National Tsing Hua University

Master Thesis

Performance Improvement of OpenFlow Switch with Queue and Per-port Cache

Author:
Yu-Chung Cheng

Advisor:
Professor Youn-Long Lin

A thesis submitted in fulfilment of the requirements
for the degree of Master of Computer Science
in the
Department of Computer Science

November 2016

Abstract

Software-Defined Networking (SDN) separates control planes from distributed network devices. For the popular white box SDN platforms, we propose a Queue-based Non-blocking Per-Port Cache design to increase switch throughput. We implement it with the NetFPGA Network Interface Card and the Open vSwitch software search engine. We compare our prototype with a blocking cache design. Experiment results show that the proposed design achieves high throughput.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The legacy network employs a distributed control architecture. Each network device has its control plane and data plane. If we want to add a new network function, we have to replace or update all devices' control planes. Besides, it is hard to modify a device or implement a new function without vendors help since the network is a closed platform.

On the other hand, a Software-Defined Networking (SDN) [1] system decouples control plane from network devices into a centralized controller. Therefore, it can be easily updated or

added a function through the programmable controller, such as OpenDaylight [2], Floodlight [3] and, POX [4].

In an SDN system, the controller controls devices through the OpenFlow protocol [5]. The protocol defines an OpenFlow switch as an SDN data plane device. The switch searches a flow table for flow actions when receiving a packet. If there is a match, it forwards out the packet; otherwise, it sends an OpenFlow message to a controller to ask for how to handle the packet. Then, the controller will reply a flow action to the switch and the flow action will be inserted into the switch's flow table.

There are two methods of building an OpenFlow switch. The first one uses a hardware search engine to build OpenFlow switch, e.g., BlueSwitch [6] and RMT [7]. The hardware switch implements its flow table using TCAM [8] or RAM [9] for wire-speed table lookup and forwarding. The second method employs a software search engine (SSE), e.g., OpenSwitch [10] and Open vSwitch [11]. The software switch implements its flow table as a software-based hash table. Although the first method achieves higher throughput, the second one is more flexible and can support larger flow table.

A White box switch [12] has the advantages of hardware high-speed forwarding and software flexibility. The switch employs a Linux-based Open vSwitch [11] as the SSE and a Network Interface Card (NIC) [13] as the forwarding accelerator. For improving the switch throughput, Reference [14]

adds a blocking per-port cache to a white box switch. With the cache, the white box switch can take advantage of packet traffic's spatial locality to reduce the number of SSE operations and thus achieves higher throughput. However, the blocking cache scheme suffers from low cache utilization and hence limits the degree of performance improvement.

In this thesis, we propose a Queue-based Nonblocking Per-Port Cache (Q-NPPC). It achieves higher throughput due to high cache utilization. We insert a queue to compensate for the nonblocking cache's adverse effect by storing subsequent cache miss packets. The queue also helps in reducing the number of cache updates and therefore increasing the hit rate.

The remainder of this thesis is organized as follows. Chapter 2 surveys some related work. Chapter 3 describes our

proposed architecture. Chapter 4 presents our simulation and evaluation results. Finally, we conclude and point to possible directions for future research in Chapter 5.

Chapter 2

Related Work

We first review the Open vSwitch architecture in Section 2.1. Section 2.2 explains temporal locality and spatial locality of network traffic. Section 2.3 describes a typical white box switch. Section 2.4 describes a control-message quenching algorithm. Section 2.5 presents the per-port blocking cache and discusses its problem.

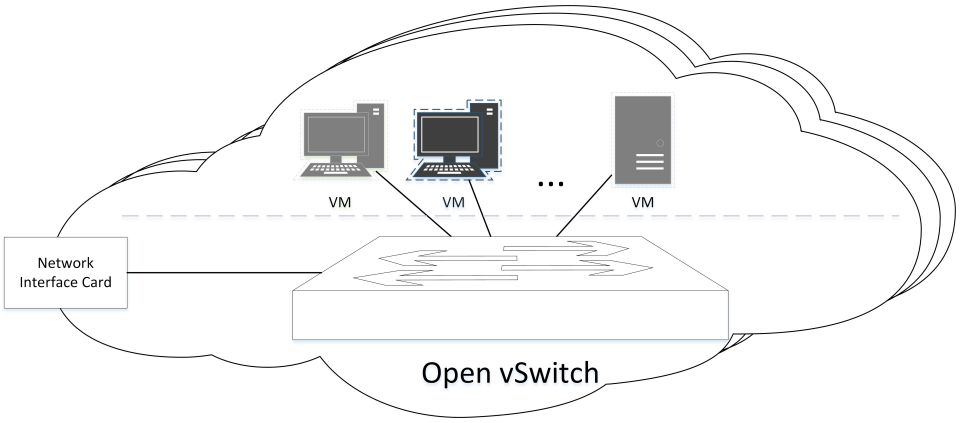


FIGURE 2.1: Open vSwitch

2.1 Open vSwitch

An Open vSwitch (OVS) [11] as depicted in Figure 2.1 is a virtual switch for massive network automation under the Apache 2.0 license [15]. It supports security, monitoring, QoS and automated control across multiple virtual machines and physical servers. OVS's automated control follows the OpenFlow protocol as following:

Exact Match Lookup: The OVS looks up its flow table when a packet arrives. It performs associated action if a match in found; otherwise, it sends out an UPCALL.

UPCALL: An UPCALL will look up a wildcard flow table in the user space. The packet will follow the OpenFlow protocol to communicate with an SDN controller when a miss happens; otherwise, it will perform the associated action.

Communicate With Controller: The switch sends a *packet_in* message to the controller requesting for flow action. The controller will reply with a *packet_out* message and a *flow_mod* message to be inserted into the flow table.

2.2 Network Traffic Locality

In a network, we can observe temporal locality in the flow and spatial locality in switch ports. Temporal locality means that an item is promising to be reused again in a short time. Spatial locality means that nearby items will be referenced to soon.

Chiueh and Pradhan [16] showed that temporal locality is an important characteristic in network flow. They designed an Intelligent Host Address Range Cache (IHARC) for fast table lookup and low miss rate.

Reference [14] reported that network flow exhibit spatial locality. Same-source-IP packets tend to enter the switch via the same port. They proposed a Per-Port Cache for reducing average flow table access time and increasing cache hit rate.

2.3 White Box Switch

A white box switch [12] [17] serves as a data plane device in an SDN network. Figure 2.2 depicts our white box switch platform. It consists of a Network Processing Unit (NPU) for output port lookup, a Central Processing Unit (CPU)

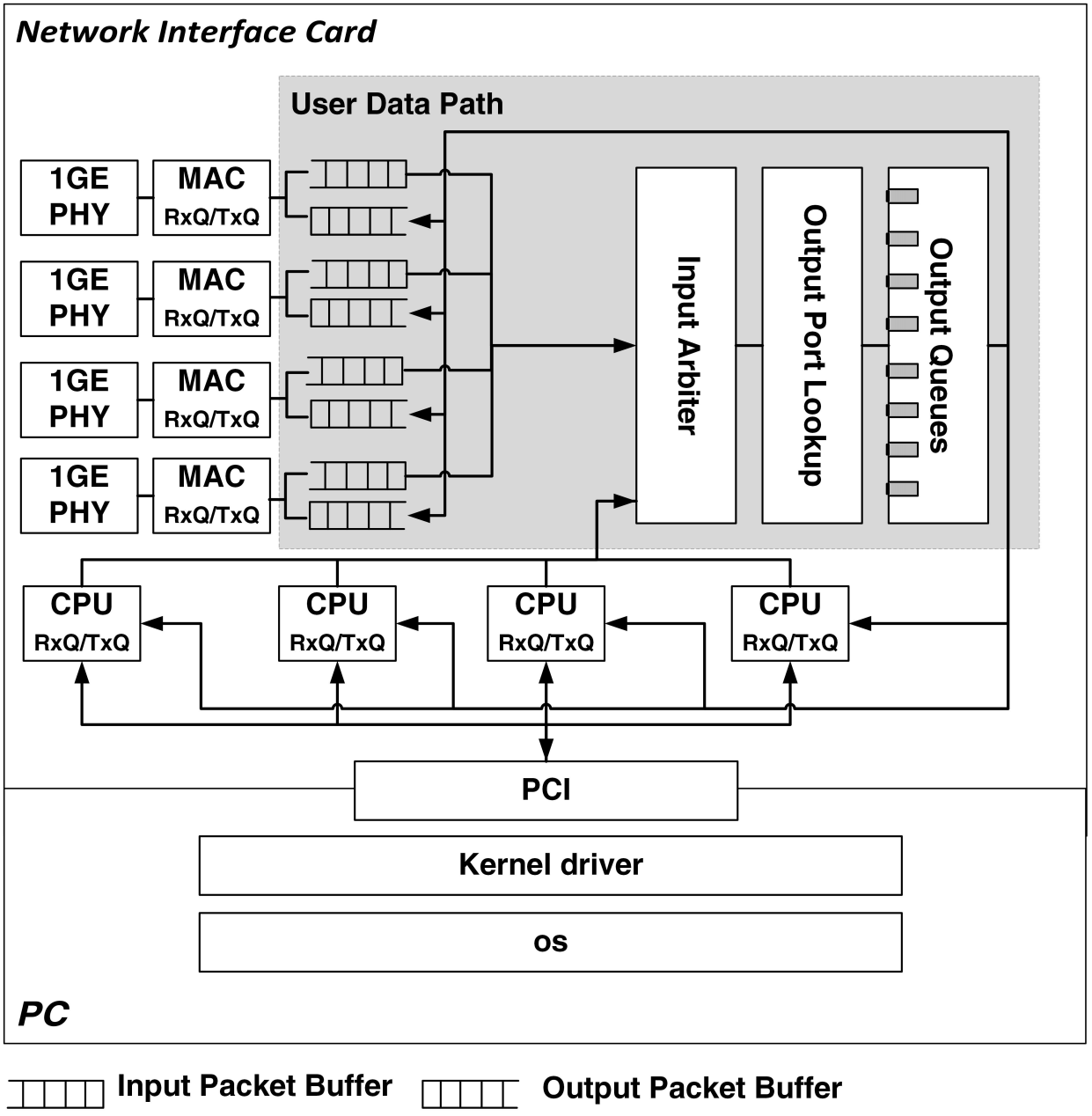


FIGURE 2.2: White box switch architecture

for running Linux OS, and an Open vSwitch software search engine (SSE).

The switch sends and receives packets through 1GB PHY and MAC ports. An input arbiter fetches packets from the ports based on a round-robin scheme. The NPU chooses a destination port for each packet according to its metadata. The output queue forwards packets to the output ports or the PC. If the packet is destined for the PC, it will go through CPU RxQ and PCI [18]. The CPU RxQ and CPU TxQ are the internal ports for sending packets to and receiving packets from the PC's CPU. Finally, the PC CPU runs the SSE to find the required actions and sends the actions back to NIC through CPU TxQ and PCI.

