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

Tian Pan, Xingchen Lin, Enge Song, *Graduate Student Member, IEEE*, Cheng Xu, Jiao Zhang, *Member, IEEE*, Hao Li, Jianhui Lv, Tao Huang, *Senior Member, IEEE*, Bin Liu, and Beichuan Zhang

***Abstract*— Information-Centric Networking (ICN) provides scalable and efficient content distribution at the Internet scale due to in-network caching and native multicast. 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). In this work, we build an analytical model of the router and identify that CS is the actual bottleneck. Then, we propose a novel mechanism called "NB-Cache" to address CS's performance issue from a network-wide point of view. 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. NB-Cache follows a design pattern of on-demand load balancing and can be formulated into a non-trivial N-queue bypass model. We use the Markov chain to establish its theoretical base and find an algorithm for automated transition rate matrix generation. Experiments show significant improvement of data plane performance: 70% reduction in round-trip time (RTT) and 130% increase in throughput. NB-Cache decouples the fast packet forwarding from the slower content retrieval thus substantially reducing CS's heavy dependency on fast but expensive memory.**

***Index Terms*— ICN, content router, bottleneck bypassing, non-blocking I/O, Bloom filter, N-queue bypass model.**

Manuscript received May 20, 2019; revised December 29, 2019 and May 14, 2020; accepted April 6, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Hay. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1802600; in part by the National Natural Science Foundation of China under Grant 61872401, Grant 62032013, and Grant 61872213; in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2019B1515120031; and in part by the Beijing University of Posts and Telecommunications (BUPT) Excellent Ph.D. Students Foundation under Grant CX2021301. A preliminary version of this work was presented at the 2019 IEEE/ACM International Symposium on Quality of Service (IWQoS'2019), June, 2019. *(Corresponding author: Tian Pan.)*

Tian Pan, Xingchen Lin, Enge Song, Cheng Xu, Jiao Zhang, and Tao Huang are with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications (BUPT), Beijing 100876, China (e-mail: pan@bupt.edu.cn; linxingchen@bupt.edu.cn; songenge@bupt.edu.cn; xu_cheng@bupt.edu.cn; jiaozhang@bupt.edu.cn; htao@bupt.edu.cn).

Hao Li is with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: hao.li@xjtu.edu.cn).

Jianhui Lv is with the International Graduate School at Shenzhen, Tsinghua University, Shenzhen 518057, China (e-mail: lvjianhui2012@163.com).

Bin Liu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: liub@tsinghua.edu.cn).

Beichuan Zhang is with the Department of Computer Science, The University of Arizona, Tucson, AZ 85721 USA (e-mail: bzhang@arizona.edu).

Digital Object Identifier 10.1109/TNET.2021.3083599

I. 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) [1] 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 more sophisticated compared with the stateless IP [2]. 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 [3]–[5]. 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 "NB-Cache", a novel solution addressing the performance issue from a global view instead of individual router's. In current ICN, 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 the routers after that one may not have any 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 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 is reduced. This *network-wide load balancing* leads to more efficient use of network resources, less congestion at routers, 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" (§II).
- 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 (§III).
- Theoretically, NB-Cache can be formulated into a novel *N-queue bypass model* which considers a unique queuing behavior that the traffic will *migrate* from the first pipeline stage to the next one if the first waiting line reaches a certain upper *limit*. We adopt the Markov chain to thoroughly analyze the problem and successfully find an *algorithm* to automatically generate its Markov transition rate matrix at any scale. The theoretical results are helpful for predicting NB-Cache performance when the number of content routers becomes large (§IV).
- We implement an NB-Cache-enabled router prototype with 3124 lines of C++ code (§V). All source code is *available* at out git repository [6]. In our experiment, NB-Cache can reduce RTT by 70% and improve throughput by 130% compared with the current ICN data plane under a network emulation testbed with four NB-Cache-enabled routers and synthetic content requests (§VI).
- NB-Cache decouples fast packet forwarding from slower content retrieval thus substantially reducing CS's rigid dependency on fast but expensive memory. This makes ICN economically feasible, since vendors can manufacture content routers with either high-end fast memories or low-end disk storages, tagged with different prices, but all can be deployed together. In NB-Cache, the packet forwarding performance can be increased linearly with the number of CPU cores or forwarding engines, while the content retrieval performance will improve with memory/storage technology promotion, today or in the future. They'll no longer be intertwined with each other.

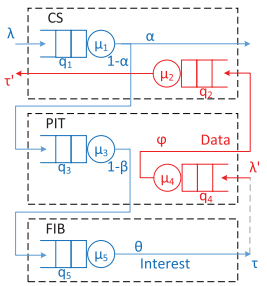


Fig. 1. Modeling a content router via queuing networks.

II. Where Is the Bottleneck?

A. Bottleneck Identification via Modeling

Observing content router's packet forwarding, we can figure out the following characteristics. The forwarding process can be divided into the ingress and the egress. In each part, packets are processed through multiple pipelined components. 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.

1) Assumptions: Assuming (i) in each direction, a component can be abstracted as a unit consisting of a FIFO *queue* for buffering and a *server* for 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) [7] to model the ICN forwarding process (as shown in Fig. 1). Table I lists the notations in our model.

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

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

Without considering packet losses, one Data packet always corresponds to one Interest packet. Hence, in the long run,

TABLE I
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

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

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

3) Identifying the Performance Bottleneck: According to the queuing theory, each server's utilization can be derived as

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

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

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

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

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

Obviously, given the traffic transfer probabilities are no larger than 1 and thus the denominators of the components containing $\mu_2, \mu_3, \ldots, \mu_5$ are no larger than 1, the component containing μ_1 is most likely to become the minimum. That is to say, the ingress part of CS will become the potential bottleneck of the 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 may still be idle, which makes the packet pipeline full of bubbles and inefficient. To squeeze pipeline bubbles, the service rate of each queuing subsystem (μ_i) should be budgeted ahead of time to let all the equality in Eq. (6) roughly hold simultaneously.

But if it is possible that the ingress speed 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 §VII that in recent research articles, the reported performance of FIB and PIT have already gone beyond 100Gbps [3]–[5] while that of CS is only around 10Gbps [8], [9]. Regardless of cost, we can put all hardware resources into the speedup of CS to achieve an ultimate CS implementation. 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.

