NB-Cache: Non-Blocking In-Network Caching for High-Speed Content Routers

Tian Pan
BUPT
pan@bupt.edu.cn

Xingchen Lin
BUPT
linxingchen@bupt.edu.cn

Jiao Zhang
BUPT
jiaozhang@bupt.edu.cn

Hao Li
Xi'an Jiaotong University
hao.li@xjtu.edu.cn

Jianhui Lv
Huawei Technologies
lvjianhui@huawei.com

Tao Huang
BUPT
htao@bupt.edu.cn

Bin Liu
Tsinghua University
liub@tsinghua.edu.cn

Beichuan Zhang
University of Arizona
bzhang@arizona.edu

ABSTRACT

Information-Centric Networking (ICN) provides scalable and efficient content distribution at the Internet scale due to its in-network caching and native multicast capabilities. To support these features, a content router needs high performance at its data plane, which consists of three forwarding steps: checking the Content Store (CS), then the Pending Interest Table (PIT), and finally the Forwarding Information Base (FIB). While prior works focus on performance optimization of a single step, we build an analytical model of content router's entire data plane and identify that CS is the actual bottleneck in the pipeline. Compared with PIT and FIB, CS is more challenging because it has more data to read/write, may have more entries in its table to store and lookup, and needs to organize content objects to sustain frequent cache replacement. Then, we propose a novel mechanism called "NB-Cache" to address CS's performance issue from a network-wide point of view rather than a single router's. In NB-Cache, when packets arrive at a router whose CS is fully loaded, instead of being blocked and waiting for the CS, these packets are forwarded to the next-hop router, whose CS may not be fully loaded. This approach essentially utilizes Content Stores of all the routers along the forwarding path in parallel rather than checking each CS sequentially. Our experiments show significant improvement of data plane performance: 70% reduction in round-trip time (RTT) and 130% increase in throughput.

CCS CONCEPTS

• **Networks → Routers**; **In-network processing**; *Network performance modeling*.

KEYWORDS

ICN, Content Router, Bottleneck Bypassing, Non-Blocking I/O

ACM Reference Format:
Tian Pan, Xingchen Lin, Jiao Zhang, Hao Li, Jianhui Lv, Tao Huang, Bin Liu, and Beichuan Zhang. 2019. NB-Cache: Non-Blocking In-Network Caching for High-Speed Content Routers. In *IEEE/ACM International Symposium on Quality of Service (IWQoS '19), June 24–25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3326285.3329041

1 INTRODUCTION

The Internet has witnessed explosive growth of content distribution, from web pages, files, to videos and gaming. This drives the underlying network architecture to evolve from the traditional host-to-host communication towards large-scale content dissemination and retrieval. This evolution includes Content Distribution Networks (CDN) as well as more recently proposed Named Data Networking (NDN) [20] or Information-Centric Networking (ICN) in general. The common theme in these recent architectures is that they make content name explicit in each packet, and the *content routers* forward packets based on names instead of addresses, and store returned copies of content objects to facilitate in-network caching and native multicast. In this way, content objects can be cached at and retrieved from any place, thus reducing network congestion and content retrieval delay. Although this architectural innovation gains advantages of optimized data delivery, it also adds extra states into the network intermediary nodes and makes ICN packet forwarding sophisticated compared with the stateless IP [11]. As a result, promoting data plane performance becomes a prerequisite to large-scale ICN deployment.

A content router owns a three-stage processing pipeline: the Content Store (CS), the Pending Interest Table (PIT), and the Forwarding Information Base (FIB). The CS caches content objects, the PIT records Interests (*i.e.*, requests for data) that have already been forwarded, and the FIB is a routing table indexed by content name prefixes. An Interest packet goes through these three steps in the order of CS, PIT and FIB: if it finds the requested content in CS, it will return the content; if it finds the same Interest in the PIT, it will be recorded but not forwarded; otherwise, it will be forwarded to a next hop based on FIB lookup. Generally, a pipeline runs only as fast as its *slowest* stage and the overall data plane performance is determined by the bottleneck of these three steps. Prior works, however, focus only on improving an individual step, mostly on

FIB or PIT [16–19]. It is unclear where the exact bottleneck is in the content router pipeline and how to address the bottleneck to improve the overall data plane performance.

In this work, we develop a model to analyze the data plane performance of content routers. With this model, we are able to quantitatively identify that CS is the bottleneck of the router. While it makes sense intuitively since all incoming traffic will first check CS at the first pipeline stage, the parameterized model further allows us to quantify the traffic load distribution at each pipeline stage, helping the design and evaluation of any future solution on content router architectural innovation.

Next, we investigate solutions to address the CS bottleneck problem. The straightforward approach would be to improve CS performance by techniques such as designing elegant data structures, optimizing implementations, leveraging latest hardware, and so on. Though we can certainly include these techniques in the solution, they are not likely to be able to mitigate CS as the bottleneck. Compared with PIT and FIB, CS has some intrinsic properties that make it slower: larger size of data chunks to read/write, more entries in the table to store and lookup, and caching policies add constraints to data structure design. Besides, a generic algorithm or hardware-based solution often can be applied to CS as well as PIT/FIB, making all faster but CS still remains the system bottleneck.

We propose a novel solution called "NB-Cache", which addresses the performance issue from a network-wide point of view instead of individual router's. In current ICN network, when packets arrive at a router whose CS is fully loaded, these packets have to be queued in the front of CS to wait for the next processing cycles. Thus many packets are blocked at the first overloaded router, but all the routers after that one may not have CS queue. *Instead of queuing these packets at the first router, NB-Cache forwards them directly to the next-hop router, bypassing the overloaded local CS.* At the next router, the packets will probably have CS cycles to process them. If later the next router also becomes overloaded, packets will again bypass it and go to the next router further upstream. Eventually, given enough workload, all the routers along the forwarding path will have their CS working fully loaded to process the traffic. From network's point of view, NB-Cache turns a sequential access of many router's CS into a parallel access of these CS at nearly the same time. Although some packets need to travel a longer distance, they may end up with retrieving data in an even shorter time because the queuing at the CS is reduced. This *network-wide load balancing* leads to more efficient use of network resources, less congestion at routers, and shorter RTT and higher throughput.

The major contributions can be summarized as follows:

- We build a queuing network-based model for ICN data plane and quantitatively identify CS as the bottleneck. The timely result will potentially *shift* research interests from previous FIB and PIT to CS, preventing "blind optimization" or "over-optimization" (§2).
- We propose NB-Cache to relieve the local CS congestion via network-wide load balancing. It includes four techniques: Bloom filter as the first-stage bypass; active queue management as the second-stage bypass; non-blocking I/O for immediate control return; router-assisted congestion control with lazy congestion notification (*i.e.*, NB-CC). NB-Cache does not change router's interfaces to the outside, thus it is *compatible* with existing content routers. Besides, when CS is not congested, NB-Cache's traffic bypass mechanism will *not* be triggered to disturb the nearest content fetch by default (§3, §4).
- We implement an NB-Cache-enabled router prototype and a network emulation environment with 3124 lines of C++ code. All source code is *available* at out git repository [2]. In our experiment, NB-Cache can reduce RTT by 70% and improve throughput by 130% compared with the current ICN data plane architecure (§5).