2.4 Control-Message Quenching

Reference [19] observed large overhead for an SDN controller to serve switch requests and proposed a Control-Message

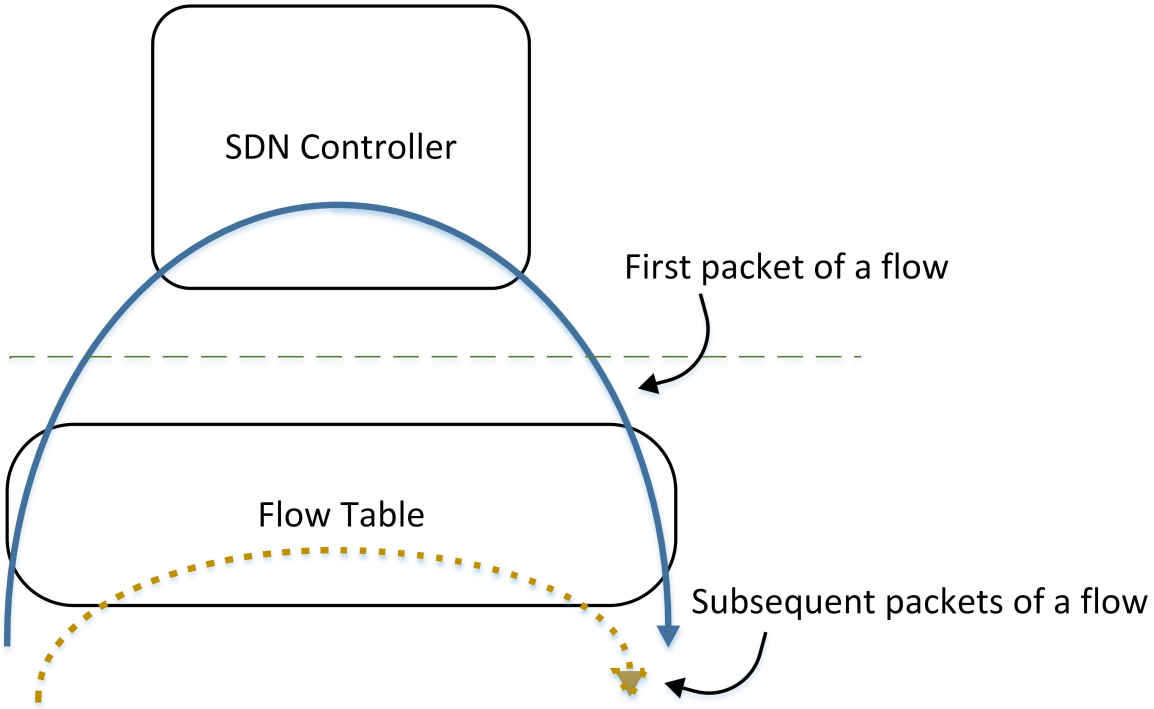


FIGURE 2.3: The Control-Message Quenching

Quenching (CMQ) algorithm to reduce the number of requests. Subsequent packets are the second or later packets of the same flow. The CMQ algorithm uses a queue to store table-miss subsequent packets.

The CMQ queue can store almost all subsequent packets of a flow due to temporal locality. Figure 2.3 shows that CMQ sends the first packet to the controller to request for a flow action and holds subsequent packets in the flow table. After

the controller replies, it updates the flow table and sends subsequent packets to an output port.

2.5 Accelerating OpenFlow Switches with Per-port Blocking Cache

Reference [14] uses per-port blocking cache to boost switch performance. The cache reduces the number of SSE requests as it will stall until the SSE finishes updating a cache miss. Because subsequent packets cannot access the cache, they will not induce redundant cache miss. Although the scheme gives a shorter average flow table access time, it may increase the miss penalty and the probability of dropped packets.

Chapter 3

Proposed Methodology

We propose a Queue-based Nonblocking Per-Port Cache (Q-NPPC) architecture and apply it to a white box switch design to achieve higher throughput. Section 3.1 presents the essential of nonblocking cache. In Section 3.2, we describe our switch design. The Q-NPPC implementation is presented in Section 3.3.

3.1 Nonblocking Cache Problems

A nonblocking cache often suffers from low hit rate due to flow' temporal locality. We use the following to explain. First, a compulsory miss occurs when the first packet of a flow accesses the cache. While the first packet triggers sending requests to the SSE, subsequent packets will also access the cache and then result in a series of misses. Therefore, the cache has low hit rate. Second, the low hit rate causes a lot of requests from the same flow being sent to the SEE. The cache frequently updates the same flow. However, only the first update will affect the cache entry while the remaining updates are redundant. If there is another flow packet trying to access the cache, a cache miss will happen.

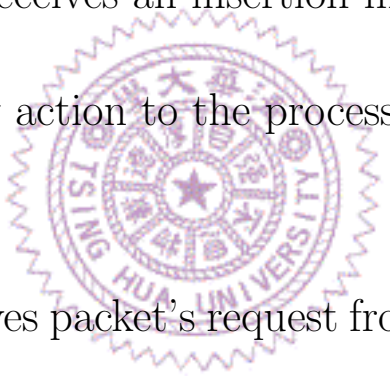
Reference [20] deals with the problem using a Miss-status Handling Registers (MSHR). Unfortunately, MSHR is incompatible with our switch design.

Instead, we call for a queue to handle cache miss packets. The queue stores subsequent cache-miss packets until the SSE updates the flow action in the cache. It can reduce the number of SSE requests and enable the port processor to write different flow actions into the cache earlier.

3.2 Switch Design Framework

As depicted in Figure 3.1, we modify a NetFPGA NIC [13] to be our forwarding platform and use an Open vSwitch (OVS) to perform Software Search Engine (SEE) function.

The switch equips each port with a processor and a cache. The processor receives packets and accesses the cache. Then, it writes some metadata and sends the packet to both the input arbiter and an output port. According to the metadata, the output port's lookup module will send the packet to the OVS or the Output Packet Buffer. In the meantime, the insertion engine receives an insertion message from the OVS and sends an flow action to the processor for cache update.



The OVS receives packet's request from the port processor. Then, it searches for flow action in its flow table. If a flow action is found, it will send out an insertion messages including flow action and flow metadata to the NIC' insertion engine; Otherwise, it will request flow action from the SDN controller and send the action and metadata to the insertion engine.

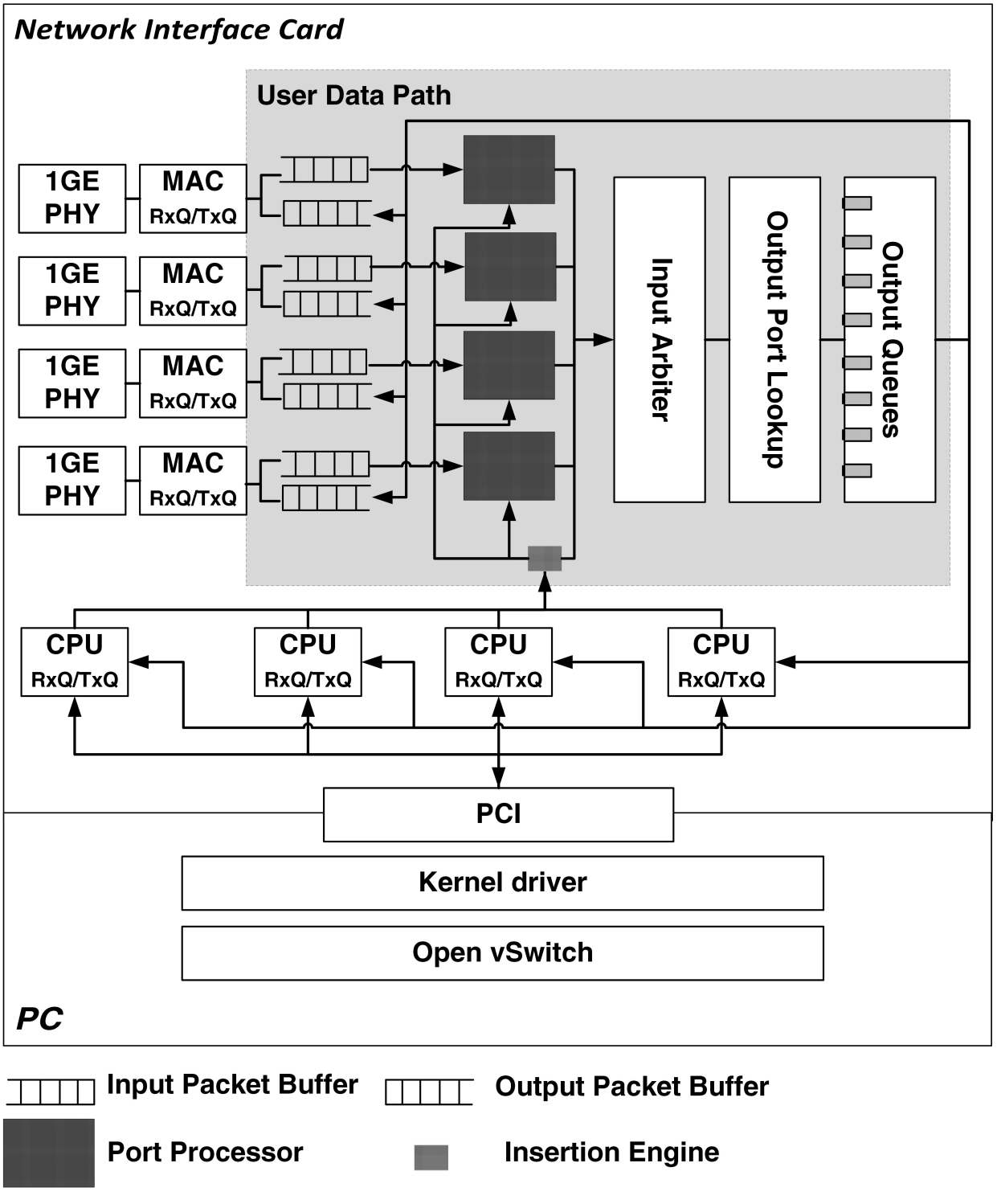


FIGURE 3.1: White box switch with nonblocking per-port cache

3.3 Q-NPPC Implementation

Figure 3.2 depicts our Q-NPPC port processor block diagram. The port processor receives insertion message from the insertion engine and stores flow action into the cache. It writes packet metadata and forwards packets to the output port's lookup module. If a cache miss occurs and the packet is the first packet of the flow, the processor will send the packet to the OVS to request for a flow action; Otherwise, it will store subsequent packets of the flow into the queue.

Table 3.1 defines the actions associated with all control signals of Figure 3.2 while Figure 3.3 depicts the flow chart of how our port processor.

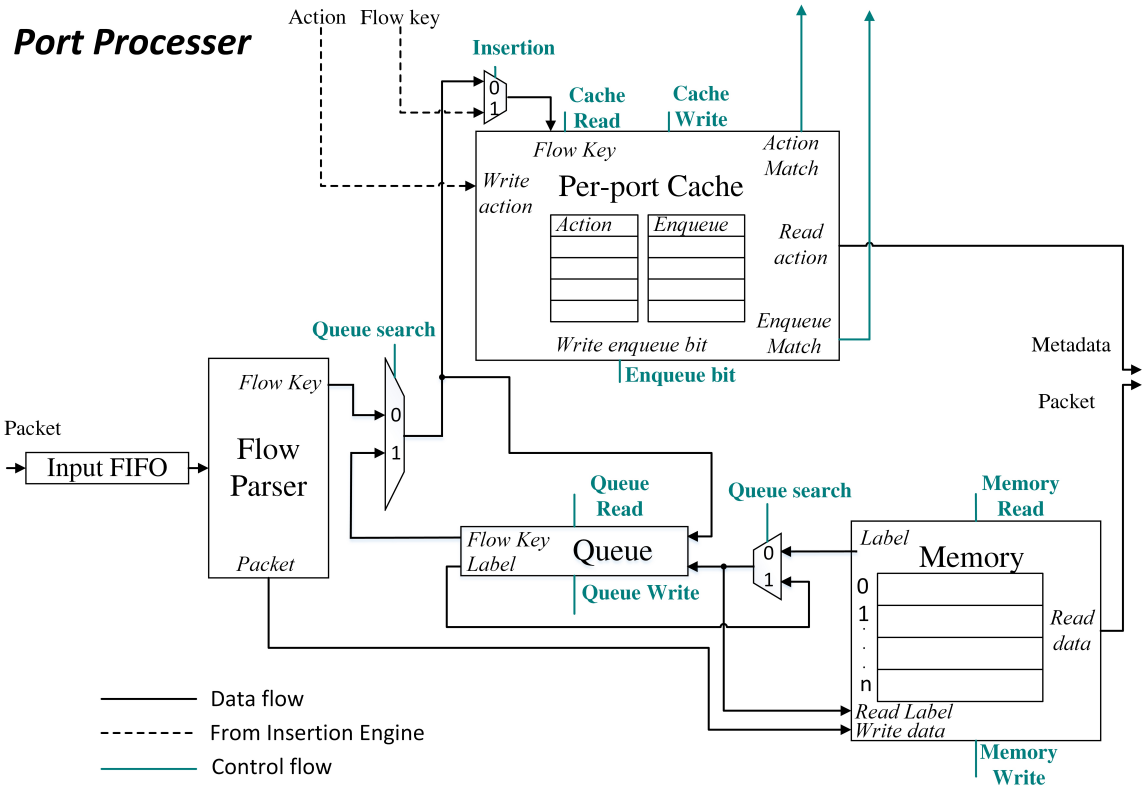


FIGURE 3.2: Nonblocking port processor architecture

Read Input FIFO: If the port processor does not receive any insertion message, the insertion signal is set to 0. When the insertion signal is equal to 0, the port processor will read a packet from the Input Packet Buffer and writes the packet into the input FIFO. Then the processor reads a packet from the input FIFO and sends it to the flow parser.