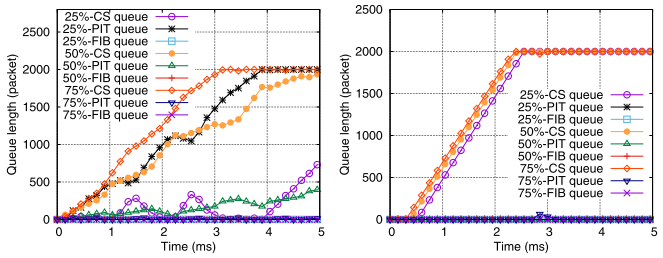
B. Bottleneck Identification via Prototyping

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 requests and the inter-component queue is implemented by a ring buffer. We fully implement CS using the data structures and the mechanisms described in §V. 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 25, 50 and 75%, respectively, for conducting sensitivity analysis.

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, in most of the time, 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)). Only at 25% CS hit rate, the PIT becomes the new bottleneck as most packets bypass CS processing and pour into PIT. In the HDD scenario (Fig. 2(b)), the situation becomes even worse as the queue in the front of CS gets overflowed quickly (with CS speed drops to 414pps) while PIT and FIB are still in a nearly idle state. Generally, promoting CS hit rate will flood more traffic into CS to increase its queue length. The experimental results are remarkable especially in the SSD scenario.

The prototype also convinces us that CS tends to become the choke point in the content router packet pipeline.

[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 storage 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).

Fig. 2. CS, PIT and FIB's queue length under 15.94kpps stress test (FIB and PIT's speed are fixed to 4kpps and CS hit rate is 25, 50, 75%, respectively).

III. Non-Blocking In-Network Caching

A. 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 value of 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 a 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 refinement should not modify router's interfaces to the outside to guarantee the *compatibility*, such that content routers with the improved architecture can be deployed *incrementally* with the classic content routers.

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 a 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 this mechanism "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 (first-stage bypass), (ii) enabling *active queue management* for I/O congestion avoidance (second-stage bypass), (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 network bandwidth utilization.

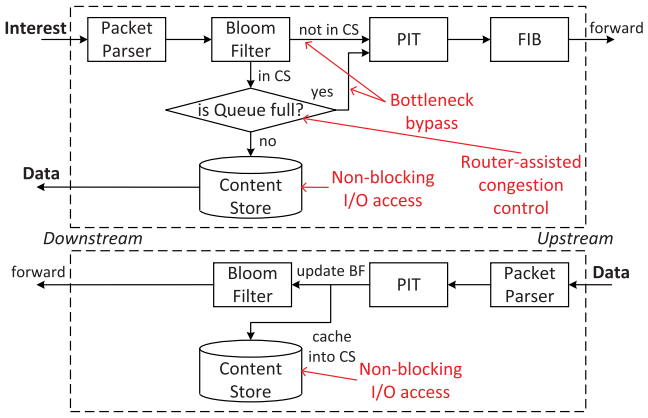


Fig. 3. A content router with non-blocking in-network caching capability.

B. Bloom Filter for Content Store Bypass (First-Stage Bypass)

A Bloom filter [10] is a 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 CS 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* [11], 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 packet processing.

C. AQM for I/O Congestion Avoidance (Second-Stage Bypass)

In Internet routers, active queue management (AQM), such as random early detection (RED) [12], 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

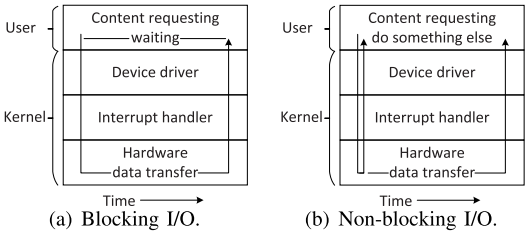


Fig. 4. Blocking I/O function calls do not return until the I/O is complete; non-blocking I/O calls return immediately, and the process can do something else and will be notified later when the I/O is complete.

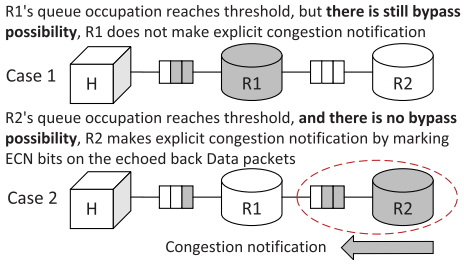


Fig. 5. Lazy (delayed) congestion notification in NB-CC.

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 §III-E).

D. Non-Blocking I/O for Fast Control Return

Since the Internet content to be cached is typically of bulky 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). Fig. 4 shows the comparison of blocking I/O and non-blocking I/O. 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 proposals, CS is implemented within DRAM [13], however, compared with the high-speed on-chip ASIC logic, the off-chip DRAM can still be considered as a slow I/O device relatively, and our design philosophy can still well apply. We disclose the detailed non-blocking I/O implementation of the content router in §V and illustrate the extensive experimental results in §VI.

E. 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 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 [14]. 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.

The design of NB-CC is described in detail as follows.

1) Nack Flooding for Hop-by-Hop Congestion Notification: In NB-CC, a router's outgoing interface will be labeled as unavailable if its upstream router (more close to content sources) suffers traffic congestion. When an upstream router's CS queue goes beyond the bypass threshold, it will *periodically* flood the Nack_α packets to the downstream adjacent router(s). Otherwise, when the CS queue goes below the threshold, it will flood the Nack_β packets to reset the label(s).

2) Outgoing Interface Unavailability Labeling: On receiving the Nack_α packet, the downstream router will update the adjacency table and label the *unavailability* of the outgoing interface(s) where the Nack_α comes in. On receiving the Nack_β packet, the corresponding outgoing interface(s) will be labeled as *available*.

3) ECN Marking for End-Host Congestion Notification: If the CS bypass threshold has not been reached, the Interest will be responded with a normal return Data. Otherwise, the Interest will bypass the overloaded CS given there is still a bypass possibility. The ECN marking on Data packets will be *delayed* until the CS bypass threshold has been reached and there is no available outgoing interface.

4) AIMD-Like End-Host Rate Adjustment: The consumer will keep sending Interests unless $unacked$ (the number of unacknowledged packets) exceeds $cwnd$ (the congestion window size). On sending an Interest, $unacked = unacked + 1$; on receiving a Data, $unacked = unacked - 1$. On receiving a Data without ECN marking, $cwnd = cwnd + 1$; on receiving a Data with ECN marking, $cwnd = cwnd/2$.

IV. Theoretical Analysis of NB-Cache

In this section, we formulate NB-Cache as an *N-queue bypass model* and adopt the *Markov chain* to find its theoretical base. Specifically, we find an algorithm to automatically generate the Markov transition rate matrix at any scale, which could help us estimate NB-Cache performance approximately when the number of content routers becomes very large.