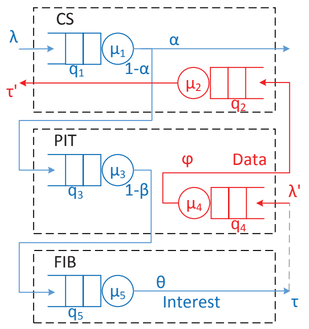


Figure 1: Modeling a content router via queuing networks.

2 WHERE IS THE BOTTLENECK?

2.1 Content Router Architecture

In ICN, there are two types of packets: the *Interest* and the *Data*. Content consumer sends the Interest packets to the network for content retrieval and the requested content will be pulled back encapsulated as the Data packets. When an Interest goes into a content router, it will first query CS for its requested content. If the piece of content exists, a Data containing that content will be directly sent back. Otherwise, the Interest will further query PIT, and if a matching PIT entry exists, it will simply add its incoming interface to that entry. In the absence of a matching PIT entry, the router will create a new entry in PIT and forward the packet by searching FIB. When a Data from upstream networks comes back, the router will find a matching PIT entry and forward the Data to all the downstream interfaces listed in the PIT entry. The Data packets always take the reverse path of the Interests, and, without considering packet losses, one Data corresponds to one Interest, which provides *flow balance*.

2.2 Bottleneck Identification via Modeling

Observing content router's packet forwarding process, we can figure out the following characteristics. The forwarding process can be divided into two parts, namely, the ingress and the egress. In each part, packets are processed through multiple pipelined components. Packets leaving a component may have options as to where to go to next. All the functional components are connected by directed line segments denoting the packet flow. Incoming packets can be blocked or even dropped due to component overloading.

2.2.1 Basic assumptions. Assuming (i) in each direction, a functional component can be abstracted as a unit consisting of a FIFO *queue* for packet buffering and a *server* for packet processing, (ii) packets arrive at each queue follow the Poisson process, (iii) the packet service time at each server follows the negative exponential distribution, (iv) each component in either upstream or downstream direction can be regarded as an M/M/1 queuing subsystem, and (v) the packet transfer rate between any two components is steady, we can leverage the *open queuing network* (*i.e.*, the Jackson network) [8] to model the ICN forwarding process. In this model, CS and PIT both have the ingress queue and the egress queue for both types of packets, while FIB only has an ingress queue (as shown in Fig. 1). Table 1 lists the notations in our model.

Table 1: Notations in the Model

Notations	Descriptions
λ	average Interest packet arrival rate
T_i	average throughput of the ith queuing subsystem
μ_i	average service rate of the ith server
α	CS hit rate by Interest packet
β	PIT hit rate by Interest packet
θ	FIB hit rate by Interest packet
τ	average Interest packet departure rate
λ'	average Data packet arrival rate
φ	the opportunistic caching rate
τ'	average Data packet departure (cached) rate
ρ_i	the fraction of time the ith server is busy

2.2.2 *Analyzing the packet queuing system.* According to the Jackson network, for each queue, the average packet arrival rate should be equal to the average packet departure rate when the system is in a steady state. Assuming the average packet arrival rate from the outside is λ, then for the ingress we have

$$\begin{cases} T_1 = \lambda \\ T_3 = (1-\alpha)T_1 \\ T_5 = (1-\beta)T_3 \\ \tau = \theta T_5 \end{cases} \tag{1}$$

Similarly, for the egress we have

$$\begin{cases} T_4 = \lambda' \\ T_2 = \tau' = \varphi T_4 \end{cases} \tag{2}$$

As mentioned in §2.1, without considering packet losses, one Data packet always corresponds to one Interest packet. Hence, in the long run, the ingress packet departure rate should roughly be equal to the egress packet arrival rate as

$$\tau \approx \lambda' \tag{3}$$

Based on Eq. (1), Eq. (2) and Eq. (3), we can calculate the expected throughput (*i.e.*, the average packet arrival rate) of each queue when the system is in a steady state as

$$\begin{cases} T_1 = \lambda \\ T_2 = \varphi\theta(1-\beta)(1-\alpha)\lambda \\ T_3 = (1-\alpha)\lambda \\ T_4 = \theta(1-\beta)(1-\alpha)\lambda \\ T_5 = (1-\beta)(1-\alpha)\lambda \end{cases} \tag{4}$$

2.2.3 *Identifying the performance bottleneck.* According to the queuing theory, each server's utilization can be derived as

$$\rho_i = \frac{T_i}{\mu_i} \le 1 \tag{5}$$

Notice that the expected throughput and the service rate are different. The expected throughput is the average packet arrival rate from the outside while the service rate shows server's inherent processing capability. Apparently, for each queue in the steady state, the packet arrival rate should be no larger than the service rate to prevent the overflow of that queue, *i.e.*, each server's utilization should be no larger than 1.

Next, by plugging Eq. (4) into Eq. (5), we have

$$\begin{cases} \lambda \le \mu_1 \\ \lambda \le \frac{\mu_2}{\varphi\theta(1-\beta)(1-\alpha)} \\ \lambda \le \frac{\mu_3}{1-\alpha} \\ \lambda \le \frac{\mu_4}{\theta(1-\beta)(1-\alpha)} \\ \lambda \le \frac{\mu_5}{(1-\beta)(1-\alpha)} \end{cases} \tag{6}$$

Here, we can figure out that content router's maximum throughput (*i.e.*, the packet arrival rate of q_1) is constrained by the *service rate* at each queue (*i.e.*, μ_i) plus the *traffic transfer probabilities* between these servers (*i.e.*, α, β, θ and φ) as

$$\max(T_1) = \max(\lambda) = \min\{\mu_1, \frac{\mu_2}{\varphi\theta(1-\beta)(1-\alpha)}, \frac{\mu_3}{1-\alpha}, \frac{\mu_4}{\theta(1-\beta)(1-\alpha)}, \frac{\mu_5}{(1-\beta)(1-\alpha)}\} \tag{7}$$

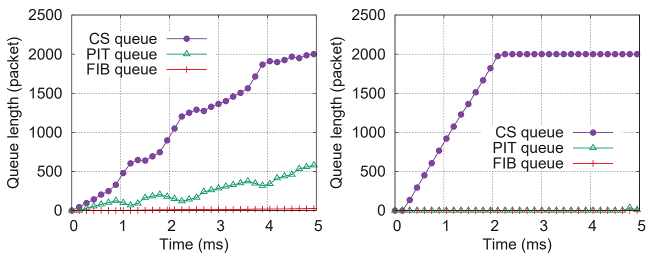
Obviously, given the traffic transfer probabilities are no larger than 1, the ingress part of CS will become the bottleneck of the entire router, except that its service rate μ_1 becomes much higher than that of the other queuing subsystems. In other words, if μ_1 is not large enough, the ingress of CS will be overloaded while the other queuing subsystems will be idle occasionally, which makes the packet pipeline inefficient. To squeeze pipeline bubbles, the service rate of each queuing subsystem (μ_i) should be budgeted to let all the equality in Eq. (6) roughly hold at the same time.