TABLE 3.1: The control signals and actions of the processor

Control signals	The signal is zero	The signal is one
Insertion	The processor does not receive any insertion message from the insertion engine.	The processor receives an insertion message from the insertion engine, raises the insertion signal and gives a flow action to the cache.
Queue_search	The per-port cache gets flow_key from the flow parser.	The per-port cache gets a flow_key from the queue.
Enqueue_bit	X	The processor activates the enqueue_bit when the first packet of the flow accesses the cache.
Action_match	The flow action is not in the cache.	The flow action is in the cache.
Enqueue_match	The flow enqueue_bit is equal to zero.	The flow enqueue_bit is equal to one.
	Read	**Write**
Queue read/write	The queue gives a flow_key and an address label to the processor.	The queue stores a flow_key and an address label.
Memory read/write	The memory gives a packet to the processor according to address label.	The packet is stored in the memory. The memory gives an address label to the processor.
Cache read/write	The processor uses a flow_key to access the cache. Then, the cache gives results to the processor.	The processor uses a flow_key and a flow action to update the cache. The flow action is acquired from insertion massage.

Fetch Flow Key: The flow parser receives packets from the input FIFO, extracts OpenFlow keys from the packet headers, and then set the queue_search signal to 0. Then, the parser will send the flow_key to the per-port cache to find the flow action. Table 3.2 lists all flow_key entries defined in the

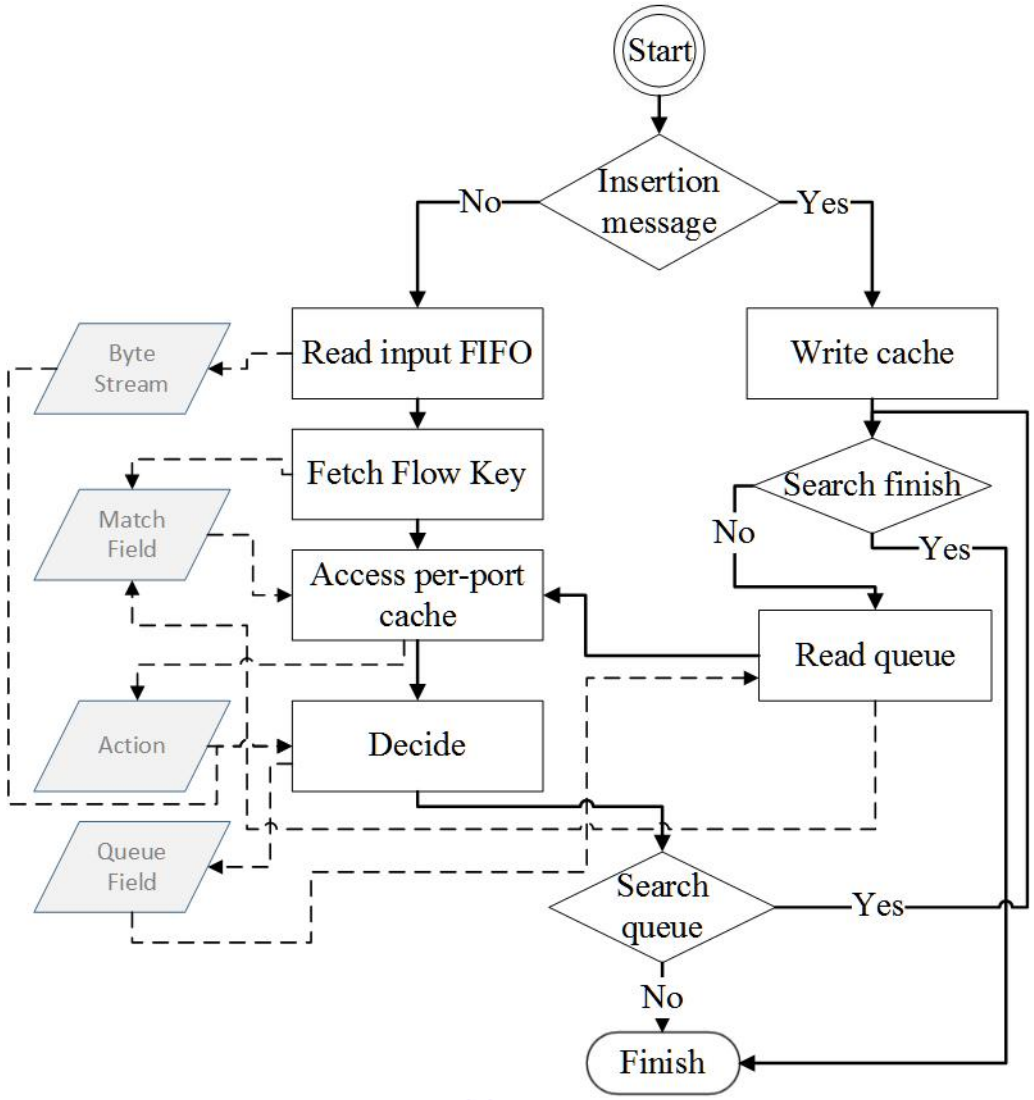


FIGURE 3.3: Port processor flow chart

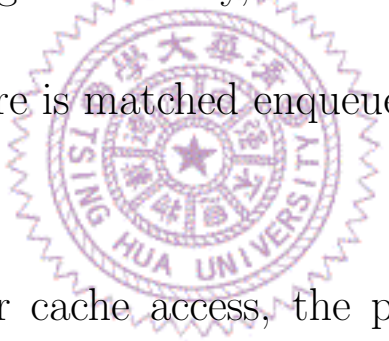
OpenFlow standard 1.0 [5].

Access Per-port Cache: The per-port cache receives a flow_key and finds corresponding flow action in the cache. If there is a matched flow action in the cache, the action_match

TABLE 3.2: The control signals of the processor

Field Name	**Field width (bits)**
Switch ingress port	32
Metadata	64
MAC Src	48
MAC Dst	48
Ether type	16
VLAN ID	12
VLAN Priority	3
MPLS	20
MPLS traffic class	3
IPv4 src	32
IPv4 dst	32
IPv4/ARP	8
IPv4 ToS bits	6
ICMP type / Transport src port	16
ICMP Code / Transport dst port	16

signal is set to high. Similarly, the enqueue_match signal is set to high, if there is matched enqueue_bit in the cache.



Decide: After cache access, the processor decides how to handle the packet according to the flow chart depicted in Figure 3.4. If the action_match signal is high, the processor forwards a packet to the input arbiter; Otherwise, it checks the enqueue_match flag to determining whether the packet is the first of the flow. For a subsequent packet, the processor writes the address label and the flow_key into the queue.

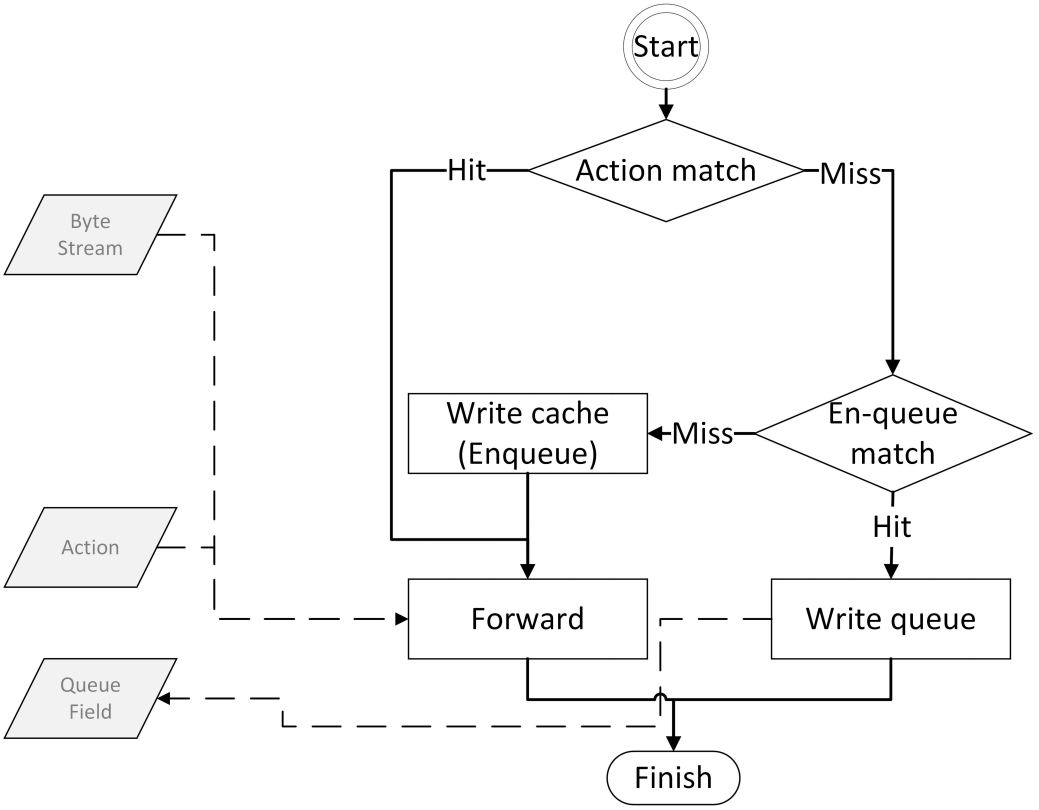


FIGURE 3.4: Decision module flow chart

If the enqueue_match is low, the processor activates the enqueue_bit in the cache and forwards the packet to the OVS to request for a flow action.

Write Cache: If the processor receives an insertion message, the insertion signal is set to high. Then, the port processor writes a flow action into the cache. When finishing writing cache, the insertion signal is set to 0.

Read Queue: After updating the cache, the processor reads flow_keys from the queue and raises the queue_search signal. If the enqueue_search signal is set high, the queue will send the flow_key to per-port cache to find the flow action.

Finish: If the search_queue signal is low or the port processor has finished searching the queue, the decision module terminates itself.

Chapter 4

Evaluation

We simulate our design and analyze some experiment results. Section 4.1 gives our experiment considerations. Section 4.2 describes our experiment environment. Section 4.3 presents and analyzes our results.

4.1 Experiment Considerations

Controller: The controller affects the flow behavior. Different routing policies result in different cache hit rates. We

employ the POX controller [4], which supports both spanning tree and random routing path policies.

Traffic pattern: We shape our traffic characteristic according to Benson and Anand's work [21] focusing on the packet inter-arrival time and ON/OFF period. The traffic characteristic is formulated according to the lognormal distribution in Eq. 4.1. To set the parameters between 1 and 10 as suggested by Kaudula and Sengupta [22], we set the mean μ to 0.9 and standard deviation σ to 2.3. Note that, x is a random positive variable, $erfc$ is complementary error function [23] and Φ is the cumulative distribution function [24].

$$\frac{1}{2}erfc\Big(-\frac{\ln x-\mu}{\sigma\sqrt{2}}\Big)=\Phi\Big(\frac{\ln x-\mu}{\sigma}\Big) \tag{4.1}$$

Topology: We use the Equal Cost Multi Path (ECMP) topology, commonly found in data centers, as our simulation

topology.