A. The N-Queue Bypass Model

In order to find the theoretical base of NB-Cache, we make the following assumptions considering the *major* factors of

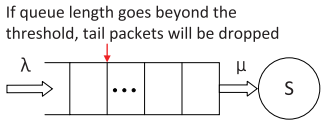


Fig. 6. Modeling conventional IP routers using a variant of the M/M/1 queue with a packet dropping threshold.

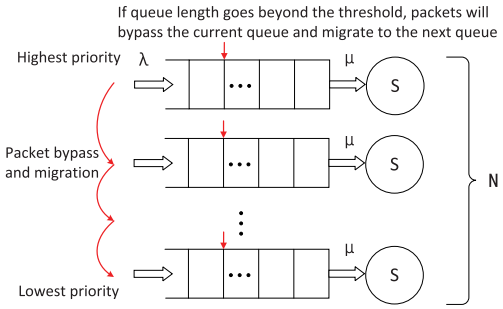


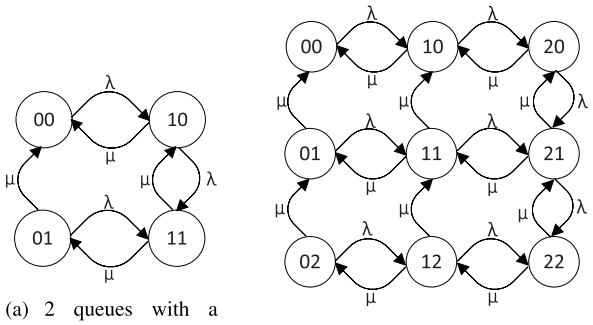
Fig. 7. Modeling NB-Cache using the proposed N-queue bypass model with packet bypass and migration.

the queuing behavior: (i) the packet arrival follows the Poisson process with the average arrival rate λ, (ii) the packet service time at each router follows the negative exponential distribution with the average service rate μ, (iii) we ignore the packet processing latency of PIT and FIB as the minor factors (compared with that of CS), (iv) the CS hit rate of each router is assumed to be 100% for mathematical simplicity.

We build the N-queue bypass model step by step, starting from a variant of the M/M/1 queue with a packet dropping threshold (as shown in Fig. 6). In fact, the queuing model in Fig. 6 can apply to either the conventional IP router or the classic content router since when the queue length of the bottleneck component reaches a pre-defined upper limit, the follow-up incoming packets will be discarded. Such a queuing behavior is slightly different from the classic M/M/1 queue model whose queue buffer is of *infinite* size.

In NB-Cache, if the queue length of the bottleneck component reaches a pre-defined upper limit, traffic will bypass the bottleneck and migrate from the local router to the next one. Such a unique queuing behavior can be depicted using multiple prioritized queues with traffic bypass threshold (Fig. 7). Given there are N queues ranked from the highest priority to the lowest priority (*i.e.*, queue$_i$ is with a higher priority than queue$_{i+1}$), different from the single-queue packet dropping model in Fig. 6, in NB-Cache, packets will arrive at queue$_0$ and continue to bypass the local overloaded queue (queue$_i$) and migrate to the next queue (queue$_{i+1}$) until all the N queues are heavily loaded. When all the queues are beyond the bypass threshold, the follow-up packets have to be dropped.

Let's first consider a simple case of the N-queue bypass model. Assuming that there are two queues (priority: queue$_0$ $>$ queue$_1$) and each has a bypass threshold of x packets. We define system state S_{ij} as the condition that there are i packets in queue$_0$ and j packets in queue$_1$. If $x = 1$, the state space can be represented as $\{S_{00}, S_{10}, S_{01}, S_{11}\}$. For each state S_{ij}, we define p_{ij} as the steady state probability of S_{ij} and we have $\sum_{i=0}^{x}\sum_{j=0}^{x} p_{ij} = 1$. Then, we can derive the theoretical values of several queuing system performance metrics.



(a) 2 queues with a bypass threshold of 1 packet (the 2*2 model).

(b) 2 queues with a bypass threshold of 2 packets (the 2*3 model).

Fig. 8. The Markov state transition diagrams of the N-queue bypass model.

1) The Number of Packets in the Queuing System: When the system is in state S_{ij} with probability p_{ij}, there will be $i + j$ packets in the queuing system. Hence, the expectation of the number of packets in the queuing system can be

$$y = \sum_{i=0}^{x}\sum_{j=0}^{x}(i+j)p_{ij} \tag{8}$$

2) Server Utilization: For a single queue, if the queue is empty, the server utilization is 0; otherwise, the server utilization is 1. Hence, the server utilization in state S_{ij} is

$$l_{ij} = \begin{cases} 1 & i > 0, j > 0 \\ 1/2 & i = 0, j > 0 \cup i > 0, j = 0 \\ 0 & i = 0, j = 0 \end{cases} \tag{9}$$

Thereby, the expectation of the server utilization of the entire queuing system can be derived as

$$l = \sum_{i=0}^{x}\sum_{j=0}^{x} l_{ij}p_{ij} \tag{10}$$

3) Packet Loss Rate: When all the queues reach the bypass threshold, the follow-up packets have to be dropped. Therefore, the packet loss rate of the two-queue bypass model is p_{xx} (*i.e.*, the steady state probability of system state S_{xx}), where x is the bypass threshold.

4) Extending the Model to Higher Dimension: For the three-queue bypass model, we can use S_{ijk} to represent the system state and p_{ijk} for the steady state probability. For an N-queue bypass model, there will be $(x+1)^N$ individual system states. If we could solve the *steady state probability* of each individual state, we can derive a large number of queuing system performance metrics even the system becomes complicated.

B. Continuous-Time Markov Chain and Transition Diagram

But how to calculate the steady state probability of each individual state? Actually, like the classic M/M/1 queue model, the N-queue bypass model can also be considered as a *continuous-time Markov chain*. Therefore, we can establish the *balance equations* of the specific markov chain for solving its steady state probabilities. Let's consider a simple example of the N-queue bypass model with two queues and a bypass threshold of only one packet (we name it "the 2*2 model" since there are two queues and each queue has two states specifying either 0 or 1 packet in the queue).