But if it is possible that the ingress processing power of CS (μ_1) will go beyond that of PIT and FIB (μ_3 and μ_5) due to technical progress someday? Generally, CS has a much larger memory footprint than that of PIT and FIB[1]. According to the memory hierarchy principle in computer architecture design, the larger-capacity memory usually exhibits lower performance. This can also be validated in §6 that in recent research articles, the reported performance of FIB and PIT have already gone beyond 100Gbps [16, 18, 19] while that of CS is only around 10Gbps [9, 10]. Anyway, by piling up a massively parallel storage/memory array or using special-purpose hardware, one can always claim a powerful CS design someday. But under a rigid budget (*e.g.*, with commercial off-the-shelf hardware), speeding up CS is not straightforward. Besides, a generic algorithm or hardware-based solution often can be applied to CS as well as PIT and FIB, making all of them faster but CS still remains the system bottleneck.

2.3 Bottleneck Identification via Prototyping

To validate the above model, we develop a multithread-based content router prototype with three pipeline stages. In the prototype, each component resides at one of the pipeline stages and is implemented by a thread. Each component also has a queue (with a fixed capacity of 2000 packets) at the front for buffering the burst

[1]Notice that the entire CS typically consists of a content index and massive content segments. Although the content index can be small enough to store in the fast memory while searched/updated by an O(1) hashing scheme, the content segments have much larger memory footprints (*e.g.*, ~GB) and relatively longer per-segment access latency (generally fetched from the external device or the off-chip DRAM). The queue between the content index and the content segments is prone to get congested.



(a) SSD as CS storage (4.7kpps). (b) HDD as CS storage (414pps).

Figure 2: CS, PIT and FIB's queue length under 15.94kpps stress test (the speed of FIB and PIT are both fixed to 4kpps and the CS hit rate is set to 50%).

requests and the inter-component queue is implemented by a ring buffer [3]. We fully implement CS using the data structures and the mechanisms described in §4. For simplicity, here, we do not implement a real PIT and FIB. Instead, we put the two threads into sleep occasionally to simulate PIT and FIB's packet processing throughput of 4kpps. The measurement is conducted on both SSD and HDD as the CS storage medium. The CS hit rate is fixed to 50%.

Fig. 2 shows the queue length of CS, PIT and FIB under 15.94kpps stress test. We can find that even the speed of CS (4.7kpps) is slightly faster than that of PIT and FIB (4kpps) in the SSD scenario, CS is still the performance bottleneck as the queue in the front of CS is accumulating much faster than that of PIT and FIB (Fig. 2(a)). In the HDD scenario (Fig. 2(b)), the situation becomes even worse that the queue in the front of CS gets overflowed quickly (CS speed drops to 414pps) while PIT and FIB are still in a nearly idle state.

The prototype also convinces us that CS is the *exact* choke point in the content router packet pipeline.

3 NON-BLOCKING IN-NETWORK CACHING

3.1 Design Space Analysis

According to Eq. (7), the CS performance is mainly impacted by two factors. The first is the inherent processing capability of CS (*i.e.*, μ_1), which can further be improved by leveraging better data structures or faster hardware. The second is the traffic transfer probabilities (*i.e.*, α, β, θ and φ), which depict the inner packet flow and are entirely decided by the router architecture. Since both of the CS and the PIT contain the exact match rules and require to handle the coexisted frequent lookups and updates, it is not that easy to develop faster data structures for CS than those for PIT. As for the latest hardware, since CS has a much larger capacity than PIT, CS's storage medium can be no faster compared with PIT's according to the memory hierarchy principle. Indeed, by modifying the traffic transfer probabilities, for example, deliberately reducing the CS hit rate, we can balance the traffic load in the three pipeline stages. However, such brute-force approach will violate ICN's initial design intent. Now that simply changing the value of the parameters in Eq. (7) does no good to the performance bottleneck elimination, we decide to *rearchitect* the content router to entirely change Eq. (7) itself. Of course, the architecture redesign should not modify router's interfaces to the outside to guarantee the *compatibility*, such that the redesigned content routers can be deployed *incrementally* with the classic content routers.

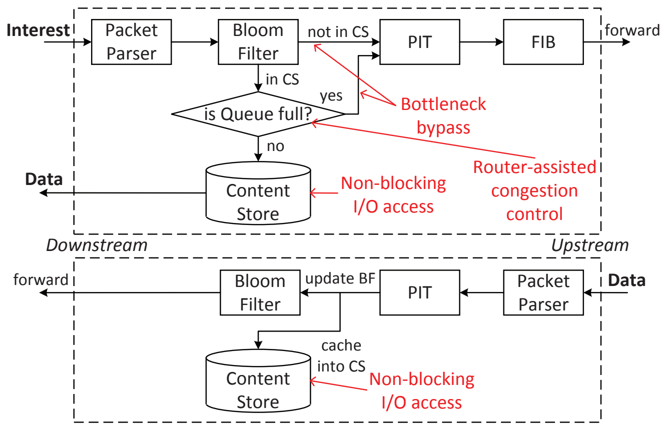


Figure 3: Non-blocking in-network caching architecture.

Before diving into the new design, we first consider how an *Internet router* handles the performance bottleneck issue. Actually, the answer is quite pessimistic. Since there is *only* one main pipeline stage (*i.e.*, FIB) in the Internet router which dominates the packet forwarding decisions. If FIB is congested, the router *has to* drop packets. Analogously, in a content router, whether or not CS has to drop packets to passively handle the congestion? In fact, the answer is *not necessarily so*. Actually, there are three pipeline stages in a content router and according to Fig. 2(a) and Fig. 2(b), PIT and FIB are usually not as busy as CS when CS is congested. Obviously, it is a radical yet reasonable idea to *migrate* the traffic from the overloaded CS to the upstream routers through the local unoccupied PIT and FIB. But if such design will break the compatibility? The answer is also no. Since CS does not decide routing, packets can still be correctly forwarded to the upstream, where the upstream router will process those packets in a normal way as if they miss the CS lookup in the downstream. Since in our proposal, packets will *bypass* the overloaded CS to access content cache in a *best-effort* manner, we call it "non-blocking in-network caching (NB-Cache)". Fig. 3 shows the design of NB-Cache. It includes four techniques: (i) using *Bloom filter* for CS bypass, (ii) enabling *active queue management* for I/O congestion avoidance, (iii) conducting *non-blocking I/O* for immediate control return, (iv) adding a *router-assisted congestion control* mechanism called "NB-CC", featuring lazy congestion notification and greedy bandwidth utilization.

3.2 Bloom Filter for Content Store Bypass (first-stage bypass)

A Bloom filter [4] is a space-efficient probabilistic data structure used to test whether an element is a member of a set. Its data structure is very compact and can be stored in the fast on-chip memory as a summary of the large-capacity item set in the slow off-chip memory. If a piece of content requested by an Interest packet does not exist in CS, by searching the Bloom filter for set testing with O(1) complexity, we can bypass the unnecessary slow CS access. The side effect of false positives is trivial since the narrowly escaped requests will be further examined in the content store finally. Since we also need to delete an element from the Bloom filter when the corresponding piece of content is removed from CS, to be more specific, we use *counting Bloom filter* [6], a variant of Bloom filter, to construct the updatable CS bypass prefilter. Compared with the counting Bloom filter, other in-memory index data structures, such as hash table or binary tree, have either larger memory footprints or higher search complexity to cope with line-rate processing.