Co-simulation: During co-simulation, the speed of simulating hardware part [25] is very slow compared to that of software. We slow down the software OVS via Mininet API [26] to cope with this timing inconsistency issue.

4.2 Experiment Environment

The environment is same as Reference [14] for fair comparison. Our computer runs Ubuntu OS [27] equipped with an Intel 2.8GHz core i7 and 32GB of DRAM. The network simulator is Mininet 2.0 [26].

Figure 4.1 shows our experiment setup. A set of Open vSwitches is connected in the Equal Cost Multi Path (ECMP)

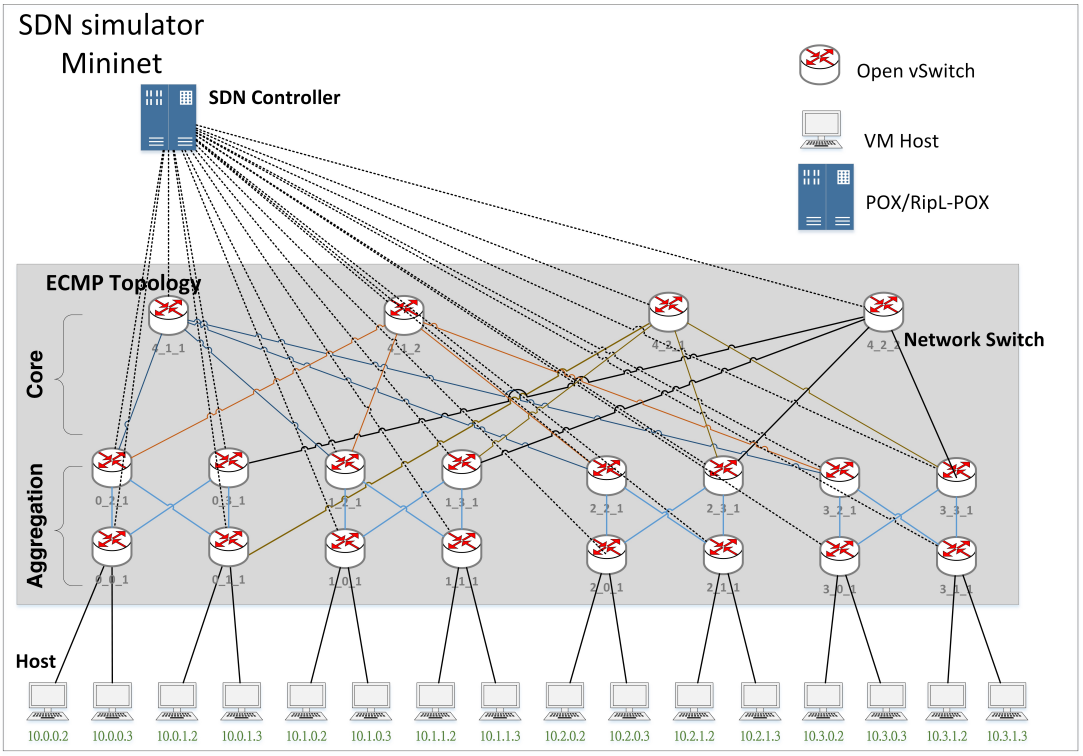


FIGURE 4.1: The experiment environment and topology

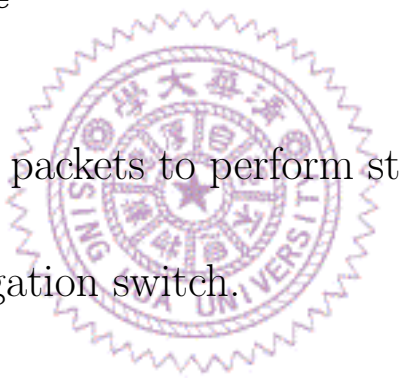
topology. The POX controller controls switches while the RipL-POX [28] controls network flow.

We use the Hedera [29] benchmark flow scheduler and iperf [30] with NumPy API [31] to generate traffic flow. The Hedera scheduler can detect elephant flows [32] and find suitable paths.

4.3 Results and Analysis

We show two experiment results. Section 4.3.1 shows hit rate results based on different queue sizes and cache sizes. Section 4.3.2 gives throughputs of nonblocking per-port cache and blocking per-port cache.

4.3.1 Cache hit rate



We use 64-byte packets to perform stress test in both core switch and aggregation switch.

Figure 4.2 shows the aggregation switch (agg-switch) hit rate. The hit rate increases as cache size grows. For a fixed cache size, the hit rate also increases as the queue size increases.

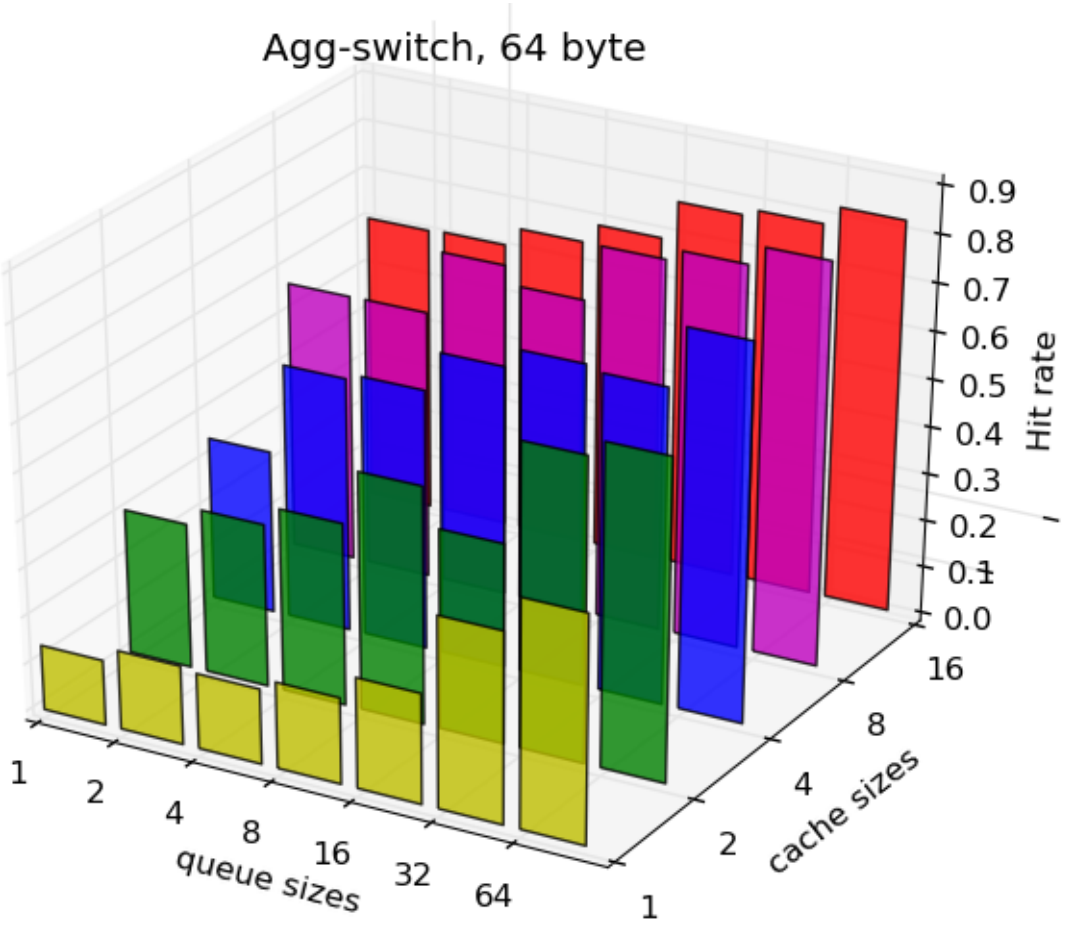


FIGURE 4.2: Agg-switch's cache hit rates

Figure 4.3 shows the cache hit rate for packets from the queue. Longer queue leads to more packet hit in the cache because the queue reduces the number of SSE requests.

Figure 4.4 and 4.5 show the hit rate data of the core switches. The RipL-POX controller distributes flows into different paths so that the core switches can take advantage of flow locality.

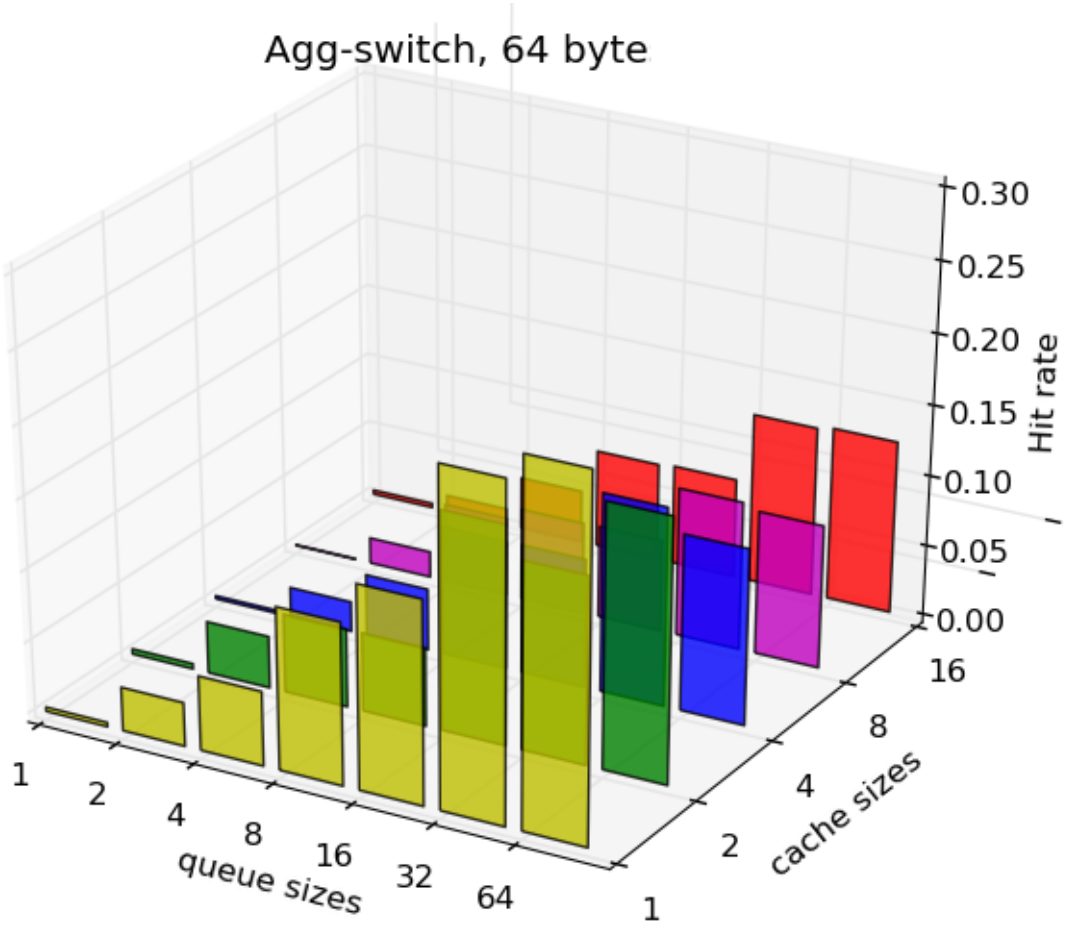


FIGURE 4.3: Agg-switch's hit rates from the queue entry

Overall, core switch hit rates are higher than that of aggregation switches.

