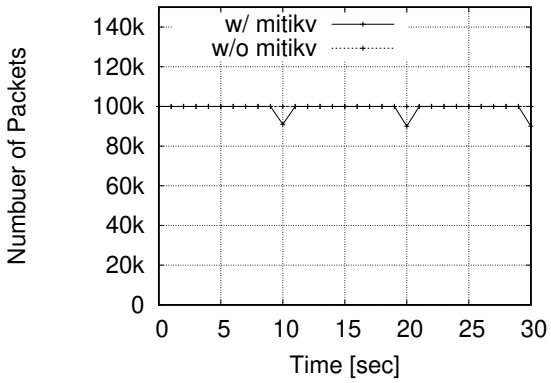
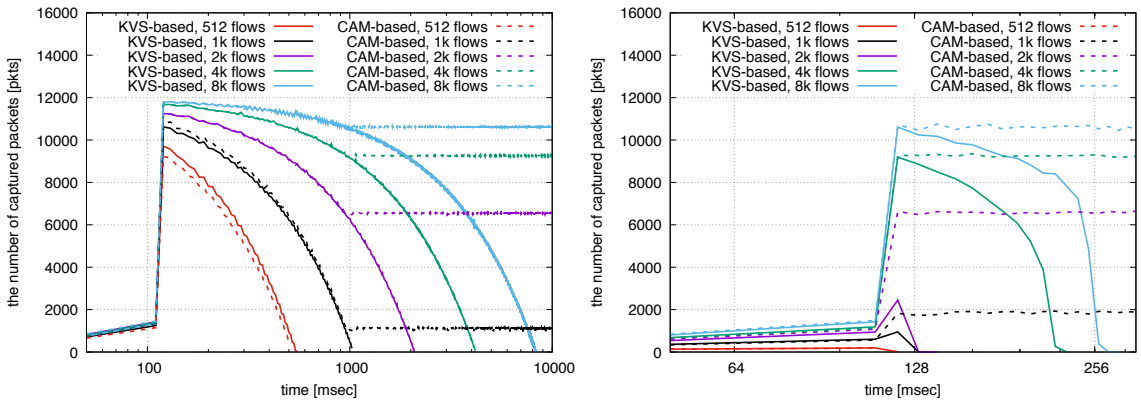


Figure 5.11: Mitigation test with extracted traffic of Internet backbone.



(a) One targeted victim emulation

(b) Multiple targeted victims emulation

Figure 5.12: The mitigation completion time in a local environment. FPGA-based attacker emulator generates packets with N_{amp} source IP addresses to one/multiple target(s). A victim host has limitation of returning ICMP port unreachable messages. In the (a), we did not modify Linux kernel related to ICMP. Thus, a victim host returns an ICMP error packet per 1ms. In the (b), we tuned kernel parameters related to rate limiting of ICMP.

the completion time of mitigation due to the limitation of the response rate of ICMP error messages by a victim host machine configuration. Thus, two scenarios were built using a single machine as victim(s) in the following: 1) the single victim host and 2) multiple victim hosts. 1) uses default Linux machine used as a victim host. 2) uses the same machine with 1) and tuned ICMP kernel parameters. In this experiment, we use CAM-based mitigator to compare key-value store with CAM. This CAM-based mitigator is implemented on the same FPGA board and uses the same detection logic with mitiKV. Note that we use NetFPGA-SUME boards as (a) and (b) on Table 5.3, and replace machines as (c) and (d) with machines that have Intel Core i7 series CPU. Victim host which is denoted as (d) utilizes 64GB RAM.

In this experiment, the following two scenarios were assumed.