3.3 AQM for I/O Congestion Avoidance (second-stage bypass)

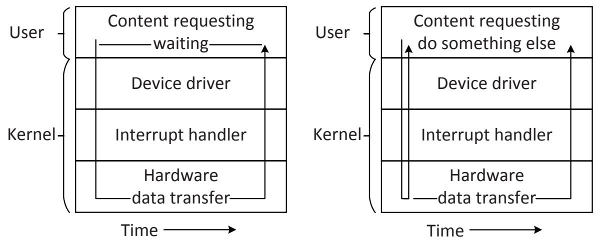
In Internet routers, active queue management (AQM), such as random early detection (RED) [7], is the intelligent drop of network packets inside a buffer when that buffer gets close to becoming full. As mentioned earlier, in a content router, CS has more options than just dropping packets when congestion occurs. Here, we propose the active queue management for I/O congestion avoidance in a content router simply by forwarding the Interest packets to the upstream routers from the local overloaded CS, although these Interests *should have* successfully hit entries in the local CS because they have already passed the examination of the counting Bloom filter as the first-stage CS bypass prefilter. These Interests will predictably fall into the CS of the light-loaded routers along the routing path in a *best-effort manner*. In essence, the idea is to load balance the CS lookups over the network to lessen the load on the local congested CS in order to minimize the overall end-to-end packet latency. In the worst case, if the CS of all the downstream routers are heavily loaded, content *sources* are prone to get flooded under high Interest rate. In this case, we have to resort to a flow/congestion control mechanism for content consumer rate adaptation (detailed in §3.5).

3.4 Non-Blocking I/O for Fast Control Return

Since the Internet content to be cached is typically of huge size and we have to use large-capacity, (relatively) slow storage medium to hold a usable CS, the I/O efficiency becomes vital to CS design. Most I/O requests are considered as *blocking* requests, meaning that control does not return to the application until the I/O is complete, and the I/O access latency can be quite long. In an interactive computing environment, the I/O wait is not really a problem. However, in cases such as high-speed network traffic processing, the blocking approach is of low efficiency which increases the average packet queuing latency. Here, we propose the *non-blocking* I/O for immediate control return dedicated to the content router. In our design, when the main thread issues an I/O command, the command can be further handled by a newly spawned thread and the main thread can return immediately (to handle the next request). By appropriately adjusting the number of spawned worker threads, the excessive multithreading can maximally drain I/O bandwidth thus improve the overall CS throughput. Although in some proposal, CS is implemented within DRAM, however, compared with the high-speed on-chip ASIC logic, the off-chip DRAM can still be considered as a slow I/O device and our design can still well apply.

3.5 Congestion Control for NB-Cache (NB-CC)

Although NB-Cache attempts to balance the traffic load across the network, content *sources* are prone to get flooded under high Interest packet rate if all the downstream routers are heavily loaded and forward the traffic upstream. Generally, the flow/congestion control is always considered as the duty of the transport layer, and here we propose NB-CC (congestion control for non-blocking content caching) as NB-Cache's transport-layer companion. Instead of trying to infer congestion at the consumer by monitoring packet losses, NB-CC detects congestion by monitoring the incoming CS queue of each router. If the CS queue occupation reaches some threshold *and there is no further traffic bypass possibility*, the router will assist explicit congestion notification by marking ECN-like bits [12] on the locally echoed back *Data* packets, which will be sent to the consumer for AIMD-like rate adjustment (Fig. 5). The traffic bypass possibility (*i.e.*, availability) of router's outgoing interface can be labeled by the upstream router via periodic hop-by-hop announcement. Notice that NB-CC differs significantly from the previous ECN-based approaches [13]. While the previous approaches detect congestion based on the local queue occupation only, NB-CC requires further knowledge about the traffic bypass possibility. If the queue occupation reaches the threshold while there is still a bypass possibility, NB-CC just *delays* congestion notification and forwards the traffic through the available outgoing interface(s). Due to the *lazy* congestion notification, NB-CC is expected to achieve higher throughput at the cost of potentially longer packet latency.



(a) Blocking I/O. (b) Non-blocking I/O.

Figure 4: Blocking I/O function calls do not return until the I/O is complete; non-blocking I/O calls return immediately, and the process can do something else and will be notified later when the I/O is complete.

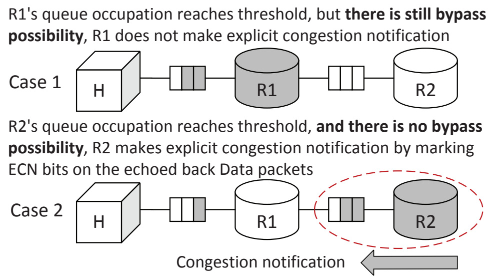

R1's queue occupation reaches threshold, but **there is still bypass possibility**, R1 does not make explicit congestion notification
Case 1
H
R1
R2
R2's queue occupation reaches threshold, **and there is no bypass possibility**, R2 makes explicit congestion notification by marking ECN bits on the echoed back Data packets
Case 2
Congestion notification

Figure 5: Lazy (delayed) congestion notification in NB-CC.

4 SYSTEM IMPLEMENTATION

4.1 System Overview

Fig. 6 illustrates a multithread-based implementation of the content router with the non-blocking in-network caching capability. The prototype can further break up into several components including a packet parser, a counting Bloom filter, a ring buffer, PIT/FIB,

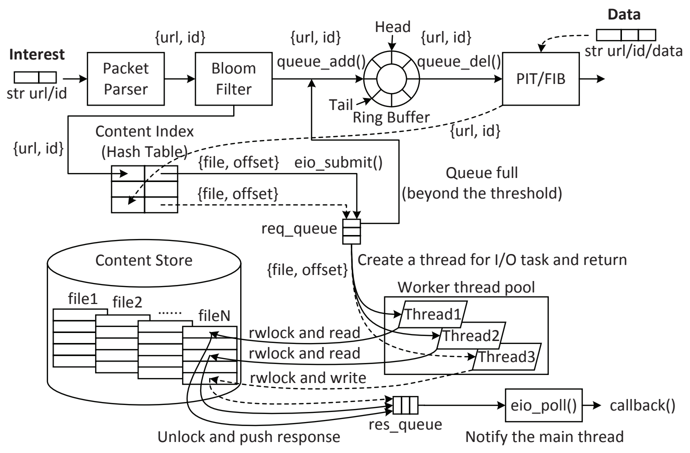


Figure 6: Multithread-based implementation of the content router with non-blocking in-networking caching capability.

a content index (hash table), content files, a request queue (*req_queue*), a response queue (*res_queue*), a worker thread pool and a polling module (*eio_poll*). On receiving an Interest packet, the router will first parse the packet header to obtain the requested content name. Then, the counting Bloom filter, the request queue length and the outgoing interface availability will be queried to determine whether the packet should bypass CS. If the queue length is beyond the threshold and there is a bypass possibility, the packet will bypass CS and be forwarded to the upstream routers. If the packet should be processed by the local CS (*i.e.*, the CS bypass threshold has not been reached or there is no bypass possibility), the main thread will issue an I/O command and return immediately (to fetch the next packet). The main thread will be asynchronously notified later when the I/O command is finished. If there is no bypass possibility, the locally echoed back Data packet will be marked according to the NB-CC algorithm as discussed in §3.5. For an incoming Data packet, it will be forwarded downstream and selectively written into CS. We use one *process* to implement a content router and multiple *threads* to implement router's components. Among them, one thread is for CS (the content object files) and its wrapper logic such as the packet parser, the counting Bloom filter and the content index; another thread is for PIT/FIB (we fit them into a single thread); a ring buffer is used to connect the two threads; a third thread is invoked to handle the Data packets from the upstream routers. We use *libeio* [1] under Linux to implement the non-blocking I/O. libeio manages a pool of OS threads and conducts the polling operation during I/O access.