Figure 4.6 shows the relationship between packet sizes and hit rates in a nonblocking cache. The hit rate for 512-byte packets is the highest. For 64-byte packets, the packet size is too small to be optimized by the controller, which results in

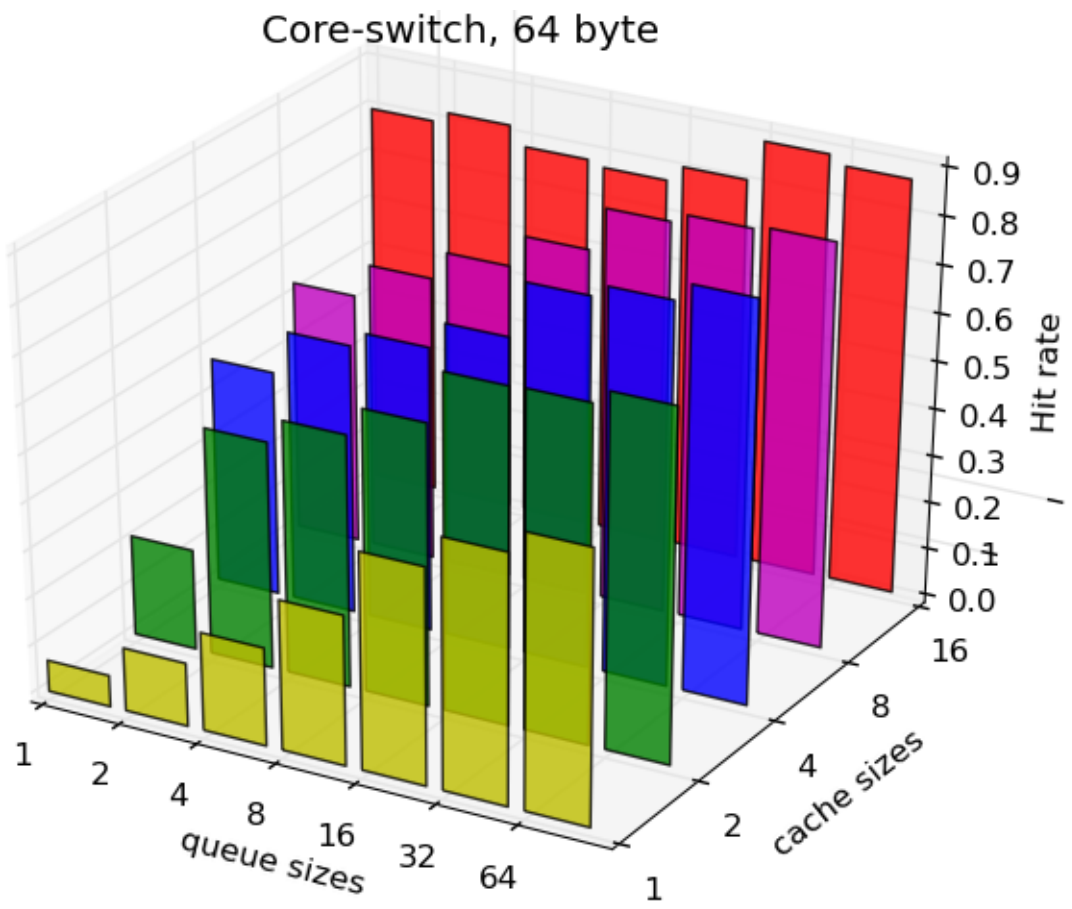


FIGURE 4.4: Core-switch's cache hit rates

low hit rates. 1500-byte packets are too long for the controller to receive more packets. Therefore, the cache is unable to show the advantage of elephant flow [32] and flow locality.