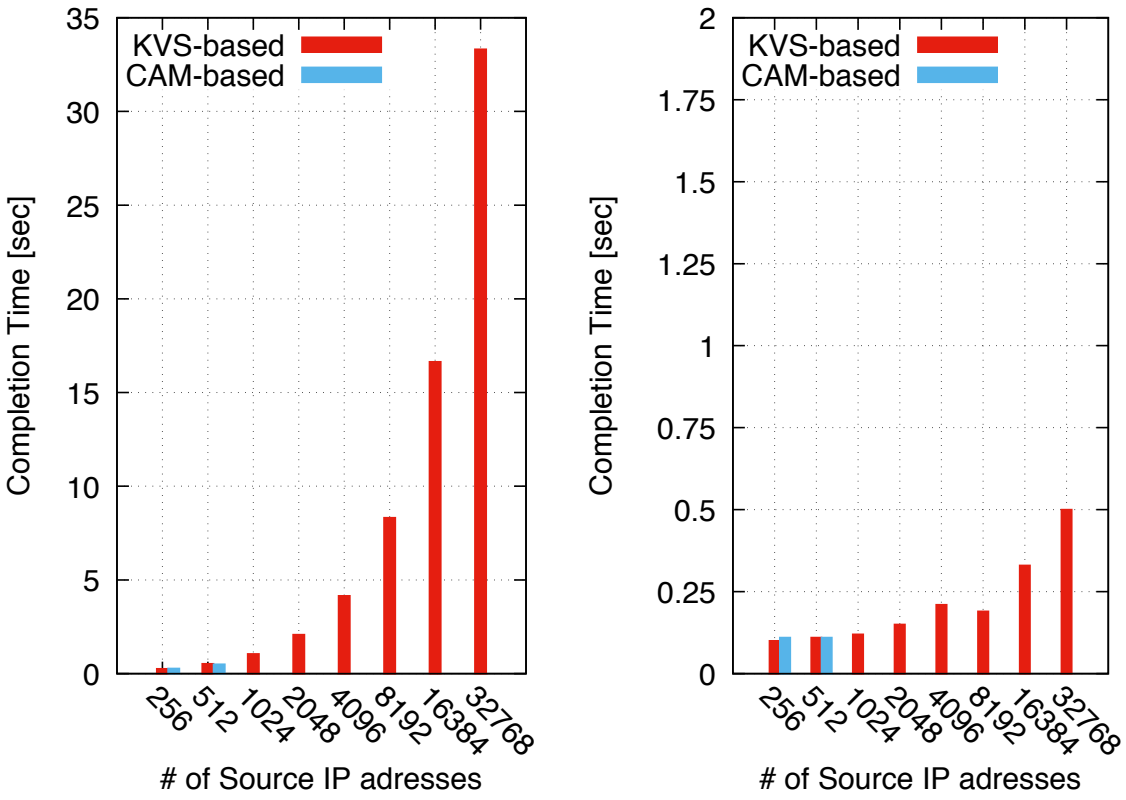


Figure 5.13: The mitigation completion time in a local environment related to ICMP into one victim host. Left figure shows the result when the target is a single victim host. Right figure shows the result when the target is multiple victim hosts.

1. N_{amp} amplifiers attack against **a single targeted victim host** in order to focus on performance of a single node.

2. N_{amp} amplifiers attack against **multiple targeted victim hosts** in order to observe entire packets filtering.

Figure 5.12 shows the result of generating DDoS traffic up to 8k of N_{amp}. In the environment (a) including a single targeted victim host, a single Linux based victim host returns ICMP port unreachable messages in the default speed of 1000 packets per second. Since the proposed mitigator generates rules in the same speed by detecting ICMP port unreachable message and filters packets which are triggered by the rule, the finished time depends on the number of source IP addresses. In the environment (b) including multiple targeted victim hosts, victim hosts which are emulated by setting Linux kernel parameters related to ICMP on a single linux machine, return ICMP error messages without limiting the rate of ICMP error messages. Thus, instant mitigation was observed in the (b) of Figure 5.12. In both environments, though DRAM-based mitigator completely achieves to prevent packets after detection, CAM-based mitigator fails to prevent over 1k source IP addresses because it has limited entries.

Table 5.4: Scalability.

	10GbE	40GbE	100GbE
Clock Frequency [MHz]	156.25	156.25	322.266
Data Width [bit]	64	256	512
DNS Reply Detection [clock cycles]	8	2	1
Hash Function (CRC32) [clock cycles]	1	1	1
KVS Processing [clock cycles]	2	2	2
Pipelining Depth	11	5	4

Figure 5.13 shows the result of completion time to have detected ICMP error messages and to have mitigated DDoS traffic related to them. In the figure, x-axis denotes the number of source IP addresses and y-axis denotes the completion time. As a result, while CAM-based mitigation has not finished over 1k flows in both environments, KVS-based mitigation using DRAM has finished depending on the rate of ICMP error packets.