Fig. 8(a) shows the *Markov state transition diagram* of the 2*2 model. In the diagram, each state has both *outgoing (out)*

and *incoming (in)* transitions. For example, S_{10} has one out transition to S_{00} with the transition rate of μ and another out transition to S_{11} with the transition rate of λ; S_{10} also has one in transition from S_{00} with the transition rate of λ and another in transition from S_{11} with the transition rate of μ. Notice that there is *no* transition from S_{00} to S_{01} *since queue*$_0$ *has a higher priority than queue*$_1$. When the queuing system is in a steady state, for each state S_{ij}, the total in transition rate should be equal to the total out transition rate for establishing the balance equations. For the 2*2 model, we have the following established balance equations.

$$\begin{cases} \lambda p_{00} = \mu(p_{10} + p_{01}) \\ (\lambda + \mu)p_{01} = \mu p_{11} \\ (\mu + \lambda)p_{10} = \lambda p_{00} + \mu p_{11} \\ 2\mu p_{11} = \lambda(p_{10} + p_{01}) \end{cases} \tag{11}$$

On the other hand, the sum of the probability of all the system states should be 1 as

$$p_{00} + p_{10} + p_{01} + p_{11} = 1 \tag{12}$$

Finally, the steady state probability of each state in the 2*2 model can be solved with Eq. (11) and Eq. (12). Fig. 8(b) shows the Markov state transition diagram of the 2*3 model. And with the identical procedure, we can easily solve its state probability distribution.

C. Automated Markov Transition Rate Matrix Generation

From §IV-B we can figure out that if we can correctly draw the state transition diagram, we can always successfully solve the steady state probability distribution. However, when the number of system states explodes along with the content router number as well as the queue depth, one can hardly conceive the correct transition diagram at any scale all from the scratch. Here, we rely on a proposed algorithm for automated transition diagram construction at any scale which equally means automated *Markov transition rate matrix* generation.

1) Definitions: For the n*m model (with n queues and $m-1$ queue bypass threshold), there are m^n states in total (we use n_{id} to represent m^n for short). Each state can be represented as a string of integers as $\{a_1, a_2, \ldots, a_n\}$ (*e.g.*, we use {0,1,2,0,0} to denote state S_{01200}). The string length n represents n queues and each integer a_i in the string represents the number of packets in the i^{th} queue. To solve the probability of the n_{id} states, we need to establish n_{id} balance equations like Eq. (11) to quantify the equality between incoming and outgoing transitions. Actually, for each state, all the transitions can be divided into four categories: λ-out, μ-out, μ-in and λ-in. Among them, λ-out is the out transition due to packet arrival, μ-out is the out transition due to packet dequeue, μ-in includes in transitions due to packet dequeue, and λ-in includes in transitions due to packet arrival. Taking S_{10} in Fig. 8(a) as an example, S_{10} to S_{11}, S_{10} to S_{00}, S_{11} to S_{10} and S_{00} to S_{10} belong to λ-out, μ-out, μ-in and λ-in transitions, respectively. For a particular state, if we could calculate the exact number of the four types of transitions, the balance equation for that state can be well established. The exact number of the four types of transitions are calculated as follows.

2) λ-Out: For a given state, since queue$_i$ has a higher priority than queue$_{i+1}$, the newly arrived packet will be served in a *deterministic* queue, unless all the queues are beyond the threshold. We use a_{out_w} to indicate whether or not state w can transfer to the adjacent state in the state transition diagram by receiving a new packet. In most time, $a_{out_w} = 1$ (state transfer happens mostly). If and only if all the queues are beyond the threshold (*i.e.*, $w = \{m-1, m-1, \ldots, m-1\}$), $a_{out_w} = 0$.

3) μ-Out: We use n_{use_w} to represent the number of non-empty queues in state w. Obviously, for state w, the number of μ-out transitions is equal to n_{use_w}. For example, state {1, 1, 2, 0} has 3 μ-out transitions to {0, 1, 2, 0}, {1, 0, 2, 0} and {1, 1, 1, 0} due to packet dequeue.

4) μ-in: For incoming transitions, we need to consider the relationship between the current state and the other states from which the transitions come. We use $w = \{a_1, a_2, \ldots, a_n\}$ and $temp = \{b_1, b_2, \ldots, b_n\}$ to represent the current state and the other states, respectively. Obviously, state $temp$ can transfer to state w due to packet dequeue if and only if one packet is dequeued from one of the queues in state $temp$. That is to say, there should be exactly one -1 in vector $\{a_1 - b_1, a_2 - b_2, \ldots, a_n - b_n\}$ and the other integers in the vector should be all 0s. We use flag $n_{in_{w\&temp}}$ to indicate whether the above condition for the μ-in transition between $temp$ and w can hold. $n_{in_{w\&temp}} = 1$ if and only if $temp$ can transfer to w via the μ-in transition. This can be calculated by checking vector $\{a_1 - b_1, a_2 - b_2, \ldots, a_n - b_n\}$ based on the above rule.

5) λ-in: Similarly, if state $temp$ can transfer to w via the λ-in transition, there must be exactly one $+1$ in vector $\{a_1 - b_1, a_2 - b_2, \ldots, a_n - b_n\}$ and the other integers in the vector should be all 0s (*cond1*). However, *cond1* is *necessary but not sufficient*. Since in the N-queue bypass model, queue$_i$ has a higher priority than queue$_{i+1}$, if $temp$ can transfer to w due to packet arrival at queue$_j$, then, queue$_0$ to queue$_{j-1}$ must have already reached the threshold (*cond2*). Taking the 5*3 model as an example: $w = \{2, 2, 1, 1, 0\}$, $temp_1 = \{2, 1, 2, 1, 2\}$, $temp_2 = \{2, 2, 0, 1, 0\}$, $temp_3 = \{2, 2, 1, 0, 0\}$. It can be observed that $temp_1$ does not satisfy *cond1*, $temp_3$ satisfies *cond1* but does not satisfy *cond2*, only $temp_2$ satisfies both *cond1* and *cond2* thus can transfer to w via the λ-in transition. We use flag $a_{in_{w\&temp}}$ to indicate whether the above *cond1* and *cond2* can hold at the same time for $temp$ and w.