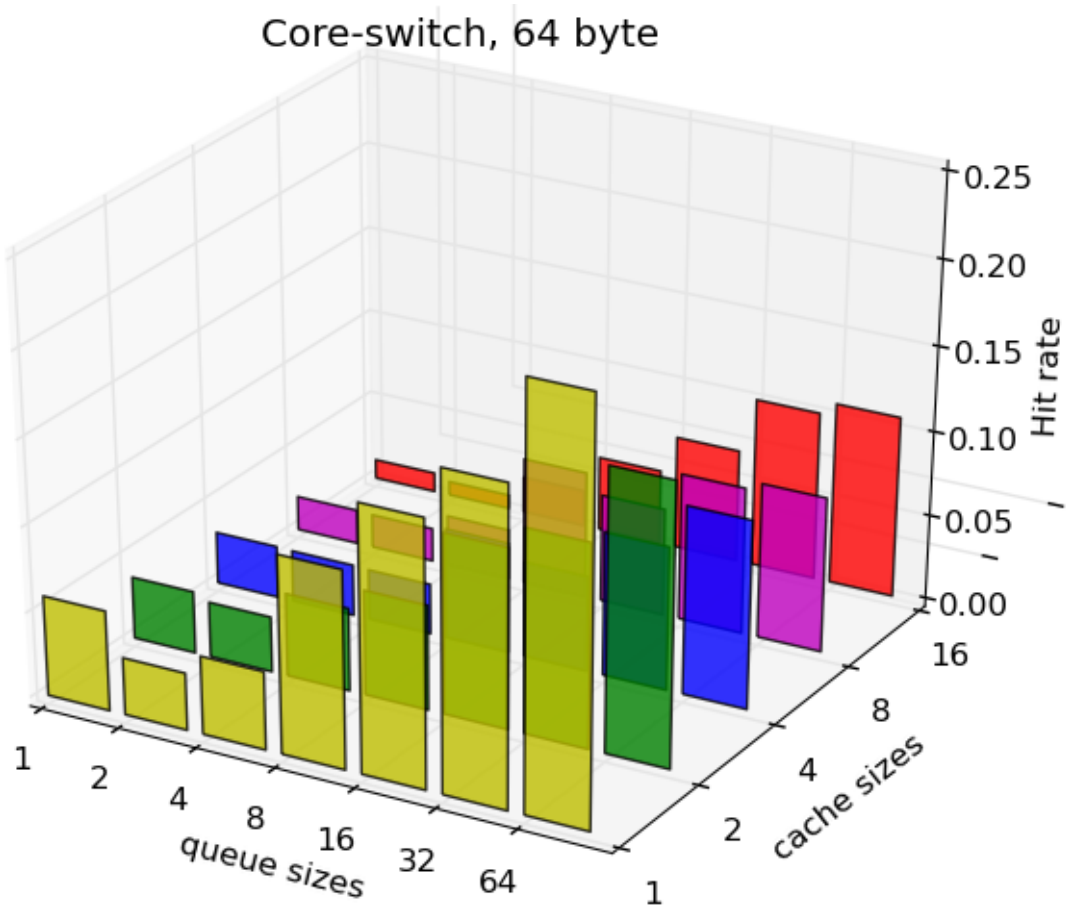


FIGURE 4.5: Core-switch's hit rates from the queue entry

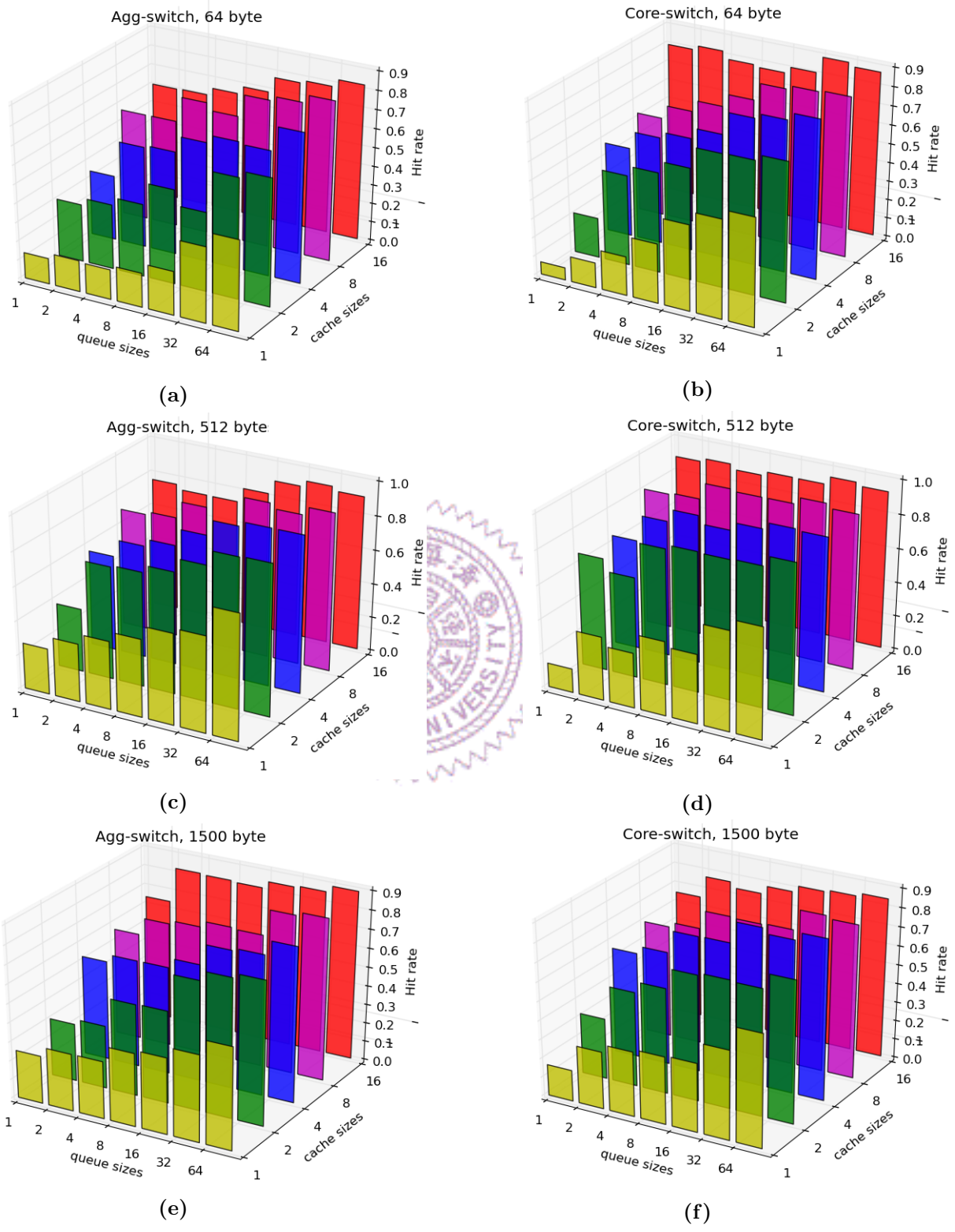


FIGURE 4.6: Showing all configuration nonblocking cache switch's hit rate

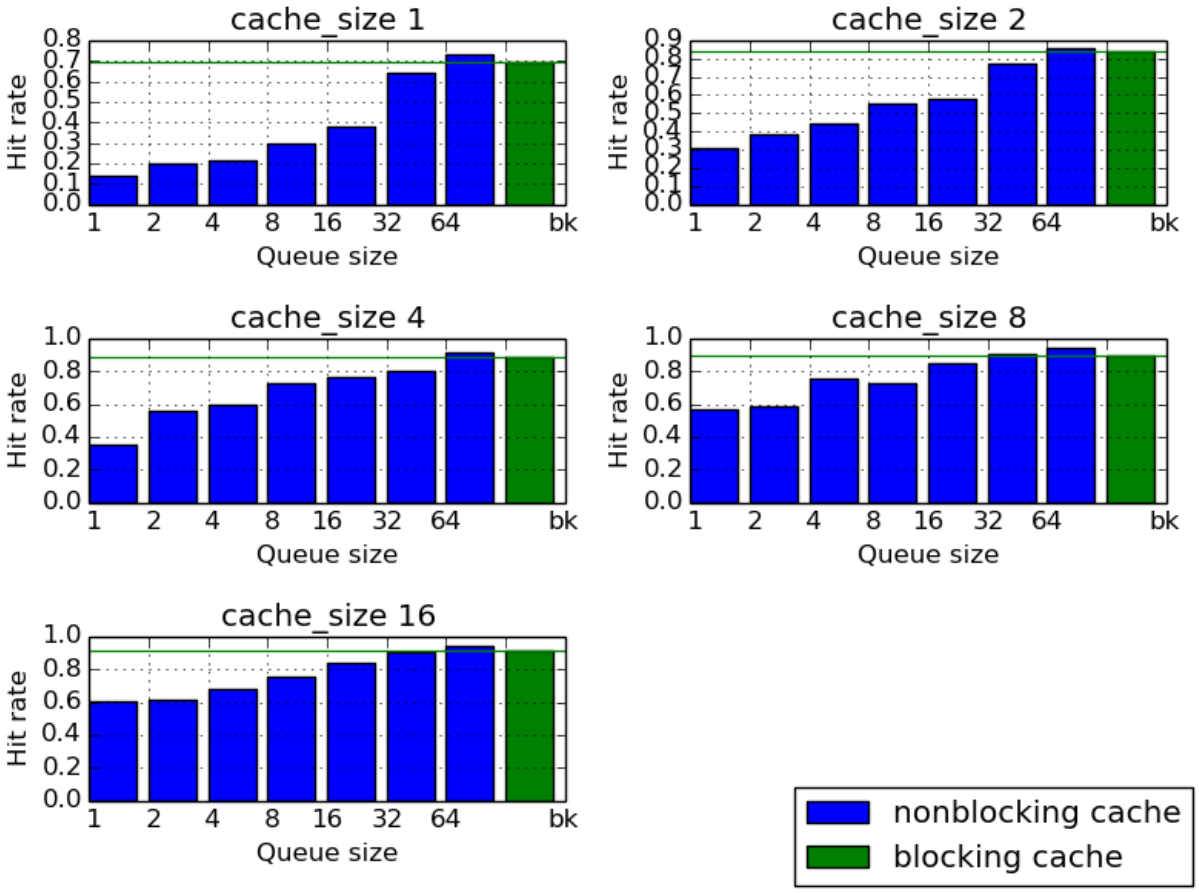


FIGURE 4.7: Comparing blocking cache and nonblocking cache hit rate in agg-switch

Finally, we compare our design with that uses blocking cache. Figure 4.7 and 4.8 show the hit rates. The nonblocking cache hit rate is higher than that of blocking cache when the queue size is about 64. Besides, the results show that the queue indeed effectively improve nonblocking cache hit rate.

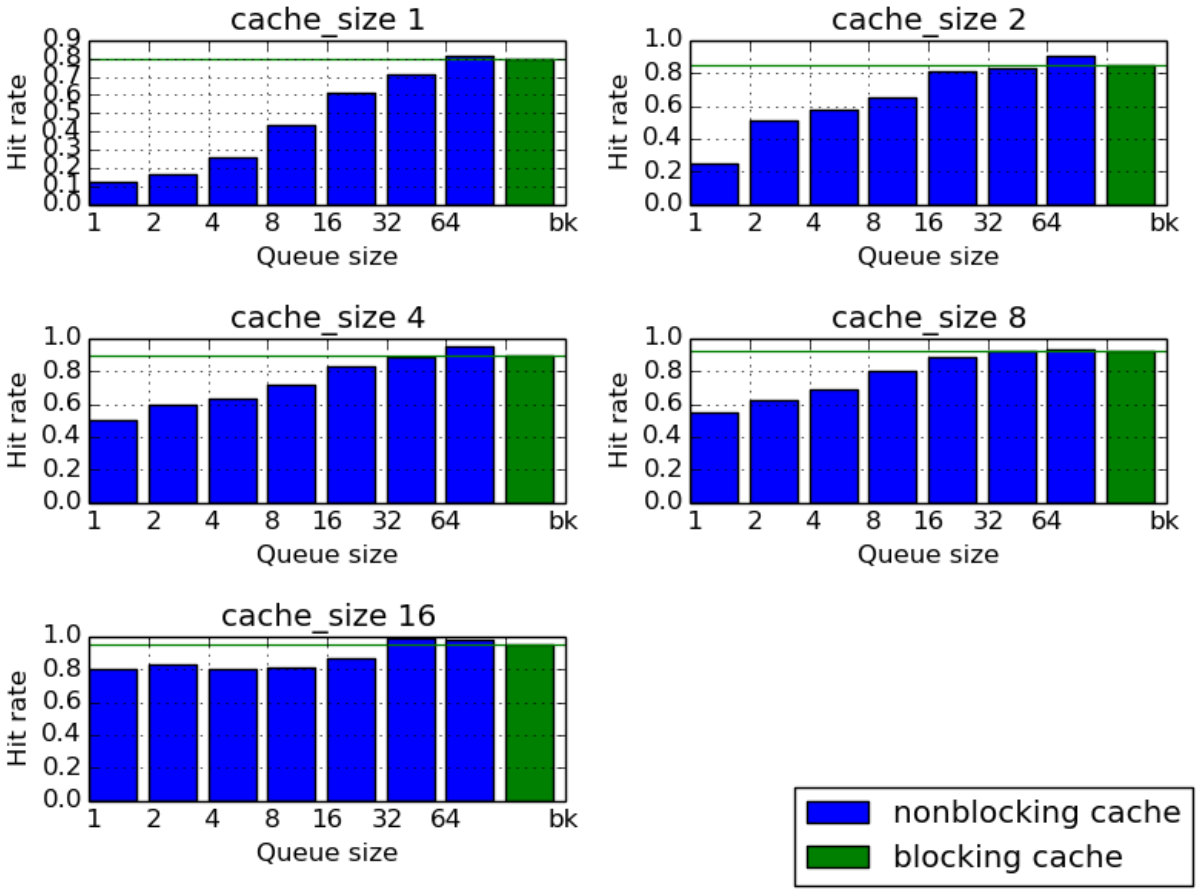


FIGURE 4.8: Comparing blocking cache and nonblocking cache hit rate in core-switch

4.3.2 Switch throughput

In our experiment, the switch has 4 ports and each port will receive 1G bps. We try to saturate the switch. For 64-byte packets, the switch will receive ((1Gbit/8bits)/64bytes)*4 = (2M)*4 = 8M packets in a second. For 512-byte and 1500-byte packets, they are 0.26M*4 = 1.04M and 0.089M*4 =

0.356M, respectively.

Figure 4.9 shows the switch throughput defined as packets forwarding per second. The throughput increases as the queue size or cache size increase.

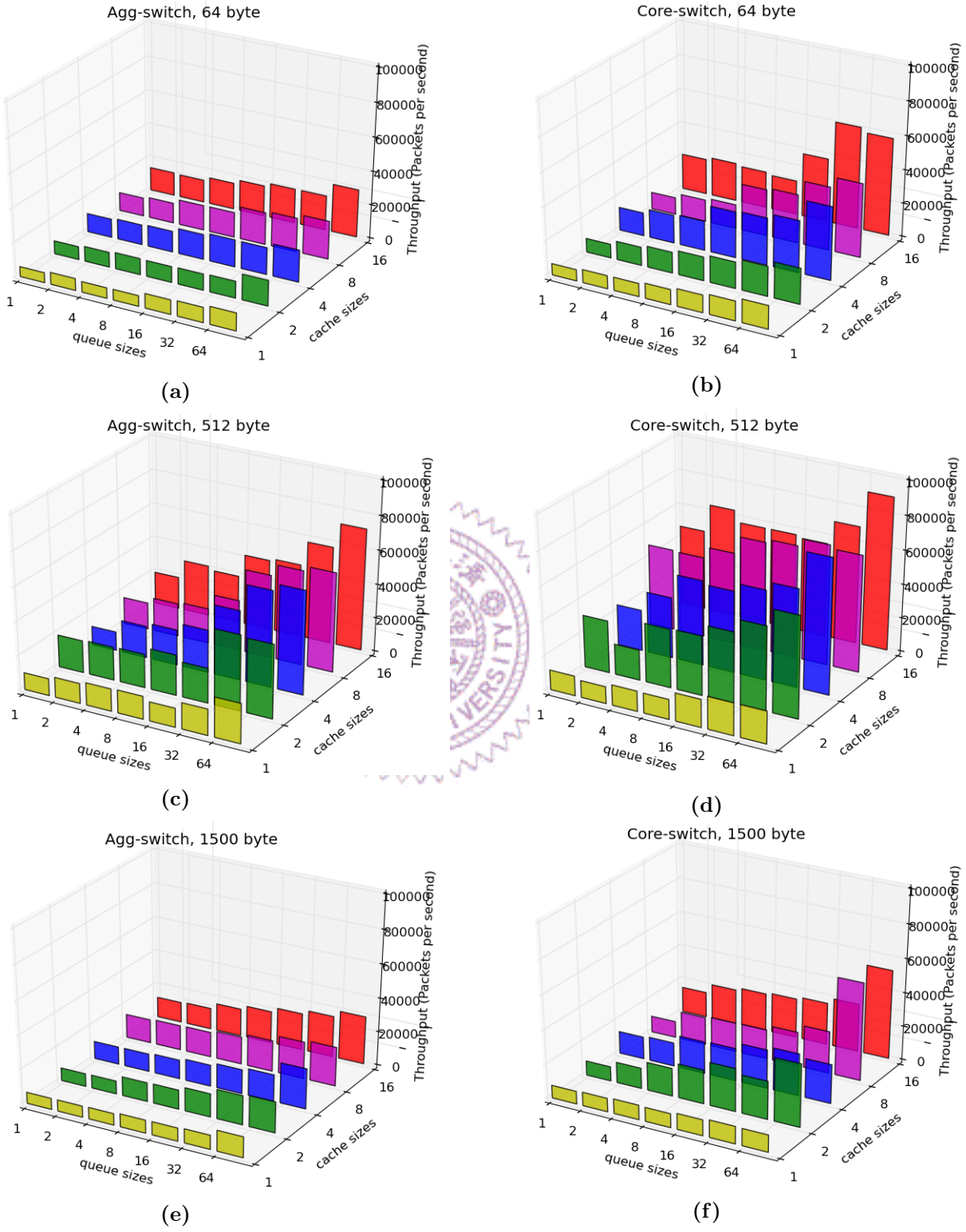


FIGURE 4.9: Showing all configuration nonblocking cache switch's throughput

Overall, the throughput for 64-byte packets is the lowest because the switch has the lowest hit rate. The throughput for 1500-byte packets is lower than that for 512-byte packets because 1500-byte packets need more time to transfer packet payload.

We study the relationship between switch throughputs and cache hit rates. Figure 4.10 and 4.11 show agg-switch and core-switch results when handling 64-byte packets. The throughputs increase when hit rates increase. Since a longer queue can store more subsequent packets, the number of requests to the software is effectively reduced. In addition, higher cache hit rate leads to more packets processed by the hardware.