5.5 Discussion

Scalability up to 100Gbps link

Transit links will be replaced with 40Gbps and 100Gbps high bandwidth link interfaces to provide network services to their customers. We investigated MAC IP specification provided by Xilinx and calculated required clock cycles for a hardware-based pipeline. Table 5.4 shows the scalability on 10GbE, 40GbE and 100GbE. To support 100GbE interface, high-end FPGA series are required to implement mitiKV. It is a simple pipeline to design 40GbE and 100GbE because required clock cycles related to packet parsing for detection of DNS response are reduced due to increasing data width of MAC on 40GbE and 100GbE. To support high-speed interfaces such as 40GbE and 100GbE, a custom FPGA board equipped with these interfaces is required. The mitiKV core design is applicable to these interfaces.

Deployment level

It is assumed that mitiKV is placed in front of the customer AS on aspect of AS's transit link. To manage high volume traffic, DRAM implementation is not negligible. On the other hand, mitiKV can be deployed on the various scale location. This mitiKV can also be effective for the security to locate in front of routers on home networks, office, and university by using proper memory module including DRAM, SRAM, and BRAM.

False positive

ICMP port unreachable message is used for network diagnosis to confirm the port available or not. Therefore, false positive may occur when network diagnosis is used. To avoid this false positive,

expiry time for DDoS duration was introduced. Thus, network diagnosis may fail due to mitiKV's filtering.

Application to programmable switches

Programmable switches are emerging (e.g., Tofino [54]). The proposed scheme is suitable for match-action tables to manage rules processed on the key-value store. Therefore, we are going on developing it for the programmable switch. However, the current model can utilize external SRAM chip. We will also extend this architecture with larger external memory (e.g., DRAM).

5.5.1 Future Work

For the future work, this section summarizes three points.

Protocol supporting

While this chapter focuses on DNS protocol, mitiKV can support other protocols (e.g., NTP, SSDP and memcached) by adding filters of the specific protocol and logic. Further research can explore other protocol extension.

RTT and thoughput

This chapter described the middlebox design which is assumed to place on the Internet path. Thus, this middlebox, mitiKV affected network latency. So, extended evaluation is measuring RTT and analized the impact of network sevices. Besides, the mitiKV can be evaluated whether mitiKV processes at line speed accurately.

Implementation

In this implementation, the version with 262k bit BRAM has a negative slack, which means target frequency was not matched. Some normal packets lost can be observed. Thus, the improvement of the implementation would be performed using new generation FPGA (e.g., ultrascale+ FPGA in Xilinx).

5.6 Summary

This chapter proposed a novel hardware-based DDoS mitigation system, called mitiKV, which focuses on DNS-based amplification attack. We focused on protocol behavior of the DNS-based amplification attack and the ICMP port unreachable message. The mitiKV detects and manages DDoS attacks on hardware-based key-value store. This chapter analyzed real-world traffic and showed that ICMP port unreachable messages have packet payload including required data for our protocol-based approach. A prototype system was implemented on an FPGA board to show that mitiKV can mitigate malicious traffic in 10GbE. Evaluation results demonstrated that mitiKV has capable of preventing DDoS packets on the Internet traffic with DDoS attack. We discussed the scalability of the proposed approach for further high throughput interfaces, such as 40GbE and 100GbE.

Chapter 6

Conclusions

6.1 Discussion

This dissertation proposed multi-layer key-value cache architecture using in-NIC and in-kernel caches, which can improve the performance by increasing hit ratio on lower level caches. In this thesis, simulation results showed that a variety of design options can be adopted in the cases on the proposed architecture.

This dissertation introduced the case for DDoS mitigation as an application of the proposed architecture. A prototype of DDoS mitigation middlebox was developed with on-chip RAM of an FPGA. While exteranl memory modules can be replaced with on-chip RAM and store more flows on the memory in order to support massive DDoS traffic, latency increases on data plane due to memory access latency to external memory modules.