6) Balance Equations and Transition Rate Matrix: Now we can establish n_{id} balance equations for each individual state w (with steady state probability p_w) as

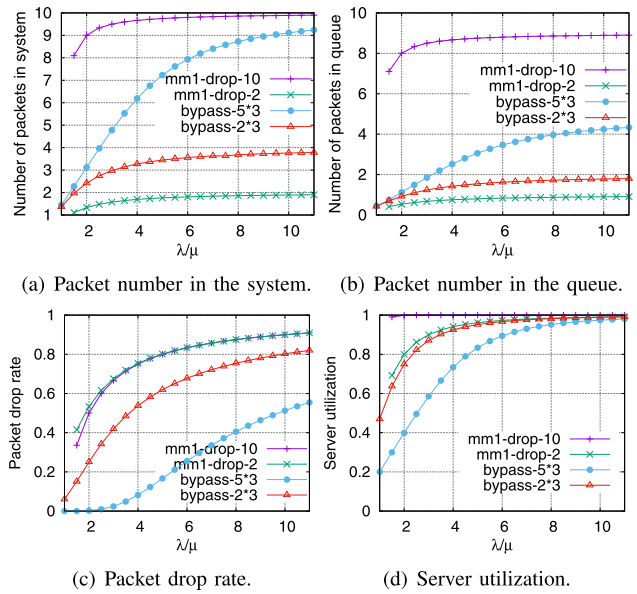
$$\begin{aligned} &(a_{out_w} * \lambda + n_{use_w} * \mu)p_w \\ &\quad = \sum_{temp_i \neq w} (a_{in_{w\&temp_i}} * \lambda + n_{in_{w\&temp_i}} * \mu)p_{temp_i} \end{aligned} \tag{13}$$

By plugging $a = \frac{\lambda}{\mu}$ into Eq. (13) and moving all variables to the left side, we have

$$\begin{aligned} &(a_{out_w} * a + n_{use_w})p_w \\ &\quad - \sum_{temp_i \neq w} (a_{in_{w\&temp_i}} * a + n_{in_{w\&temp_i}})p_{temp_i} = 0 \end{aligned} \tag{14}$$

If we use $s_1, s_2, \ldots, s_{n_{id}}$ to represent each state and organize the state probabilities into a solution vector as $x = \{p_{s_1}, p_{s_2}, \ldots, p_{s_{n_{id}}}\}$, we can turn Eq. (14) into the $Ax = b$ format as (15), as shown at the bottom of the next page.

Actually, A is the *Markov transition rate matrix* of the N-queue bypass model. And from Eq. (15), we can always solve the steady state probability distribution at any scale.



(a) Packet number in the system. (b) Packet number in the queue.

(c) Packet drop rate. (d) Server utilization.

Fig. 9. Comparison between the N-queue bypass model and the variant of M/M/1 queue with dropping threshold.

Run-Time Complexity: Given n queues and $m-1$ queue bypass threshold, there are m^n states in the state transition diagram. It equally means there are m^{2n} elements in the corresponding Markov transition rate matrix. For each element in the matrix, we need to calculate its λ-out, μ-out, μ-in and λ-in transitions, which have the complexity of $O(n)$ due to the queue-by-queue comparison. Hence, the overall complexity of automated Markov transition matrix generation is $O(nm^{2n})$.

D. Numerical Results

We adopt Matlab's *simplify()* to obtain the algebraic simplification of the analytic solution.

Fig. 9 shows the performance comparison between the N-queue bypass model and the variant of M/M/1 queue with dropping threshold (as shown in Fig. 7 and Fig. 6, respectively). Among the test cases, mm1-drop-x is the single-queue dropping model with the threshold of x packets and bypass-n*m is the N-queue bypass model with n queues and the threshold of m-1 packets for each queue.

Fig. 9(a) and Fig. 9(b) show the number of packets in the system/queue, respectively. Although mm1-drop-10 and bypass-5*3 consume the same queue buffer (for fair comparison), bypass-5*3 has less packets in the system/queue, which also means a lower queuing latency. The reduced latency can be ascribed to the load balancing among multiple servers.

Fig. 9(c) shows the packet drop rate of the four test cases. We can observe that the N-queue bypass model has a lower packet drop rate than the single-queue dropping model. For the bypass model, a system with more queues means higher bypass possibility and lower packet drop rate.

Fig. 9(d) shows the server utilization (which has been defined in Eq. (10)) of the four test cases. The bypass model has a lower server utilization than the single-queue dropping model due to a larger queue number.

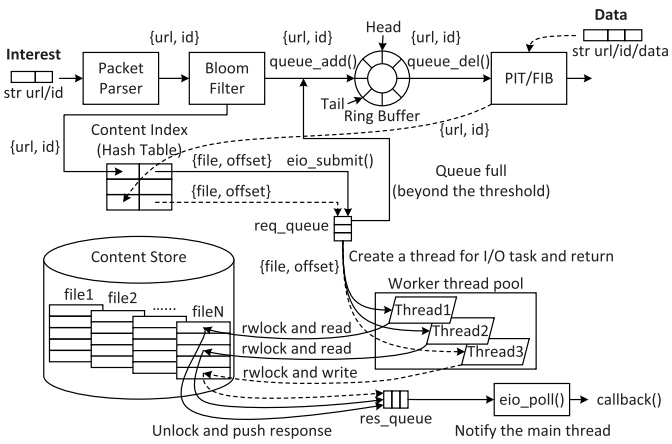


Fig. 10. Multithread-based implementation of the content router with non-blocking in-networking caching capability.

V. System Implementation

A. System Overview

Fig. 10 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, 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. For an incoming Data packet, it will be forwarded downstream and selectively written into CS. We use one *process* to implement a 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 $libeio$ [15] under Linux to implement

$$\begin{pmatrix} a_{out_{s_1}} * a + n_{use_{s_1}} & \cdots & -(a_{in_{s_1\&sn_{id}}} * a + n_{in_{s_1\&sn_{id}}}) \\ \vdots & \ddots & \vdots \\ -(a_{in_{sn_{id}\&s_1}} * a + n_{in_{sn_{id}\&s_1}}) & \cdots & a_{out_{sn_{id}}} * a + n_{use_{sn_{id}}} \end{pmatrix} x = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \quad (15)$$

the non-blocking I/O. libeio manages a pool of OS threads and conducts the polling operation during I/O access.

B. Key Data Structures

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.

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.

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) 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.

C. Main Control Logic

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 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.

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. Fig. 11 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 becomes available. When I/O is complete, an unlock operation via $pthread\_rwlock\_unlock()$ is performed. Currently, we lock in the granularity of *files*. If two requests fall into two files, no lock operation is needed.

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.

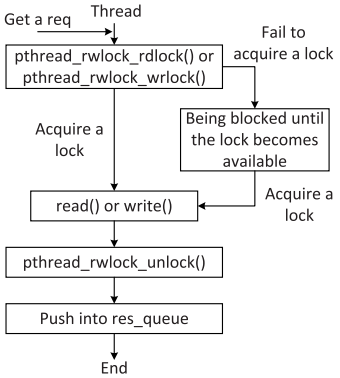


Fig. 11. Resolving interest/data contention via the readers-writer lock.

3) Inter-Component Communication: Fig. 12 shows the inter-component communication implemented via the *producer-consumer lock* mechanism on a ring buffer.