4.2 Key Data Structures

4.2.1 Packet parser. The packet parser is fairly simple. It receives a string with a format like "[url]/[id]", parses that string and produces a C structure of url and id.

4.2.2 Bloom filter. We use open Bloom filter, an open source library to implement the counting Bloom filter. The library will select the optimal parameters according to the expected false positive rate and the number of inserted elements.

4.2.3 Content store. Content objects are organized into *files* in the disk. The content objects under the same name prefix (*i.e.*, the url) will be stored into the same file with the offsets decided by the segment IDs. In the memory, we build a content index to map the urls to the file names (using C++ Map). The urls and the segment IDs are parsed by the packet parser module mentioned earlier. CS is also responsible for ECN marking according to the NB-CC algorithm.

4.2.4 FIB and PIT. FIB and PIT are implemented into a single thread. We do not implement the real PIT/FIB. Instead, we configure static routes for the minimal forwarding capability.

4.3 Main Control Logic

Procedure 1 Steps that NB-Cache follows to invoke non-blocking I/O access and active queue management using *libeio*

1: The main thread calls $eio\_nready()$ to obtain the number of pending requests in the I/O queue req_queue; if the queue length has reached a pre-defined threshold, the main thread will forward the request req directly to PIT/FIB (for Interest) or forward req downstream (for Data) and return immediately (*i.e.*, CS bypass).
2: If the queue length is lower than the threshold, the main thread will invoke $eio\_read()$ or $eio\_write()$ for I/O access.
3: $eio\_read()$ or $eio\_write()$ calls $reqq\_push(\&req\_queue, req)$ inside $eio\_submit()$ to put a request req into I/O queue req_queue; $eio\_submit()$ also invokes $etp\_start\_thread()$ to start a thread in the worker thread pool to handle the I/O request; at this time, the main thread will return and do something else.
4: The started thread from the worker thread pool invokes $reqq\_shift(\&req\_queue)$ to get a request req from req_queue.
5: The worker thread tries to acquire the readers-writer lock via $pthread\_rwlock\_rdlock()$ or $pthread\_rwlock\_wrlock()$ according to the request types (*i.e.*, read or write).
6: If a lock is successfully acquired, the worker thread starts to perform I/O access via $read()$ or $write()$; otherwise, it will be blocked until the lock becomes available again.
7: When the I/O access is finished, the worker thread invokes $pthread\_rwlock\_unlock()$ to release the lock; then, it puts the I/O response req' into res_queue via $reqq\_push(\&res\_queue, req')$ and asynchronously notifies the main thread via $want\_poll()$ that there is a response; finally, the worker thread calls $etp\_worker\_clear(self)$ to free itself.
8: Once the main thread gets notified, in $eio\_poll()$, it will obtain the I/O response from res_queue via $reqq\_shift(\&res\_queue)$ and call user-defined callback function to handle the response.

4.3.1 Non-blocking I/O and active queue management. The steps that NB-Cache follows to invoke non-blocking I/O access and active queue management are shown in Procedure 1. At first, we obtain the number of the queuing requests in *req_queue* to decide whether or not to let the current request bypass CS. If the queue length has reached a pre-defined threshold, the main thread will forward the request directly to PIT/FIB (for Interest) or forward the request downstream (for Data) and return immediately. Otherwise, the request should be processed locally and will be pushed into *req_queue*. At the same time, a new thread will be started in the worker thread pool, responsible for handling the I/O access of

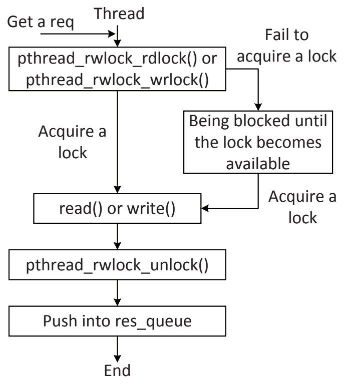


Figure 7: Resolving Interest/Data contention via locking.

the requests in *req_queue*. At this time, the main thread will return and do something else (*e.g.*, to handle the next request). Then, the newly started thread obtains a request from *req_queue* and issues the I/O access command. Here, the *read/write contention* issue is resolved via the *readers-writer lock*. When I/O access is finished, the worker thread will put the I/O result into *res_queue* and asynchronously notify the main thread to fetch the I/O response from *res_queue*. The I/O response will be further processed in a user-defined callback function.

4.3.2 Interest/Data resource contention. Since content objects are stored in files, resource contention will occur if multiple read requests (from the Interest) and write requests (from the Data) access the same file at the same time. Fig. 7 shows how to resolve such problem via the *readers-writer lock*. When we get an I/O access request from *req_queue*, we try to acquire a readers-writer lock via *pthread_rwlock_rdlock*() or *pthread_rwlock_wrlock*(). *read*() or *write*() is issued only after we obtain the lock. If we fail to obtain the lock, we have to be blocked until the lock become available. When I/O is complete, *pthread_rwlock_unlock*() is performed as an unlock operation. Currently, we lock in the granularity of *files*. If two requests fall into different files, no lock operation is needed.

4.3.3 Inter-component communication. Fig. 8 illustrates the inter-component communication implemented via the *producer-consumer lock* mechanism on a ring buffer. When adding a request to the ring buffer, we need to acquire the mutex lock first. If the ring buffer is already full, we have to wait; otherwise, we enqueue the request and modify the tail pointer (if now the ring buffer becomes full, we also need to set the full flag). Then, we unset the empty flag since there is at least one request in the ring buffer. Now we can send a notification to the ring buffer reader side to signal that the dequeue operation has been prepared. We unlock the mutex in the final. When deleting a request from the ring buffer, we have the almost mirror steps to the previous discussed insertion procedure.

4.3.4 Consumer rate adjustment. The content consumer conducts AIMD-like rate adjustment and is implemented with two threads. One thread keeps sending Interest packets unless *unacked* exceeds *cwnd*. The other thread handles *cwnd* update according to the congestion status marked on the received Data packets. Specifically, the *cwnd* should *not* be halved more than once during one RTT, guaranteed by a timer.