In Chapter 4, level 0 cache which uses on-chip RAM of an FPGA to in-NIC cache, was introduced. In this prototype, 1024 entries were developed as level 0 cache due to design limitation. More entries can be expected in next generation FPGA technologies. Besides, two power management shemes were introduced: host-based controller and in-NIC based controller. Realistic workload experiments on the proposed implimentation using these power reduction schemes are important as future work.

The proposed architecture can be applied to not only DDoS mitigation but also message queuing system, real-time data streaming, domain name system, and so on. Note that it is important to have temporal locality to enable the proposed architecture since lower level cache has small capacity. Massive lower level cache capacity is expected to improve hit ratio on caches.

6.2 Concluding Remarks

In this dissertation, multi-layer key-value cache architecture using in-NIC cache and in-kernel caches was studied in order to bridge the growth of networking and of CPU performance.

Networking equipment evolution has increased rapidly, resulting in the gap between network

interface speed and CPU performance. Historically, cache hierarchy has been used in case we encounter the speed gap between CPU and memory. However, we have faced the gap between CPU and network. Chapter 3 introduced the concept of the network-based multi-layer cache hierarchy, which introduced in-NIC cache as the first level cache and in-kernel cache as the second level cache, and the implementation of cache hierarchy organization on an open source platform. Simulations were performed for the design options such as write policy, inclusive cache vs. non-inclusive cache and eviction policy and so on. These results imply that this architecture is effective for the gap mentioned above.

Chapter 4 introduced the proposed architecture implemented on an FPGA equipped on NIC and showed the architecture integrated with network datapath on NIC. Level 0 cache (on-chip RAM) is introduced in in-NIC cache design to reduce DRAM latency and to improve in-NIC cache's performance. When the query is hit on the level 0 cache, the latency and the performance turned out to be improved. Further, scheduling scheme for more energy efficiency was introduced in aspect of in-NIC traffic measurement and power measurement in a host machine.

Chapter 5 applied the proposed FPGA architectur eapplied to DDoS mitigation, which is one of the important Internet security issues. To detect DDoS flows quickly, an ICMP-based detection was introduced. Traffic analysis results indicated a capability of the detection scheme on the Internet backbone. In this chapter, hardware-based key-value store to mitigate DDoS traffic using an ICMP-based detection scheme was introduced. A prototype system was developed using hardware-based key-value store. Experiment results of DDoS testing in 10GbE line rate imply a capability of DDoS mitigation. To protect a massive flows, external memory modules are expected to work. In this way, it is expected that the proposed multi-layer key-value store could be applied for various applications.

The main contribution in this dissertation is exploring a variety of design space on multi-layer key-value cache architecture: inclusive vs. non-inclusive cache, write-through vs. write-back, associativities and eviction policies. In addition, in-NIC cache architecture combining on-board DRAM and on-chip memory was designed and explored in terms of low latency, high throughput, and power efficiency. As a result, it achieved higher performance than existing memcached design in terms of the three metrics. The case for an application using in-NIC cache design showed DDoS security box for DDoS mitigation system.

Possible future work is optimization of multi-layer key-value caches using in-NIC cache in the context of in-network computing, since FPGA takes power consumption even when query workload is low. Thus, we will explore highly energy efficient system with in-NIC cache using proper scheduling approach for the in-network computing.

Bibliography

[1] Xilinx. 7 Series FPGAs Memory Resources UG473(v1.12), Sep 2016.

[2] Luiz Andre Barroso and Urs Holzle. *The Datacenter as a Computer*. Morgan & Claypool Publishers, 2 edition, July 2013.

[3] Project Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/.

[4] Amazon EC2 F1 Instance. https://aws.amazon.com/jp/ec2/instance-types/f1/.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[6] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *NSDI*, pages 385–398. USENIX, 2013.

[7] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5 edition, 2012.

[9] NetFPGA Project. http://netfpga.org/.

[10] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An FPGA Memcached Appliance. In *FPGA*, pages 245–254, February 2013.

[11] Paramod J. Sadalarge and Martin Fowler. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, August 2012.