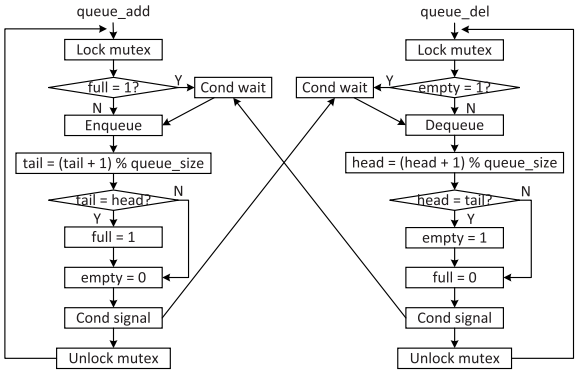


Fig. 12. Inter-component communication via the producer-consumer lock mechanism with a ring buffer.

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) 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.

VI. Evaluation

A. 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 [6]. The network emulation topology is shown in Fig. 13. 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 (for simplicity, we do not rely on real packet I/O like pf_ring or dpdk, instead, we adopt inter-process communication mechanism to mimic the packet I/O). 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 readily 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 a variable number of data segments according to the actual homepage size (ranging from 15.9KB to 26.2MB, with average size 3.6MB). 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.

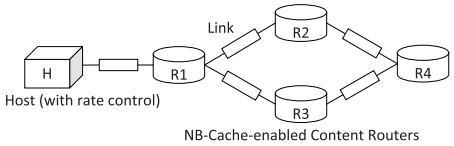


Fig. 13. Topology of NB-Cache-enabled routers. We use one process to implement a router/host/link and shared memory for inter-process communication.

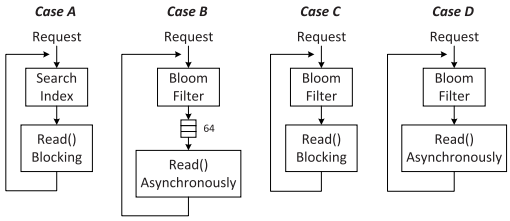


Fig. 14. Four test cases for performance comparison.

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 (as shown in Fig. 14). 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 network emulation 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. We also build a router prototype based on a more powerful machine with two E5-2686 v4@2.3GHz CPUs and 128GB DRAM.

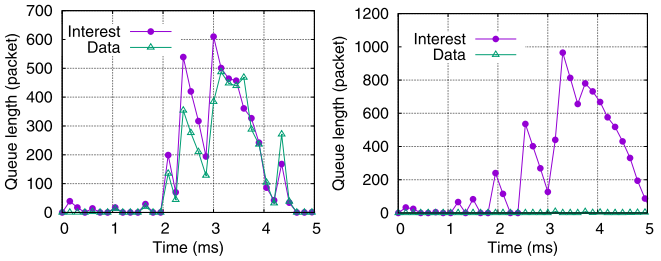
B. Experimental Results

1) Single-Node Performance in Emulation: Table II 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

TABLE II

PACKET LATENCY IN A SINGLE ROUTER (R1) OF BLOCKING AND NON-BLOCKING CACHING (COMPARISON OF 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%)



(a) First in first out (FIFO). (b) Data packet has a higher priority.

Fig. 15. Interest queue length vs Data queue length.

I/O mechanism can potentially reduce the end-to-end queuing latency. We will see extensive RTT measurement results later.

2) Content Store Resource Contention: Fig. 15 shows the Interest queue length and the Data queue length when resource contention occurs during the I/O access. Specifically, in Fig. 15(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. 15(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 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.

3) Round-Trip Time: Fig. 16 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. 17 and 18 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. 17) to 100ms (Fig. 18) by decreasing the number of queuing packets. In Fig. 18, 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. 18 can be found in §VI-B.7.

4) Throughput: Fig. 19 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

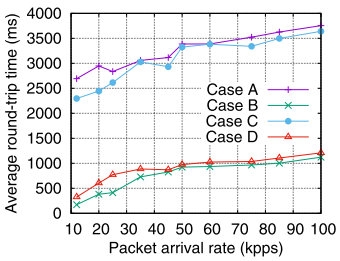


Fig. 16. Average RTT vs packet arrival rate in four test cases.

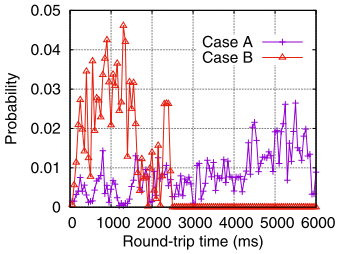


Fig. 17. Probability distribution of RTT in case A and case B.

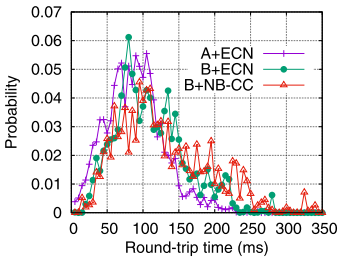


Fig. 18. Probability distribution of RTT in cases with congestion control.

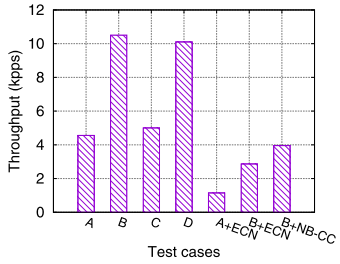


Fig. 19. Average packet throughput in cases with/without congestion control.

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 three cases with congestion control, case B+NB-CC has the highest throughput due to delayed congestion notification. Detailed analysis can be found in §VI-B.7.

5) Impact of the CS Bypass Threshold: Fig. 20 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, all the packets

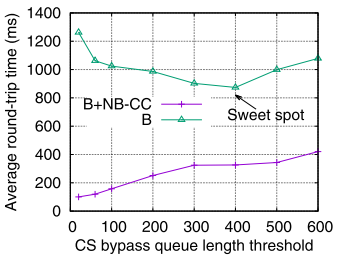


Fig. 20. RTT vs CS bypass threshold with/without congestion control.

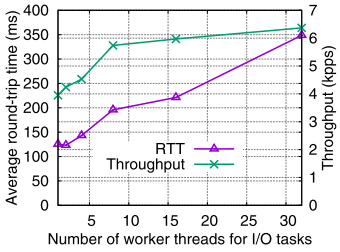


Fig. 21. The impact of #threads on RTT and throughput (case B+NB-CC).

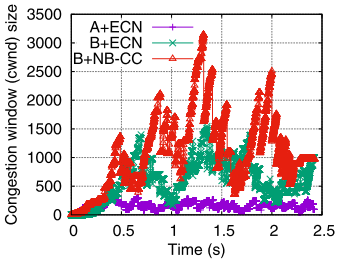


Fig. 22. Dynamics of cwnd size in cases with congestion control.