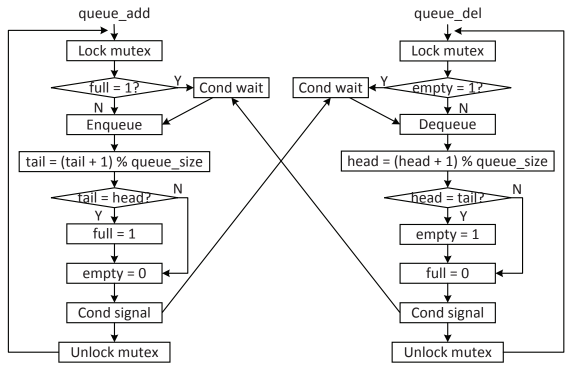


Figure 8: Inter-component communication via the producer-consumer lock mechanism with a ring buffer.

5 EVALUATION

5.1 Methodology

To evaluate the proposed NB-Cache, we build an NB-Cache-enabled content router prototype and a network emulation environment with 3124 lines of C++ code, which is *available* at our git repository [2]. The network emulation topology is shown in Fig. 9. It contains one host, four routers and the links between them. We use one *process* to emulate each router/host/link and leverage *shared memory* for inter-process communication. Inside each router (running as a process), we allocate multiple *threads* to implement router components. We also implement non-blocking I/O access using *libeio* under Linux. Since the original version of libeio cannot well handle the contention issue when multiple readers and writers access the same file, we modify its source code to address the Interest/Data contention problem. As real-world ICN traces are not quite available, we evaluate using a synthetic trace with a name format like "[url]/[id]". We select 500 url prefixes from "Alexa top-1M site urls" with each containing 40 pieces of data. Each data segment occupies 4096B. The data segments under the same url are stored into the same file and the 500 files are distributed in the CS of four routers (R1, R2, R3 and R4 contain 200, 75, 50 and 500 files, respectively). We configure static routes in FIB to achieve the minimal forwarding capability (especially for R1's two output ports). During the emulation, the queue capacity between two threads and between two processes are both set to hold 20000 packets by default. The CS bypass threshold is set to 64 by default. The default traffic generation speed is set to 100kpps.

Since NB-Cache has several novel design choices, for comprehensive comparison, we build four test cases to evaluate the performance improvement brought by each design choice (Fig. 10). Among them, case A is a classic content router without using Bloom filter (instead, it uses a content index for request prefiltering implemented by a red-black tree via the Map structure in C++ STL), AQM and non-blocking I/O; case B is an NB-Cache-enabled router; case C adopts Bloom filter but uses blocking I/O; case D adopts Bloom filter, non-blocking I/O but without using AQM. We also evaluate case A and case B working with ECN and NB-CC.

The evaluation is conducted on a desktop with Intel i5-6500 quad-core CPU, 8GB DRAM, 256GB SSD, 1TB HDD. The OS is Ubuntu 16.04 with Linux kernel 4.4.

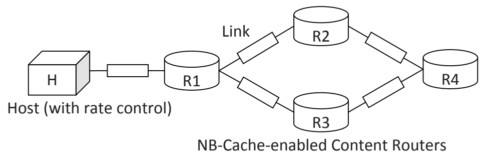


Figure 9: Topology of NB-Cache-enabled routers. We use one process to implement a router/host/link and shared memory for inter-process communication.

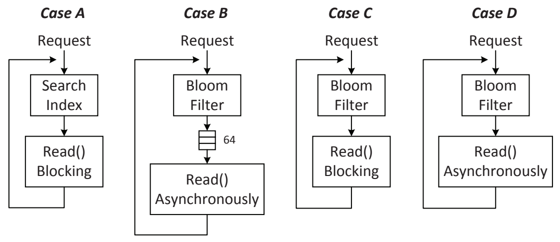


Figure 10: Four test cases for performance comparison.

5.2 Experimental Results

5.2.1 Single-node performance. Table 2 shows the packet latency of different processing paths inside a single router (R1) of blocking and non-blocking in-network caching (*i.e.*, case A and case B). In case A, 40% of the packets complete the I/O access with an average latency of 0.1934ms and 60% of the packets bypass the CS due to the early misses of the content index with an average latency of 0.00639ms. In case B, 20.05% of the packets issue the I/O access command but return immediately with an average latency of 0.0439ms and the same group of the packets finally complete the I/O access with a pretty long average latency of 18.699ms. The overlong latency is ascribed to the thread manipulation overhead according to our debugging. In case B, 60% of the packets bypass the CS due to the early misses of the Bloom filter and 19.95% of the packets bypass the I/O access due to the traffic congestion at the I/O queue. Although the thread manipulation overhead increases the I/O access latency, the bypass techniques via the Bloom filter and the active queue management (AQM) as well as the non-blocking I/O mechanism can potentially reduce the end-to-end queuing latency. We will see extensive RTT measurement results in later subsections.

Table 2: Packet Latency in a Single Router (R1) of Blocking and Non-Blocking Caching (Comparing Case A and Case B)

Processing Paths	Blocking (Case A)	Non-Blocking (Case B)
I/O Issue and Return	——	0.0439ms (20.05%)
I/O Complete	0.1934ms (40%)	18.699ms (20.05%)
BF/Index Miss Bypass	0.00639ms (60%)	0.009ms (60%)
Queue Full Bypass	——	0.0916ms (19.95%)

5.2.2 Content store resource contention. Fig. 11 shows the Interest queue length and the Data queue length when resource contention occurs during the I/O access. Specifically, in Fig. 11(a), the contention resolution strategy is set as first-in-first-out (FIFO) thus the Interest has the same priority with the Data. In this situation, the queue length of both types of the packets grows only depending on the real-time traffic congestion. In Fig. 11(b), when the contention resolution strategy is changed to assign a higher priority to the processing of the Data, the Data queue length drops sharply

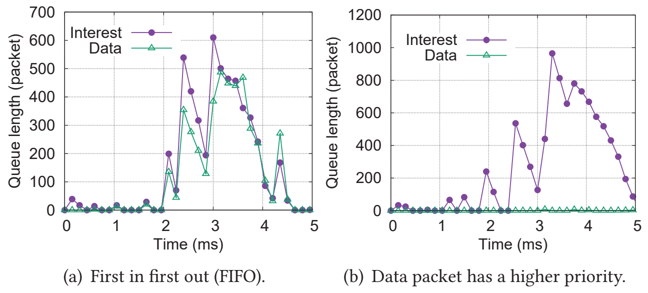


Figure 11: Interest queue length vs Data queue length.

while the thread for Interest processing is blocked frequently. In real ICN deployment, we can *tame* the ingress/egress performance by flexibly adapting the data plane traffic prioritizing strategies.

5.2.3 Round-trip time. Fig. 12 shows how the packet arrival rate impacts the RTT without congestion control. As the packet arrival rate grows, the RTT of the four cases increase as well. The reason lies in that when the packet arrival rate grows, the number of queuing packets in routers will increase rapidly, which contributes significantly to the average RTT. Among the four cases, NB-Cache outperforms the other solutions thanks to the CS bypass techniques. Under the 100kpps packet arrival rate, the RTT in case B and case A are 1122.5ms and 3754.5ms, respectively. NB-Cache can tremendously reduce the RTT by 70.10% compared with the classic architecture. Fig. 13 and 14 show the probability distribution of per-packet-based RTT in cases with/without congestion control. We can figure out that applying congestion control can remarkably reduce the average RTT from around 1000ms (Fig. 13) to 100ms (Fig. 14) by decreasing the number of queuing packets. In Fig. 14, the average RTT of case A+ECN, B+ECN and B+NB-CC are 86.23ms, 107.82ms and 125.48ms, respectively. Detailed analysis of Fig. 14 can be found in §5.2.7.