[12] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[13] E. S. Fukuda, H. Inoue, T. Takenaka, Dahoo Kim, T. Sadahisa, T. Asai, and M. Motomura. Caching Memcached at Reconfigurable Network Interface. In *FPL*, pages 1–6, September 2014.

[14] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Baer, and Zsolt Istvan. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *HotCloud*, June 2013.

[15] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*, pages 429–444, April 2014.

[16] Yuehai Xu, Eitan Frachtenberg, and Song Jiang. Building a High-performance Key-value Cache as an Energy-efficient Appliance. *Performance Evaluation*, 79:24–37, September 2014.

[17] Low latency RPC in RAMCloud.

[18] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA*, pages 13–24, June 2014.

[19] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *ISCA*, pages 36–47, June 2013.

[20] Michaela Blott and Kees Vissers. Dataflow Architectures for 10Gbps Line-rate Key-value-Stores. In *HotChips*, August 2013.

[21] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based In-Line Accelerator for Memcached. *IEEE Computer Architecture Letters*, 13(2), July 2014.

[22] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database Analytics Acceleration Using FPGAs. In *PACT*, pages 411–420, September 2012.

[23] Jae Min Cho and Kiyoung Choi. An FPGA Implementation of High-throughput Key-value Store Using Bloom Filter. In *VLSI-DAT*, pages 1–4, April 2014.

[24] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. A Flexible Hash Table Design for 10Gbps Key-value Stores on FPGAs. In *FPL*, pages 1–8, September 2013.

[25] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory. In *HotStorage*, July 2015.

[26] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, pages 51–66. USENIX Association, 2018.

[27] N. Zilberman, Y. Audzevich, G.A. Covington, and A.W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, September 2014.

[28] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, 2016.

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *SOSP*, pages 137–152. ACM, 2017.

[30] Danga Interactive. Memcached - A Distributed Memory Object Caching System. http://memcached.org/.

[31] J.W. Lockwood and M. Monga. Implementing Ultra Low Latency Data Center Services with Programmable Logic. In *HoT Interconnects*, pages 68–77, Aug 2015.

[32] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating System Review*, 43(4):92–105, January 2010.

[33] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, pages 371–384, 2013.

[34] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *ATC*, pages 57–69, July 2015.

[35] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *ISCA*, pages 476–488, June 2015.

[36] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *ATC*, pages 103–114, June 2013.

[37] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Security Symposium (Security'12)*, pages 101–112, August 2012.

[38] Netfilter. https://www.netfilter.org/.

[39] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 42:1–42:13, New York, NY, USA, 2018. ACM.

[40] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, pages 53–64, June 2012.

[41] Salvatore Sanfilippo. Redis. http://redis.io/.

[42] David de la Chevallerie, Jens Korinth, and Andreas Koch. ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators. In *HEART'15*, June 2015.

[43] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, pages 121–136. ACM, 2017.

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*, pages 35–49. USENIX Association, 2018.

[45] Tu Dang Huynh, Pietro Bressana, Suh Lee Ki Wang, Han, Marco Weatherspoon, Hakimand Canini, Fernando Pedone, Noa Zilberman, and Robert SoulÃl'. P4xos: Consensus as a network service. *Technical Report University of Lugano*, May 2018.

[46] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *SOSR*, page 5. ACM, 2015.

[47] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, page 10. ACM, 2018.

[48] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *HOTNETS*, pages 150–156. ACM, 2017.

[49] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W. Moore. Where has my time gone? In *Passive and Active Measurement*, pages 201–214, 2017.

[50] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, New York, NY, USA, 1997. ACM.

[51] Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, Paolo Costa, Peter Pietzuch, Jon Crowcroft, Andrew W Moore, and Noa Zilberman. Emu: Rapid Prototyping of Networking Services. In *ATC*, pages 459–471. USENIX Association, 2017.

[52] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Mckeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski. OSNT: open source network tester. *IEEE Network*, 28(5):6–12, September 2014.

[53] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[54] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/, 2018.

[55] Jeff Mogul and Jitu Padhye. In-Network Computation is a Dumb Idea Whose Time Has Come HotNets-XVI Dialogue. https://conferences.sigcomm.org/hotnets/2017/dialogues/dialogue140.pdf, 2017.