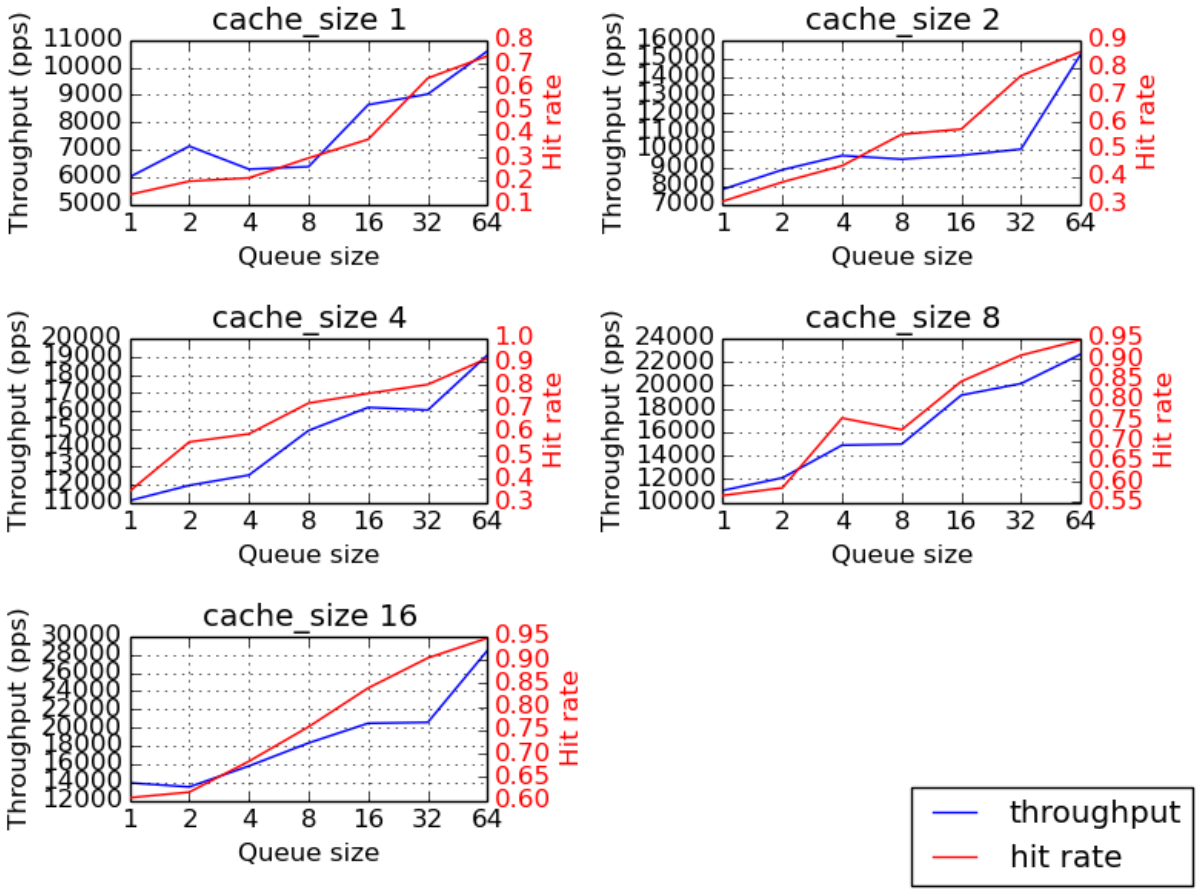


FIGURE 4.10: Agg-switch's hit rate and throughput (pps)

Figure 4.12 and Figure 4.13 compare the throughputs of switches using nonblocking and blocking cache. The nonblocking cache design achieves higher throughput when its cache queue size is 8 or more.

Finally, we use Design Compiler [33] to synthesize our design targeted toward TSMC .13 μm cell library. Figure 4.14

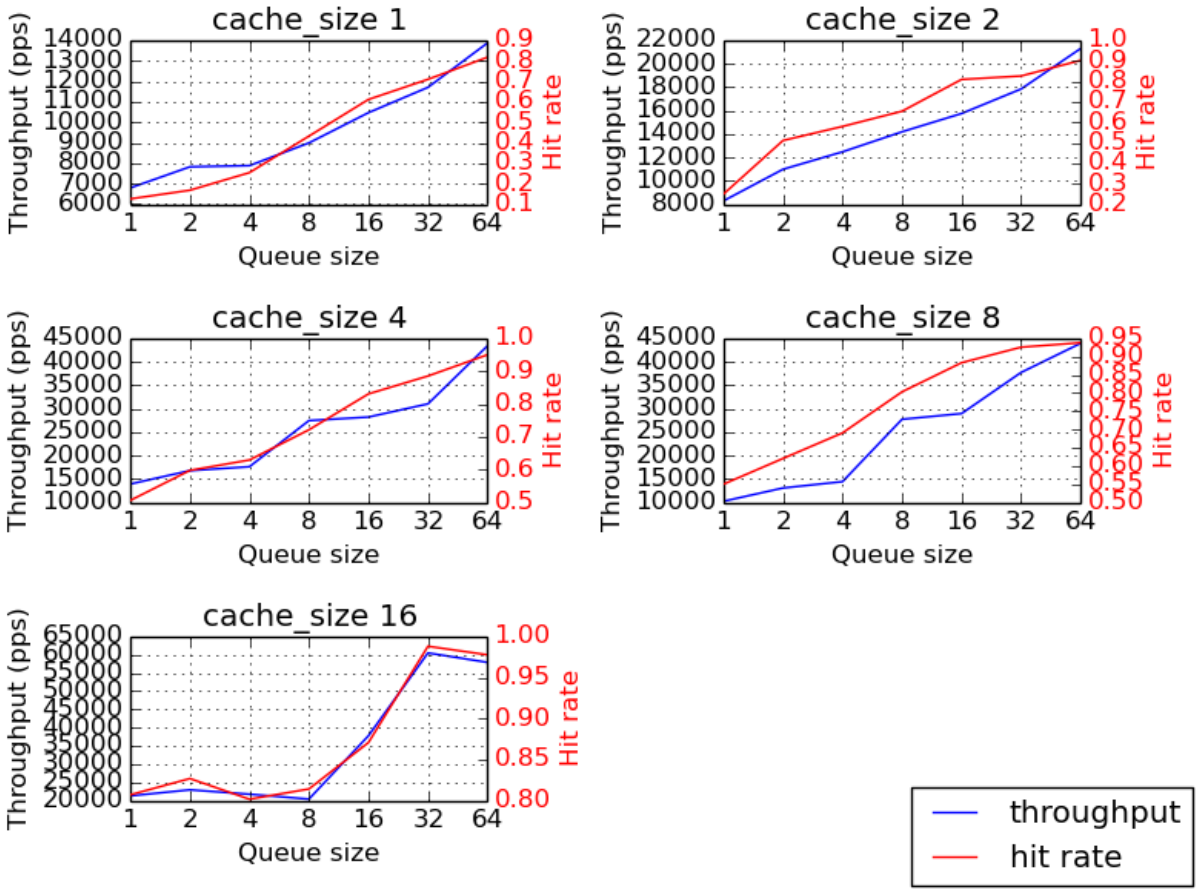


FIGURE 4.11: Core-switch's hit rate and throughput (pps)

and 4.15 show the throughput with its corresponding cache (c #) and queue size (q #). We show throughput as function of hardware cost. Blocking cache design saturate very quickly. Nonblocking cache design, on the other hand, can achieve very high throughput using certain combinations of cache on queue design (queue size is over 8 and cache size is over 16).