5.2.4 Throughput. Fig. 15 shows the average throughput in cases with/without congestion control. The throughput is measured as the number of Data packets divided by the time between the first and the last Data packet received by the consumer. Among the cases without congestion control, NB-Cache (case B) outperforms the other three cases. The average throughput in case A and case B are 4.56kpps and 10.51kpps, respectively. NB-Cache can improve the throughput by 130.48% compared with the original design. Specifically, we can figure out that non-blocking I/O access plays a dominant role in the throughput improvement; while using an additional Bloom filter and conducting active queue management also have sound contribution. Generally, applying congestion control will degrade the throughput but generate a more *balanced* performance (with much lower RTT). Among the cases with congestion control, case B+NB-CC has the highest throughput due to delayed congestion notification. Detailed analysis can be found in §5.2.7.

5.2.5 Impact of the CS bypass threshold. Fig. 16 shows the impact of the CS bypass threshold on RTT with/without congestion control. For case B, we can observe that when the threshold is set from 0 to 400, the average RTT decreases; when set from 400 to 600, interestingly, the RTT increases again. The observation can be explained considering two *extreme* cases. If the threshold is set to 0,

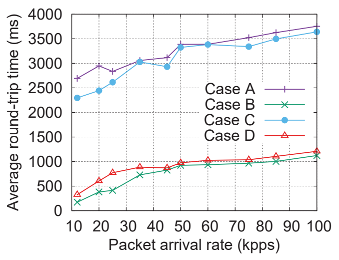


Figure 12: Average RTT vs packet arrival rate in four test cases.

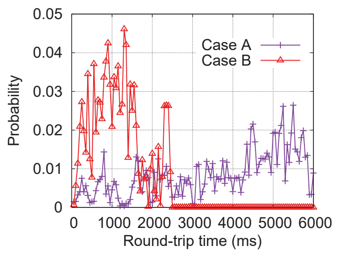


Figure 13: Probability distribution of RTT in case A and case B without congestion control.

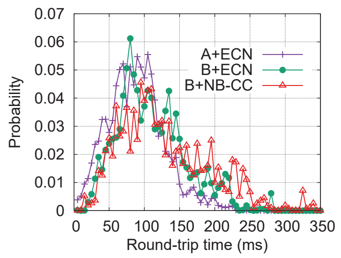


Figure 14: Probability distribution of RTT in cases with congestion control.

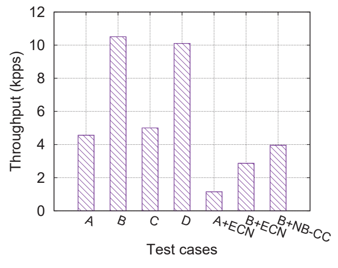


Figure 15: Average throughput in cases with/without congestion control.

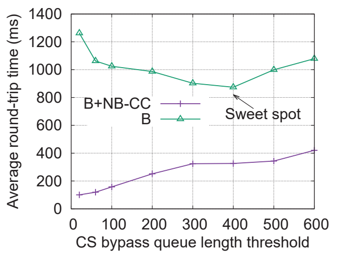


Figure 16: RTT vs CS bypass threshold with/without congestion control.

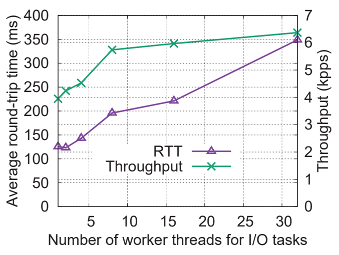


Figure 17: The impact of #threads on RTT and throughput (case B+NB-CC).

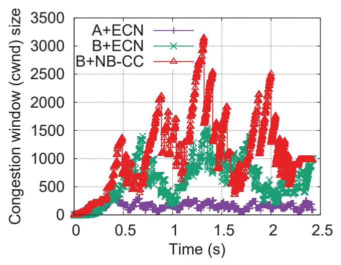


Figure 18: Dynamics of cwnd size in cases with congestion control.

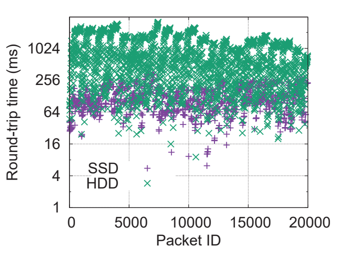


Figure 19: Per-packet-based RTT on SSD and HDD (case B+NB-CC).

all the packets will be forwarded to the upstream which will finally cause traffic congestion at R4 (R4 has all the content segments but no bypass possibility). If the threshold is set to ∞, all the packets will be queued at R1 rather than being distributed network wide. Here, 400 is a sweet spot for the CS bypass threshold. For case B+NB-CC, since the consumer will adaptively adjust the sending rate, larger bypass threshold just means delayed congestion notification, which will further cause longer RTT.

5.2.6 Impact of the number of spawned worker threads. As mentioned in §4.3.1, *libeio* will spawn a number of worker threads for non-blocking I/O access. Fig. 17 shows the impact of the number of spawned threads on RTT and throughput in case B+NB-CC. We can figure out that as the number of threads grows, both of the RTT and the throughput will increase. The reason lies in that excessive multithreading will *drain* I/O bandwidth thus improve the throughput. However, since i5-6500 is a quad-core CPU with only 4 hardware threads, excessive multithreading will also increase the thread switching and migration *overhead*, resulting in longer RTT. In our experimental settings, spawning more than 8 worker threads for each router is not recommended. Besides, here we have to emphasize that the absolute throughput (in pps) seems quite limited because we emulate all the routers on a single CPU and the Data packet/segment has a jumbo size (4096B).

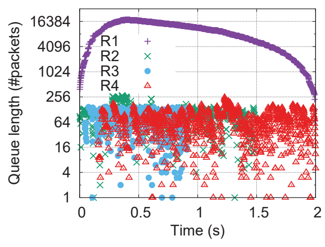
5.2.7 NB-CC vs ECN for congestion control. Fig. 18 shows the dynamics of congestion window size (cwnd). Case A+ECN has the minimum cwnd since traffic will be blocked at the first pipeline stage and router's CS queue will quickly reach ECN threshold for early congestion notification. Case B+ECN has a larger cwnd since NB-Cache allows some traffic to bypass the overloaded CS queue thus delays the time to notify congestion. Case B+NB-CC has the maximum cwnd since NB-CC delays congestion notification until the CS queue occupation reaches some threshold *and* there is no bypass possibility. Generally, larger cwnd means higher throughput and potentially longer latency (recall Fig. 15 and Fig. 14).

5.2.8 SSD vs HDD as CS storage medium. Fig. 19 shows the per-packet-based RTT when using SSD and HDD as the storage medium in case B+NB-CC. SSD is much faster than HDD, and the average RTT of SSD is only 125.48ms while that of HDD can reach 837.73ms. It is obvious that a single HDD without proper hierarchical design might be too slow to adequately accommodate CS in high-speed ICN networks.