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.

6) Impact of the Number of Spawned Worker Threads: As mentioned in §V-C.1, *libeio* will spawn a number of worker threads for non-blocking I/O access. Fig. 21 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).

7) NB-CC Vs ECN for Congestion Control: Fig. 22 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

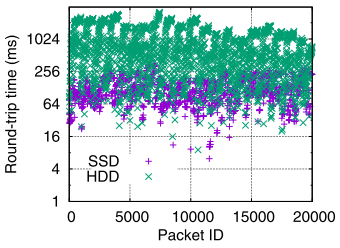
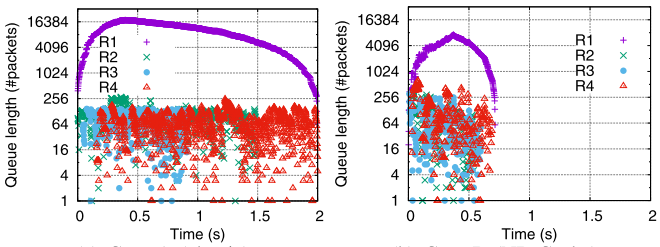


Fig. 23. Per-packet-based RTT on SSD and HDD (case B+NB-CC).



(a) Case A (classic). (b) Case B (NB-Cache).

Fig. 24. Dynamics of queue length (#packets) in the front of the four routers.

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. 19 and Fig. 18).

8) SSD Vs HDD as CS Storage Medium: Fig. 23 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.

9) Network-Wide Load Balancing: Fig. 24 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. 24(a)), most of the packets are blocked in R1's queue, and R2, R3, R4 have relatively idle queues. While in case B (Fig. 24(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.

10) A Router Prototype With Different CS Storage Mediums: All above evaluation is conducted in the emulation environment. However, we are also curious about NB-Cache's real performance on bare-metal hardware, which is considered important for real-world deployment. More specifically, we want to see: (a) NB-Cache's forwarding performance on the multicore architecture, which is widely leveraged by today's network processors embedded in high-speed routers; (b) CS lookup performance under excessive multithreading with different storage mediums (SSD and ramfs). To achieve this, we build a router prototype with multiple NB-Cache instances in parallel on a 36-core x86 machine. The ramfs is Linux's RAM-based file system which has an even faster speed than SSD. The CS hit rate is set to 100% to maximally

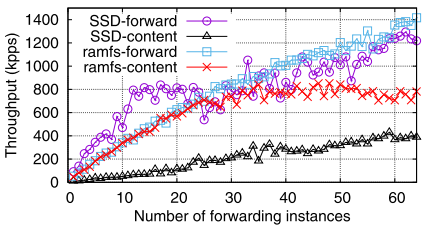


Fig. 25. The (decoupled) forwarding throughput and content retrieval throughput with different CS storage mediums (SSD, ramfs) on a 36-core machine.

exploit the I/O bandwidth. Fig. 25 shows the forwarding and content retrieval throughput with SSD and ramfs. It is observed that (a) the forwarding throughput increases roughly linearly with the number of forwarding instances; (b) the excessive multithreading drains CS bandwidth regardless of storage medium types as the number of forwarding instances grows; (c) the forwarding performance is totally decoupled from the content retrieval performance due to the non-blocking mechanism (the forwarding performance grows linearly while the CS lookup performance is bounded by the I/O bandwidth); (d) interestingly, although ramfs has a higher CS throughput (0.8Mpps), its forwarding performance (1.3Mpps) is lower than that of SSD (1.4Mpps) (this can be explained as the asynchronous I/O operations rely on additionally spawned threads, which also consume processing cycles of the CPU thus the remaining cycles for packet forwarding become less). We show that NB-Cache can achieve high performance with massive multicore machine, competitive for real deployment. More than that, NB-Cache can also be deployed with low-end machines, without blocking the traffic along the end-to-end path. The latter one is the most exciting part of our work and has never been mentioned in existing literature. By decoupling the fast packet forwarding from the slower content retrieval, NB-Cache-enabled routers with different memory/disk technologies can coexist in the same network, providing best-effort content caching service economically.

VII. Related Work

Due to content router's stateful design, achieving wire-speed forwarding is challenging [2]. 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, [3] proposes a GPU-based approach to implement wire-speed name lookup and achieves 63.52Mpps throughput. Reference [4] adopts 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. Reference [5] 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 [8], [9], [16]. Reference [8] makes use of memory hierarchy to scale in size and speed of the CS. Their design can sustain cache operations in excess of 10Gbps. Reference [16] raises CS design principles with SSD but only provides very preliminary implementation details and experimental results. Reference [9] 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* perspective in a more cost-effective way. It modifies CS's surrounding logic and router's internal packet flow to relieve CS performance burden thus can well *collaborate* with [8], [9], [16].

Most previous works treat the content router as a three-stage pipeline except for BFAST [17]. 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 one shared data structure (*i.e.*, the unified index). While NB-Cache retains the modularity of the original router architecture which is rather parallelism-friendly.

In either CDN or NDN space, cache federation [18], [19] is widely adopted to allocate local space for storing locally popular objects and remote federated space for storing less popular objects. The content request will first consult the local cache. If the data is not present locally, the node will subsequently consult the remote federated cache. While cache federation aims at reducing the popular content access latency as well as the upstream link utilization to further save the link usage expense, NB-Cache focuses more on improving the *forwarding plane* performance by mitigating the congestion in the front of CS. Even the requested content objects exist in the local CS, traffic bypass will still be triggered given CS is heavily used. Besides, cache coordination in NB-Cache is guaranteed naturally by the underlying routing mechanism without the explicit need of stateful cache federation algorithms.

Honestly, we are not the first to add Bloom filter into the content router [20], 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) [12] 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.

Historically, various queuing models are proposed to analyze diverse queuing behaviors. For example, the models considered in [21], [22] involve a tandem queue with two waiting lines, and as soon as the second waiting line reaches a certain upper limit, the first line will be blocked. Unlike the tandem queue models, our N-queue bypass model considers another queuing behavior that the traffic will *migrate* from the first pipeline stage to the next one if the first waiting line reaches a certain upper limit. In [23], an on-demand load balancing model is proposed for the "balls into bins with repeated tosses" scenario, in which if a ball is tossed into a fully occupied bin, it will be tossed repeatedly into the bins again until an empty or a partly loaded bin is met. In this scenario, the number of balls in each bin will always accumulate and never decrease. Essentially, [23] can be regarded as load balancing among multiple queues *without* servers for the dequeue operation, while our model depicts load balancing among multiple queues *with* servers for item processing (*i.e.*, the dequeue operation).