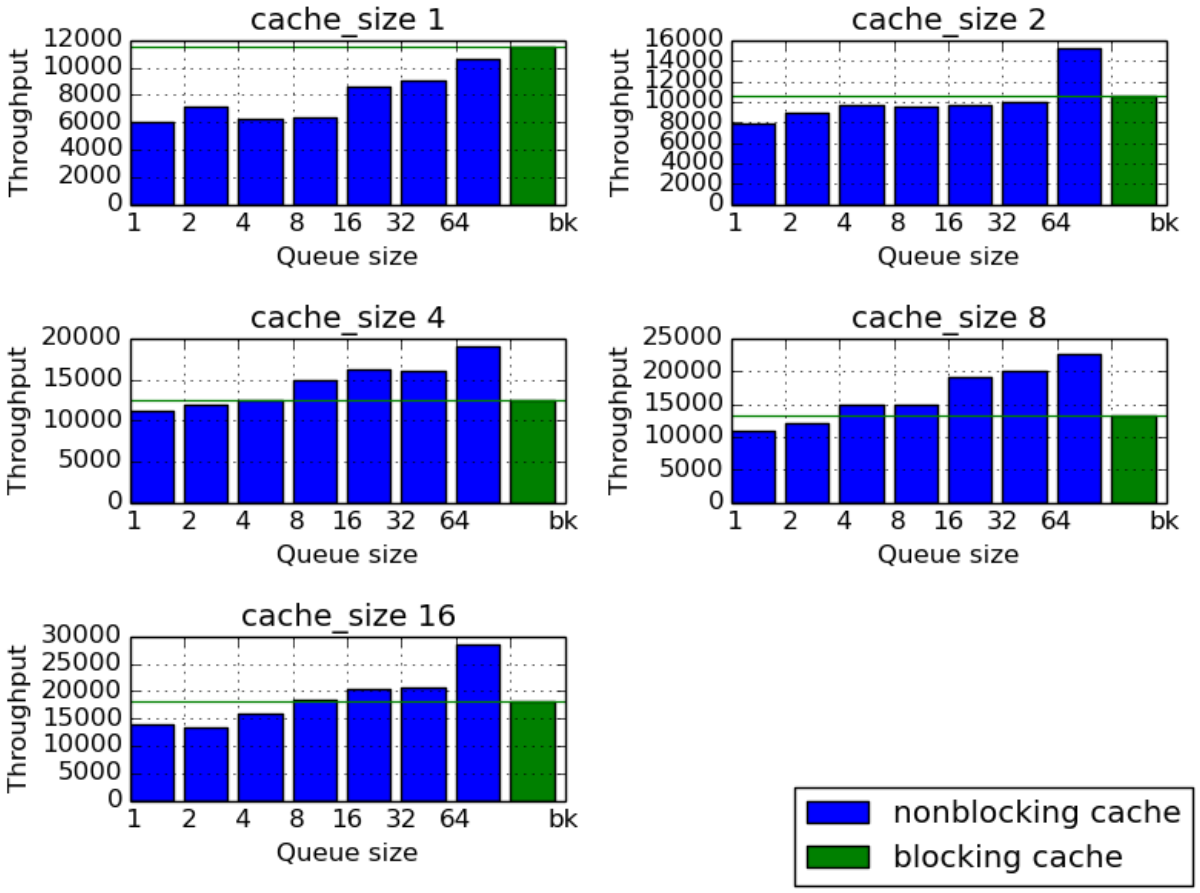


FIGURE 4.12: Agg-switch's throughput (pps) in same cache

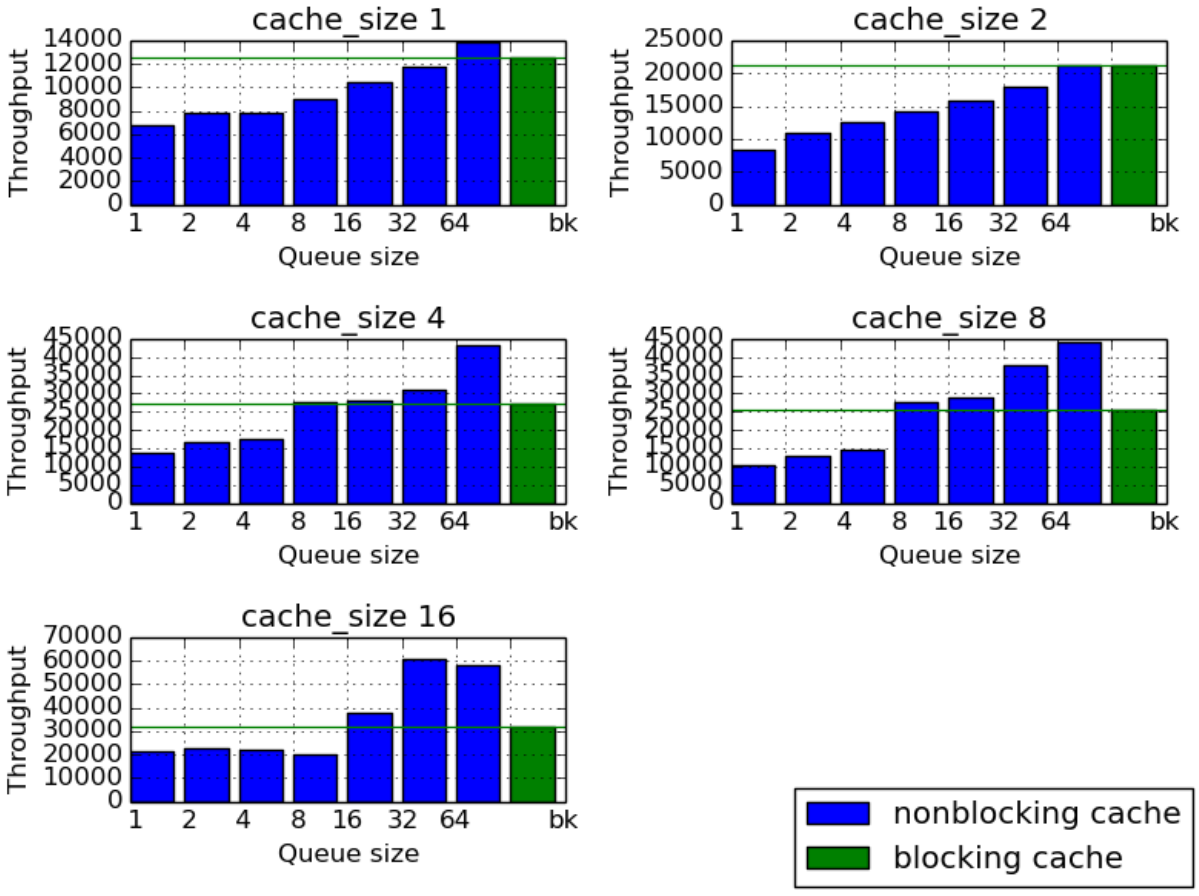


FIGURE 4.13: Core-switch's throughput (pps) in same cache

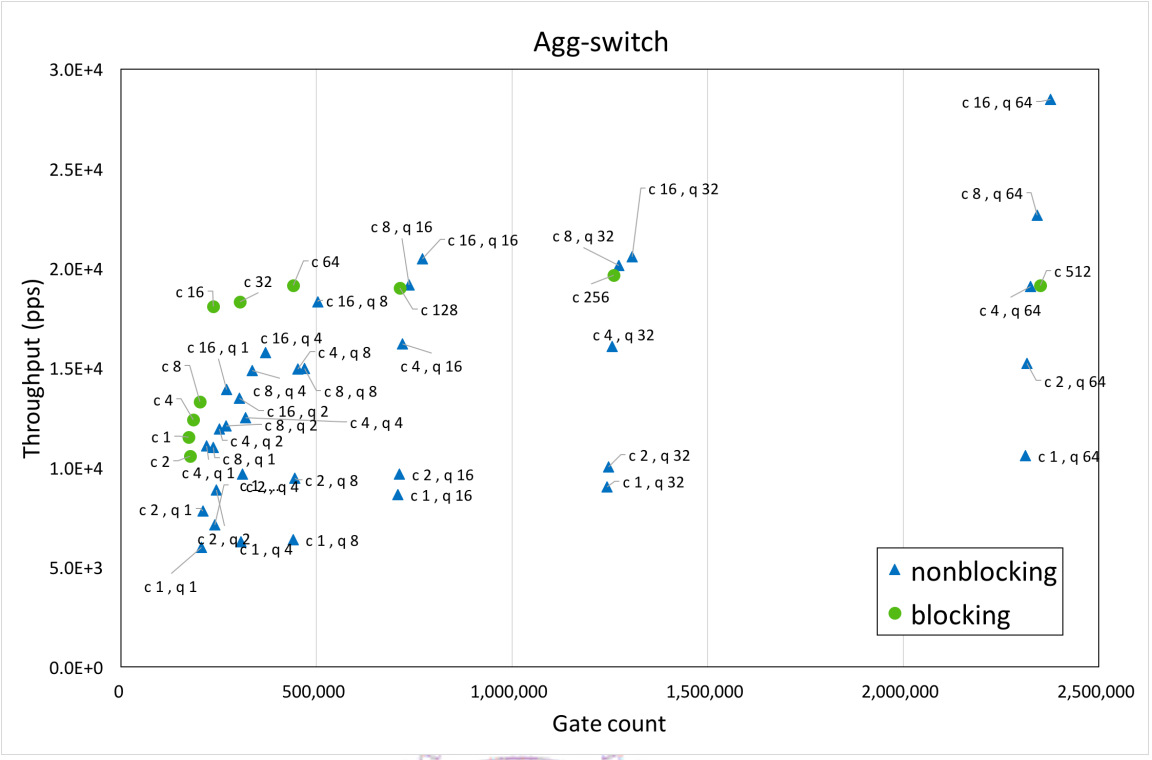


FIGURE 4.14: The different cache configuration for agg-switch with throughput and gate counts

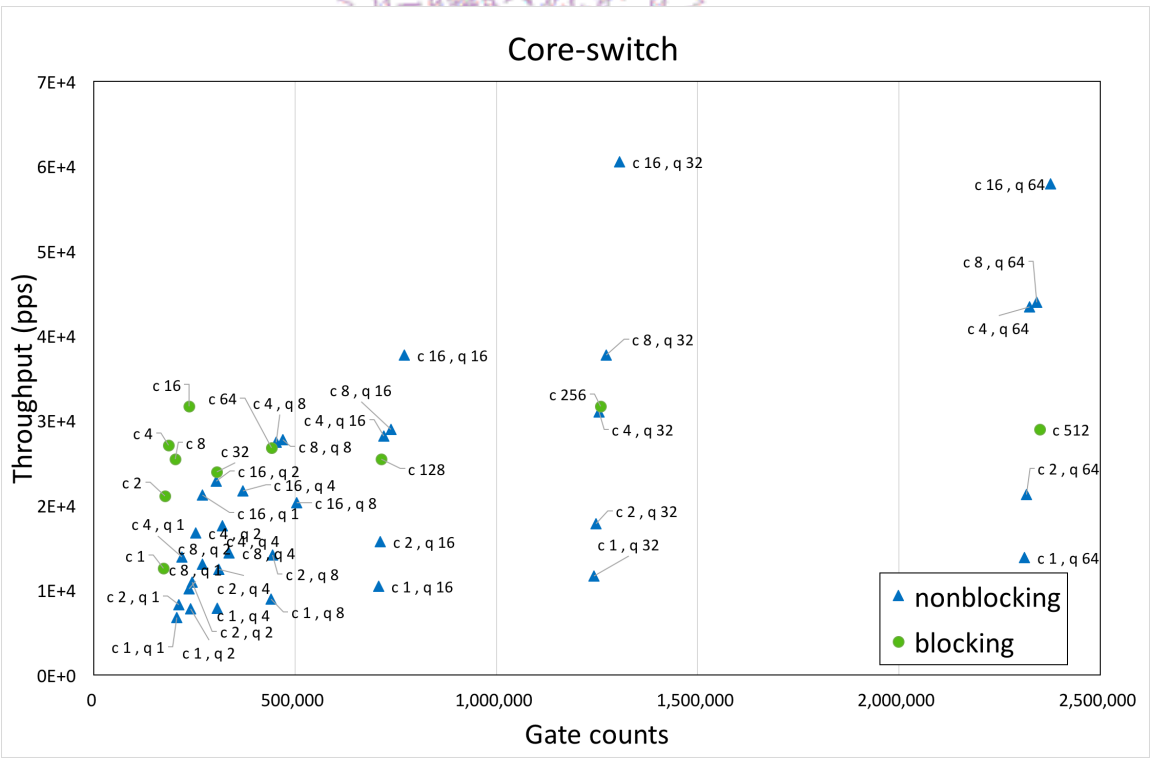


FIGURE 4.15: The different cache configuration for core-switch with throughput and gate counts

Chapter 5

Conclusion And Future Work

We have proposed a queue-based nonblocking per-port cache (Q-NPPC) design to improve the throughput of a white box switch. It tackles the problem that nonblocking cache encounters low cache hit rate. The Q-NPPC can achieve higher throughput than Reference [14]. Experiment results show that the Q-NPPC has higher hit rate compared to that of blocking cache.

In the future, we can apply more OpenFlow protocol functions to extend the proposed Q-NPPC. The Q-NPPC currently does not have timeout mechanisms or returning counter to the controller. We can implement the Q-NPPC in ASIC or a large network emulator so that we can conduct larger scale experiment.

References

[1] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[2] "L. foundation, opendaylight: an open source community and meritocracy for software-defined networking, a linux foundation collaborative project," 2016, available at https://www.opendaylight.org,.

[3] "Floodlight." [Online]. Available: http://www.projectfloodlight.org/

[4] "Pox." [Online]. Available: http://www.noxrepo.org/pox/about-pox/

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

[6] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. H. Dave, A. W. Moore, and P. G. Neumann, "Blueswitch: Enabling provably consistent configuration of network switches," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*, ser. SIGCOMM '15. ACM, 2015, pp. 363–364.

[7] P. Bosshart, G. Gibb, K. Hun-Seok, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, ser. SIGCOMM'13. ACM, 2013, pp. 99–110.

[8] "Tcam." [Online]. Available: https://en.wikipedia.org/wiki/Content-addressable_memory#Ternary_CAMs

[9] "Ram." [Online]. Available: https://en.wikipedia.org/wiki/Random-access_memory

[10] "L. foundation, openswitch, community-based, open source, full-featured network operating system," 2016. [Online]. Available: http://www.openswitch.net/

[11] "O. community. source code of open vswitch. openvirtualswitch." 2014, available at https://www.openvswitch.org.

[12] "What is white box switching and white box switches?" 2016, available at https://www.sdxcentral.com/cloud/converged-datacenter/whitebox/definitions/what-is-white-box-networking/.

[13] "Netfpga 1g cml reference nic projects," 2015. [Online]. Available: https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-1G-CML-Projects

[14] C. Y. Lin, "Accelerating openflow switches with per-port cach," 2015. [Online]. Available: http://140.113.39.130/cgi-bin/gs32/hugsweb.cgi?o=dnthucdr&s=%22GH02102062501%22.id.&

[15] "The apache license." [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0

[16] T. Chiueh and P. Pradhan, "Cache memory design for network processors," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, ser. HPCA-6, 2000, pp. 409–418.

[17] A. Siddiqui, "Whitebox switches deployment experien," 2016, available at https://conference.apnic.net/data/41/white-box-switching_1456262714.pdf.

[18] "Introduction to pci protocol," available at http://electrofriends.com/articles/computer-science/protocol/introduction-to-pci-protocol/.

[19] T. Luo, H.-P. Tan, P. C. Quan, Y. W. Law, and J. Jin, "Enhancing responsiveness and scalability for openflow networks via control-message quenching," in *Proceedings of Int. Conf. CT Convergence (ICTC)*. IEEE, 2012, pp. 348–353.

[20] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th annual symposium on Computer Architecture*. ACM, May 12-14, 1981, pp. p.81–87.

[21] T. Benson, A. Anand, A. Akell, and M. Zhang, "Understanding data center traffic characteristics," in *ACM SIGCOMM Computer Communication*, 2010.

[22] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements and analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ser. ANCS '09. ACM, 2009.

[23] "Erfc," available at http://mathworld.wolfram.com/Erfc.html.

[24] "Cumulative distribution function," available at https://en.wikipedia.org/wiki/Cumulative_distribution_function.

[25] "Modelsim," available at http://www.mentor.com/products/fv/modelsim/.

[26] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[27] "Ubuntu," available at http://www.ubuntu.com/download/desktop.

[28] "Ripcord-lite for pox: A simple network controller for openflow-based data centers." available at https://github.com/brandonheller/riplpox.

[29] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855730

[30] "Iperf," available at https://iperf.fr/.

[31] "Numpy v1.11," available at http://docs.scipy.org/doc/numpy/index.html.

[32] Broadcom, "Engineered elephant ows for boosting application performance in large-scale clos networks," 2014. [Online]. Available: https://www.broadcom.com/collateral/wp/OF-DPA-WP102-R.pdf

[33] "Design compiler," available at http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.