[56] System Artware. *SHW 3A Watt hour meter*. http://www.system-artware.co.jp/shw3a.html.

[57] Mellanox. *Mellanox Spectrum vs Broadcom and Cavium*. http://www.mellanox.com/img/products/switches/Mellanox-Spectrum-vs-Broadcom-and-Cavium.png[Online, accessed May 2018.

[58] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. *IEEE/ACM Trans. Netw.*, 25(3):1593–1606, 2017.

[59] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[60] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, (12):33–37, 2007.

[61] Daniel Wong. Peak efficiency aware scheduling for highly energy proportional servers. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 481–492. IEEE, 2016.

[62] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010.

[63] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *ACM SOSP*, pages 15–28. ACM, 2009.

[64] Hu Li. Introducing "Yosemite": the first open source modular chassis for high-powered microservers. https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-\high-powered-microservers-/, 2015. [Online; accessed May 2018].

[65] Mellanox. ConnectX-6 EN single/dual-port adapter supporting 200Gb/s Ethernet. http://www.mellanox.com/page/products_dyn?product_family=266&mtag=connectx_6_en_card, 2018. [Online; accessed May 2018].

[66] Sujal Das. The arrival of SDN 2.0: SmartNIC performance, COTS server efficiency and open networking. https://www.netronome.com/blog/the-arrival-of-sdn-20-smartnic-performance-cots-server-efficiency-and-open-networking/, 2016. [Online; accessed May 2018].

[67] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *Proc. of the USENIX NSDI 2018*, pages 51–66, 2018.

[68] Netcope Technologies. *Netcope unveils Netcope P4 - a breakthrough in smart NIC performance and programmability*. https://www.netcope.com/en/company/press-center/press-releases/netcope-unveils-np4-a-breakthrough-in-smartnic [Online, accessed September 2018.

[69] Napatech. *Napatech SmartNIC for Virtualization Solutions*. https://www.napatech.com/products/napatech-smartnic-virtualization/ [Online, accessed September 2018.

[70] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Security 17*, pages 1093–1110, Vancouver, BC, 2017.

[71] Akamai. State of the internet / security: Web attacks. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-summer-2018-web-attack-report.pdf.

[72] Brianna Boudreau. Global Bandwidth & IP Pricing Trends. http://www2.telegeography.com/hubfs/2017/presentations/telegeography-ptc17-pricing.pdf.

[73] Open Resolver Project. http://openresolverproject.org/.

[74] Arbor Networks. Worldwide infrastructure security report. https://pages.arbornetworks.com/rs/082-KNA-087/images/12th_Worldwide_Infrastructure_Security_Report.pdf.

[75] Jelena Mirkovic and Peter Reiher. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. *SIGCOMM Computer Communication Review*, 34(2):39–53, April 2004.

[76] Yoohwan Kim, Wing Cheong Lau, Mooi Choo Chuah, and H. J. Chao. PacketScore: a Statistics-based Packet Filtering Scheme Against Distributed Denial-of-service Attacks. *IEEE Transactions on Dependable and Secure Computing*, 3(2):141–155, April 2006.

[77] Snort. Snort - network intrusion detection and prevention system. https://www.snort.org/.

[78] H. J. Chao, R. Karri, and Wing Cheong Lau. CYSEP - a cyber-security processor for 10 Gbps networks and beyond. In *Proceedings of Military Communications Conference*, volume 2, pages 1114–1122 Vol. 2, Oct 2004.

[79] P. Djalaliev, M. Jamshed, N. Farnan, and J. Brustoloni. Sentinel: Hardware-Accelerated Mitigation of Bot-Based DDoS Attacks. In *Proceedings of 17th International Conference on Computer Communications and Networks*, pages 1–8, Aug 2008.

[80] K. Pandiyarajan, S. Haridas, and K. Varghese. Transparent FPGA based device for SQL DDoS mitigation. In *Proceedings of Internation Coneference on Field-Programmable Technology*, pages 82–89, Dec 2013.

[81] Luis Von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: Using Hard AI Problems for Security. In *Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'03, pages 294–311, 2003.