VIII. Conclusion

In this work, we quantitatively identify CS as content router's performance bottleneck and propose a novel traffic

bypass design called "NB-Cache" to ameliorate the bottleneck. Distinct from previous works that optimize for a single component, NB-Cache modifies CS's surrounding logic and diverts 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 and we adopt the Markov chain to establish its theoretical base. Evaluation shows that NB-Cache can remarkably improve the content delivery efficiency and reduce the hardware cost.

REFERENCES

[1] L. Zhang *et al.*, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.

[2] D. Perino and M. Varvello, "A reality check for content centric networking," in *Proc. ACM SIGCOMM Workshop Inf.-Centric Netw. (ICN)*, Aug. 2011, pp. 44–49.

[3] Y. Wang *et al.*, "Wire speed name lookup: A GPU-based approach," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 199–212.

[4] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: From millions to billions," in *Proc. 2nd ACM Conf. Inf.-Centric Netw.*, Sep. 2015, pp. 19–28.

[5] H. Yuan and P. Crowley, "Scalable pending interest table design: From principles to practice," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2014, pp. 2049–2057.

[6] (2020). *Nb-Cache Repository*. [Online]. Available: https://github.com/graytower/nbcache

[7] J. R. Jackson, "Jobshop-like queueing systems," *Manage. Sci.*, vol. 10, no. 1, pp. 131–142, Oct. 1963.

[8] R. B. Mansilha *et al.*, "Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation," in *Proc. 2nd ACM Conf. Inf.-Centric Netw.*, Sep. 2015, pp. 59–68.

[9] T. Pan *et al.*, "Fast content store lookup using locality-aware skip list in content-centric networks," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 187–192.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[11] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[12] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[13] J. Takemasa, Y. Koizumi, and T. Hasegawa, "Toward an ideal NDN router on a commercial off-the-shelf computer," in *Proc. 4th ACM Conf. Inf.-Centric Netw.*, Sep. 2017, pp. 43–53.

[14] K. Schneider, C. Yi, B. Zhang, and L. Zhang, "A practical congestion control scheme for named data networking," in *Proc. 3rd ACM Conf. Inf.-Centric Netw.*, Sep. 2016, pp. 21–30.

[15] (2019). *Libeio*. [Online]. Available: http://software.schmorp.de/pkg/libeio.html

[16] W. So, T. Chung, H. Yuan, D. Oran, and M. Stapp, "Toward terabyte-scale caching with SSD in a named data networking router," in *Proc. 10th ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS)*, Oct. 2014, pp. 241–242.

[17] H. Dai, J. Lu, Y. Wang, and B. Liu, "BFAST: Unified and scalable index for NDN forwarding architecture," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 2290–2298.

[18] A. Belloum, L. O. Hertzberger, and H. Muller, "Scalable federation of Web cache servers," *World Wide Web*, vol. 4, no. 4, pp. 255–275, 2001.

[19] J. Li, H. Wu, B. Liu, and J. Lu, "Effective caching schemes for minimizing inter-ISP traffic in named data networking," in *Proc. IEEE 18th Int. Conf. Parallel Distrib. Syst.*, Dec. 2012, pp. 580–587.

[20] J. Shi, T. Liang, H. Wu, B. Liu, and B. Zhang, "NDN-NIC: Name-based filtering on network interface card," in *Proc. 3rd ACM Conf. Inf.-Centric Netw.*, Sep. 2016, pp. 40–49.

[21] W. K. Grassmann and S. Drekic, "An analytical solution for a tandem queue with blocking," *Queueing Syst.*, vol. 36, no. 1, pp. 221–235, 2000.

[22] N. D. van Foreest, J. C. W. van Ommeren, M. R. H. Mandjes, and W. R. W. Scheinhardt, "A tandem queue with server slow-down and blocking," *Stochastic Models*, vol. 21, nos. 2–3, pp. 695–724, Jan. 2005.

[23] T. Pan, B. Liu, X. Guo, Y. Li, and H. Song, "Bandwidth-greedy hashing for massive-scale concurrent flows," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2016, pp. 659–668.

Tian Pan received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2015. He is currently an Associate Professor with the Beijing University of Posts and Telecommunications. His main research interests include data center networks, programmable data plane, and satellite networks.

Xingchen Lin received the B.S. and M.S. degrees from the Beijing University of Posts and Telecommunications in 2017 and 2020, respectively. He is currently an Engineer with Alibaba Cloud. His main research interests include network measurement, programmable data plane, and cloud networks.

Enge Song (Graduate Student Member, IEEE) received the B.S. degree from the Beijing University of Posts and Telecommunications in 2017, where he is currently pursuing the Ph.D. degree with the School of Information and Communication Engineering. His main research interests include network measurement, programmable data plane, and machine learning.

Cheng Xu received the B.S. degree from the Beijing University of Posts and Telecommunications in 2018, where she is currently pursuing the M.S. degree with the School of Information and Communication Engineering. Her main research interests include ICN, programmable data plane, and network function virtualization.

Jiao Zhang (Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2014. She is currently an Associate Professor with the Beijing University of Posts and Telecommunications. Her main research interests include data center networks, network function virtualization, and future Internet architecture.

Hao Li received the Ph.D. degree in computer science from Xi'an Jiaotong University in 2016. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His main research interests include SDN/NFV and network measurement.

Jianhui Lv received the Ph.D. degree in computer science from Northeastern University in 2017. From 2018 to 2019, he worked at Huawei Technologies, as a Senior Engineer. He is currently an Assistant Research Fellow with Tsinghua University. His main research interests include ICN, the IoT, and edge computing.

Tao Huang (Senior Member, IEEE) received the Ph.D. degree in communication and information systems from the Beijing University of Posts and Telecommunications in 2007. He is currently a Professor with the Beijing University of Posts and Telecommunications. His current research interests include network architecture, routing and forwarding, and network virtualization.

Bin Liu received the Ph.D. degree in computer science and engineering from Northwestern Polytechnical University in 1993. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University. His current research areas include high-performance switches/routers, network processors, and network architecture.

Beichuan Zhang received the B.S. degree from Peking University in 1995 and the Ph.D. degree from the University of California at Los Angeles (UCLA) in 2003. He is currently an Associate Professor with the Department of Computer Science, The University of Arizona. He has been working on Internet routing architectures and protocols, green networks, routing security, and Internet content distribution.