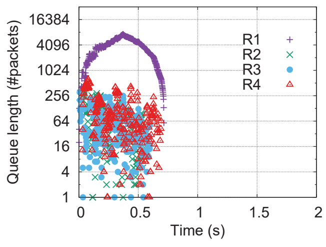
5.2.9 Network-wide load balancing. Fig. 20 shows traffic load distribution of the four routers in cases without congestion control. Since the traffic generation speed is higher than router's processing capability, packets inevitably accumulate in the front of the routers. In case A (Fig. 20(a)), most of the packets are blocked in R1's queue, and R2, R3, R4 have relatively idle queues. While in case B (Fig. 20(b)), due to CS bypass, R2, R3, R4 (especially R4) *share* considerable traffic load for R1, which lessens R1's queue occupation (the y axis is log scaling). With network-wide load balancing, NB-Cache (case B) completes the ingress processing of all the traffic in 0.7s while the classic architecture (case A) spends 2s.

6 RELATED WORK

Due to content router's stateful design, achieving wire-speed forwarding is challenging [11]. Since the data plane can be regarded as a three-stage pipeline, previous works mostly focus on accelerating the *component-level* performance at each pipeline stage. The challenge for FIB lies in performing the longest prefix match of string-based names of unbounded length. To address this technical issue, [18] proposes a GPU-based approach to implement wire-speed name lookup and achieves 63.52Mpps throughput. [16] adopts



(a) Case A (classic).



(b) Case B (NB-Cache).

Figure 20: Dynamics of queue length (#packets) in the front of the four routers.

the Patrica trie for FIB compression and achieves 284Gbps throughput based on SRAM. The challenge for PIT is to handle the frequent lookup and update of explosive pending states at line rate. [19] proposes a novel PIT design with an approximate data structure and achieves 100Gbps at a memory cost of 37MB.

Interestingly, although it is obvious that CS will also suffer from the flooding of content queries, to our best knowledge, there are only a *few* works addressing CS performance issues [9, 10, 15]. [9] makes use of memory hierarchy to scale in size and speed of the CS. Their design can sustain cache operations in excess of 10Gbps. [15] raises CS design principles with SSD but only provides very preliminary implementation details and experimental results. [10] exploits the temporal and spatial locality in content retrieval and proposes the locality-aware skip list which can achieve 1.796Mpps single-threaded throughput. Although with the similar target, NB-Cache tackles the performance issue from a *different* angle in a more cost-effective way. It redesigns CS's surrounding logic and router's internal packet flow to relieve CS performance burden thus can well *collaborate* with [9, 10, 15].

Most previous works treat the content router as a three-stage pipeline except for BFAST [5]. BFAST proposes a unified index which can simultaneously accommodate CS, PIT and FIB that can potentially improve the lookup speed by reducing the table access times. However, if implemented using multithreading on modern general-purpose processors, this design may have performance issue when multiple threads contend for a shared data structure. While NB-Cache retains the modularity of the original router architecture which is parallelism-friendly.

Honestly, we are not the first to add Bloom filter into the content router [14], but Bloom filter is indeed an integral part of our proposal. The active queue management in NB-Cache is actually inspired by the random early detection (RED) [7] in Internet routers. The key difference lies in that, when congestion occurs, NB-Cache chooses to forward the packets to the upstream while RED decides to drop the packets.

7 CONCLUSION

In this work, we quantitatively identify CS as content router's performance bottleneck and propose a novel traffic bypass architecture called "NB-Cache" to ameliorate the bottleneck. Distinct from previous works that conduct optimization for a single component, NB-Cache redesigns the entire router architecture, especially router's internal packet flow. Besides, we also propose congestion control for NB-Cache. In essence, NB-Cache follows a design pattern of on-demand load balancing. Evaluation shows that NB-Cache can remarkably improve the content delivery efficiency in ICN.

8 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thoughtful suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) (No. 61702049, 61872401, 61432009, 61702407, 61872213), the National Science Foundation (NSF) (No. CNS-1629009, CNS-1513505), the Research and Development Program in Key Areas of Guangdong Province (No. 2018B010113001), the National Science and Technology Major Project of the Ministry of Science and Technology of China (No. 2018ZX03001014-003), the Fundamental Research Funds for the Central Universities and the Huawei Innovation Research Program (HIRP).

REFERENCES

[1] 2019. libeio. http://software.schmorp.de/pkg/libeio.html.
[2] 2019. NB-Cache Repository. https://github.com/graytower/nbcache.
[3] 2019. Ring buffer. https://en.wikipedia.org/wiki/Circular_buffer.
[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[5] Huichen Dai, Jianyuan Lu, Yi Wang, and Bin Liu. 2015. BFAST: Unified and scalable index for NDN forwarding architecture. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2290–2298.
[6] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)* 8, 3 (2000), 281–293.
[7] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking* 4 (1993), 397–413.
[8] James R Jackson. 1963. Jobshop-like queueing systems. *Management science* 10, 1 (1963), 131–142.
[9] Rodrigo B Mansilha, Lorenzo Saino, Marinho P Barcellos, Massimo Gallo, Emilio Leonardi, Diego Perino, and Dario Rossi. 2015. Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM, 59–68.
[10] Tian Pan, Tao Huang, Jiang Liu, Jiao Zhang, Fan Yang, Shufang Li, and Yunjie Liu. 2016. Fast content store lookup using locality-aware skip list in content-centric networks. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 187–192.
[11] Diego Perino and Matteo Varvello. 2011. A reality check for content centric networking. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*. ACM, 44–49.
[12] K Ramakrishnan, S Floyd, and D Black. 2001. RFC 3168. *The addition of explicit congestion notification (ECN) to IP* (2001).
[13] Klaus Schneider, Cheng Yi, Beichuan Zhang, and Lixia Zhang. 2016. A practical congestion control scheme for named data networking. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM, 21–30.
[14] Junxiao Shi, Teng Liang, Hao Wu, Bin Liu, and Beichuan Zhang. 2016. Ndn-nic: Name-based filtering on network interface card. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM, 40–49.
[15] Won So, Taejoong Chung, Haowei Yuan, David Oran, and Mark Stapp. 2014. Toward terabyte-scale caching with SSD in a named data networking router. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 241–242.
[16] Tian Song, Haowei Yuan, Patrick Crowley, and Beichuan Zhang. 2015. Scalable name-based packet forwarding: From millions to billions. In *Proceedings of the 2nd ACM conference on information-centric networking*. ACM, 19–28.
[17] Matteo Varvello, Diego Perino, and Leonardo Linguaglossa. 2013. On the design and implementation of a wire-speed pending interest table. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 369–374.
[18] Yi Wang, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng, Huicheng Dai, Xin Tian, Zhonghu Xu, et al. 2013. Wire speed name lookup: A gpu-based approach. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 199–212.
[19] Haowei Yuan and Patrick Crowley. 2014. Scalable pending interest table design: From principles to practice. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2049–2057.
[20] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Patrick Crowley, Christos Papadopoulos, Lan Wang, Beichuan Zhang, et al. 2014. Named data networking. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 66–73.