[82] N. Hoque and H. Kashyap and D.K. Bhattacharyya. Real-time DDoS attack detection using FPGA. *Computer Communications*, 110:48–58, 2017.

[83] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. Gaspp: A gpu-accelerated stateful packet processing framework. In *Proceedings of the USENIX ATC)*, pages 321–332, June 2014.

[84] Yuta Tokusashi and Hiroki Matsutani. Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches. *IEEE Micro*, 37(5):44–51, 2017.

[85] Renesas. Network Search Engine. https://www.renesas.com/ja-jp/products/memory/network-search-engine.html.

[86] Jon Postel et al. RFC 792: Internet control message protocol. *InterNet Network Working Group*, 1981.

[87] D. R. Thomas, R. Clayton, and A. R. Beresford. 1000 days of UDP amplification DDoS attacks. In *Proceedings of the 2017 APWG Symposium on Electronic Crime Research (eCrime)*, pages 79–84, April 2017.

[88] Alex Conta, Stephen E Deering, and Mukesh Gupta. Ed. RFC 4443: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. *Network Working Group*, 2006.

[89] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the wide project. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 51–51, 2000.

[90] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Mckeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski. OSNT: open source network tester. *IEEE Network*, 28(5):6–12, September 2014.

[91] SINET5. Worldwide infrastructure security report. https://www.sinet.ad.jp/.

Publications

Related Papers

Journal Papers

[1] Yuta Tokusashi, Hiroki Matsutani, Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches, IEEE Micro, vol.37, No.5, pp.44-51, September/October 2017.

[2] Yuta Tokusashi, Hiroki Matsutani, Design of Key-Value Store Appriance Having a Variety of Data Strcture, IPSJ Journal, Vol.56, No.8 pp.1787-1799, Aug 2016. (In Japanese)

International Conference Papers

[3] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman, LaKe: The Power of In-Network Computing, *Proc. of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'18)*, Dec 2018.

[4] Yuta Tokusashi, Yohei Kuga, Ryo Nakamura, Hajime Tazaki, and Hiroki Matsutani mitiKV: An Inline Mitigator for DDoS Flooding Attacks *Proc of Internet Conference 2016*, Oct 2016.

[5] Yuta Tokusashi, and Hiroki Matsutani, A Multilevel NOSQL Cache Design Combining In-NIC and In-Kernel Caches, *Proc. of the 24th IEEE International Symposium on High Performance Interconnects (Hot Interconnects'16)*, pp.60-67, Aug 2016.

[6] Yuta Tokusashi, and Hiroki Matsutani, NOSQL Hardware Appliance with Multiple Data Structures, *Proc. of the 28th IEEE Symposium on High Performance Chips (Hot Chips 28)*, Poster session, Aug 2016

Domestic Conference Papers, Technical Reports and Posters

[7] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman, LaKe: An Energy Efficient, Low Latency, Accelerated Key-Value Store *Arxiv preprint 2018*, May 2018.

[8] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman, LaKe: An Energy Efficient, Low Latency, Accelerated Key-Value Store, *Eurosys Doctoral Workshop 2018*, Apr 2018.

[9] Yuta Tokusashi, Hiroki Matsutani, A Cache Hierarchy in Kernel and NIC for NOSQL Acceleration. *IEICE Technical Reports CPSY2015*, Vol.115, No.174, pp.185-190, Aug 2015. (In Japanese)

[10] Yuta Tokusashi, and Hiroki Matsutani, A Case for Accelerating Data Structure Sever using FPGA NIC, *IEICE Technical Reports CPSY2014-162 (ETNET'15)*, Vol.114, No.506, pp.1-6, Mar 2015. (In Japanese)

Other Papers

Journal Papers

[11] Koya Mitsuzuka, Yuta Tokusashi, and Hiroki Matsutani, Efficient Message Queuing System Using FPGA-Based 10GbE Switch, IPSJ Journal, Vol.59, No.9, pp.1446-1463, Aug 2018. (In Japanese)

[12] Ami Hayashi, Yuta Tokusashi, and Hiroki Matsutani, A Nonparametric Online Outlier Detector for FPGA NICs, IPSJ Journal, Vol.56, No.8 pp.1664-1679, Aug 2016. (In Japanese)

[13] Ami Hayashi, Yuta Tokusashi, and Hiroki Matsutani, A Line Rate Outlier Filtering FPGA NIC using 10GbE Interface, ACM SIGARCH Computer Architecture News (CAN), Vol.43, No.4, pp.22-27, Sep 2015.

[14] Yuta Tokusashi, Yohei Kuga, Takeshi Matsuya, and Osamu Nakamura, Design and Implementation of An FPGA-Based Low-Latency HDMI Video Synchronization System, IPSJ Journal, Vol.56, No.8 pp.1593-1603, Aug 2015. (In Japanese)

International Conference Papers

[15] Yuma Sakakibara, Yuta Tokusashi, Shin Morishima, Hiroki Matsutani, Accelerating Blockchain Transfer System Using FPGA-Based NIC *Proc. of the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'18)*, Dec 2018.

[16] Kaho Okuyama, Yuta Tokusashi, Takuma Iwata, Mineto Tsukada, Kazumasa Kishiki, Hiroki Matsutani, Network Optimizations on Prediction Server with Multiple Predictors, *Proc. of the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'18)*, Dec 2018.

[17] Koya Mitsuzuka, Yuta Tokusashi, Hiroki Matsutani, MultiMQC: A Multilevel Message Queuing Cache Combining In-NIC and In-Kernel Memories, *Proc. of the FPT 2018*, Dec 2018.

[18] Takuma Iwata, Kohei Nakamura, Yuta Tokusashi, and Hiroki Matsutani, Accelerating Online Change-Point Detection Algorithm using 10GbE FPGA NIC, *Proc. of the 24th International European Conference on Parallel and Distributed Computing (Euro-Par'18) Workshops, The*

16th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'18), Aug 2018.

[19] Ami Hayashi, Yuta Tokusashi, and Hiroki Matsutani, A Line Rate Outlier Filtering FPGA NIC using 10GbE Interface, *Proc. of the 6th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'15)*, Jun 2015. **Best Paper Award**

Domestic Conference Papers and Technical Reports

[20] Yosuke Yanai, Takeshi Matsuya, Yohei Kuga, Yuta Tokusashi, and Jun Murai, Proposition and Implementation of RISC-V Processor with Data path extension for 10G Ethernet, *IEICE Technical Reports CPSY2018*, Vol.118, No.165, pp.33-38, Jul 2018. (In Japanese)

[21] Takuma Iwata, Koya Mitsuzuka, Kohei Nakamura, Yuta Tokusashi, and Hiroki Matsutani, Accelerating Serialization Protocols for Network-Attached FPGAs, *IEICE Technical Reports CPSY2018*, Vol.117, No.378, pp.139-144, Jan 2018. (In Japanese)

[22] Mineto Tsukada, Koya Mitsuzuka, Kohei Nakamura, Yuta Tokusashi, and Hiroki Matsutani, Accelerating Sequential Learning Algorithm OS-ELM Using FPGA-NIC, *IEICE Technical Reports CPSY2018*, Vol.117, No.378, pp.133-138, Jan 2018. (In Japanese)

[23] Korechika Tamura, Ami Hayashi, Yuta Tokusashi, and Hiroki Matsutani, Accelerating NOSQLs using FPGA NIC and In-Kernel Key-Value Cache, *IEICE Technical Reports CPSY2014-123*, Vol.114, No.427, pp.7-12, Jan 2015. (In Japanese)

[24] Ami Hayashi, Yuta Tokusashi, and Hiroki Matsutani, An Online Outlier Detector for FPGA NICs, *IEICE Technical Reports CPSY2014-124*, Vol.114, No.427, pp.13-18, Jan 2015. (In Japanese)

[25] Yuta Tokusashi, Takeshi Matsuya, Yohei Kuga, and Jun Murai, Improving the Naturalness of Internet Video Conversation using a Low-Latency Pipeline, *Proc. of Multimedia, Distributed, Cooperative and Mobile Symposium (DICOMO'13)*, Vol.2013, pp911-917, Jul 2013. (In Japanese)