Efficient latency control in Fog deployments via hardware-accelerated popularity estimation

MARCEL ENGUEHARD*, YOANN DESMOUCEAUX†, and GIOVANNA CAROFIGLIO, Cisco Systems, France

Introduced as an extension of the Cloud at the network edge for computing and storage purposes, the Fog is increasingly considered a key enabler for Internet-of-Things applications whose latency requirements are not compatible with a Cloud-only approach. Unlike Cloud platforms, which can elastically accommodate large numbers of requests, Fog deployments are usually dimensioned for an average traffic load and, thus, unable to handle sudden bursts of requests without violating latency guarantees. In this paper, we address the problem of efficiently controlling Fog admission to guarantee application response time. We propose request-aware admission control (AC) strategies maximizing the number of Fog-handled requests by means of dynamic popularity estimation. In particular, the *LRU-AC*, an AC strategy based on online learning of the request popularity distribution via a Least Recently Used (LRU) filter, is introduced. We contribute an analytical model for assessing LRU-AC performance and quantifying the incurred reduction of Cloud offload cost, w.r.t. both an ideal oracle-based and a request-oblivious AC strategy. Further, we propose a feasible implementation design of LRU-AC on FPGA hardware using Ageing Bloom Filters (ABF) to mimic the function of the LRU-AC, while providing a compact memory representation. The use of ABFs for LRU-AC is theoretically validated and verified through simulation. The current implementation shows a throughput of 16.7 Mpps and a processing latency of less than 3 µs while multiplying the Fog acceptance-rate by 10 in the evaluated scenario.

ACM Reference Format:
Marcel Enguehard, Yoann Desmouceaux, and Giovanna Carofiglio. 2019. Efficient latency control in Fog deployments via hardware-accelerated popularity estimation. *ACM Trans. Internet Technol.* 1, 1, Article 1 (January 2019), 22 pages. https://doi.org/10.1145/3366020

1 INTRODUCTION

The emergence of low-latency applications in the context of the Internet of Things (IoT) has created a need for computing and storage platforms geographically and topologically close to the access network. IoT deployments used to typically rely on the *Cloud* (i.e., a large-scale, distributed, and elastic data center) to perform these functions, but the Cloud does not possess the required geographical properties. Introduced as an extension of the Cloud at the network edge (i.e., geographically near the users' devices, for instance in an ISP's premises) for computing and storage purposes [1], the *Fog* is a highly virtualized, potentially distributed, computing and storage platform capable of processing IoT data under low latency. Since its definition, there has been a growing interest in the research community for Fog computing, encompassing areas such as workload placement [2–4], caching [5, 6], or application profiling [7]. Accelerated by the advent of 5G, the Fog is expected to

*Also with Télécom ParisTech.
†Also with École Polytechnique.

Authors' address: Marcel Enguehard; Yoann Desmouceaux; Giovanna Carofiglio, Cisco Systems, Chief Technology and Architecture Office, 11 rue Camille Desmoulins, Issy-les-Moulineaux, 92130, France, firstname.lastname@cisco.com.

1533-5399/2019/1-ART1 $15.00
https://doi.org/10.1145/3366020

play a key role for IoT applications in cases where latency requirements and/or security reasons preclude the use of a Cloud-only approach [8].

Fog platforms are not intended as a replacement for Cloud processing, rather as a complementary computing and storage platform to use if and whenever beneficial [3]. For instance, with frameworks such as AWS Greengrass [9], IoT providers can use their own devices (e.g., an IoT gateway or a local compute node) to perform some stateless pre-processing of data on its path to the Cloud. In addition, the Fog does not enjoy the same elasticity as the Cloud. Indeed, while Cloud datacenters inherently scale due to their size and number of tenants, Fog nodes have physical limits that cannot be infringed [4]. Placing and scheduling of Fog applications to meet Quality of Service (QoS) requirements has thus been the object of recent studies [3, 4]. They, however, rely on centralized optimization and do not consider sudden bursts of requests, e.g., flash crowds, that can increase response time and raise scalability issues for Fog deployments. Such bursts of queries can be due to a user-led increase in demand for certain resources, but also to redirection of queries originally destined to another network for disaster mitigation [10, 11]. In such cases, the natural solution is to redirect part of the Fog load to the Cloud [12, 13]. However, if done incorrectly, that could actually worsen the response latency. Furthermore, renting resources in the Cloud is expensive, as opposed to user-owned Fog devices. This paper thus investigates the question of: *how to minimize Cloud costs while offering statistical latency guarantees in case of high load in a Fog network?*

IoT applications can roughly be categorized as: (i) *latency-critical*, when processed data must be received within 1-10 ms, (ii) *latency-sensitive*, where the timescale of user interaction is in the order of 100 ms [14], and (iii) *latency-tolerant*, that have no specific delay constraint. Whereas latency-critical applications cannot run in the Cloud (but rather in the device or at Fog level), latency-tolerant applications would naturally be deployed there. Thus, Cloud redirection is most relevant for applications of the latency-sensitive class, where the computing bottleneck in the Fog may force to offload part of request processing to the Cloud. At the same time, the use of faraway Cloud resources not only increases the *cost* for the operator but may also increase the *service latency*. There is thus a need for clever Fog admission control (AC) to keep response time bounded.

In [15], an approach for Fog AC is presented that relies on using request popularity to optimize the usage of the Fog platform, and in particular of the caching capacities of the Fog. Specifically, two AC strategies are introduced: the LFU-AC, which exploits *perfect knowledge* of the request popularity distribution and the LRU-AC, itself based on learning the most popular requests via a cache of request identifiers with the Least-Recently-Used (LRU) policy. Using an analytical model, the LFU-AC is shown to significantly reduce cost as compared to optimized request-agnostic AC strategies. The LRU-AC performance is also shown to be close to the LFU-AC bound.

In this paper, this work is extended by studying the system implementation of the LRU-AC strategy on programmable hardware, thus allowing for a line-rate, low-latency deployment that does not require provisioning extra computing resources. As implementing an LRU cache in hardware is inconvenient due to the necessity of using linked-list structures [16], an implementation of the LRU-AC using Ageing Bloom Filters (ABF) [17] in NetFPGA [18] is proposed. An analytical model of the expected hit-rate of the ABF is built to guarantee its behavior w.r.t. the LRU-AC. The use of an ABF is shown to reduce memory footprint by up to 4×, making the implementation fit within memory sizes available in programmable hardware devices. As one example of an underlying network protocol that can be used by this approach, the implementation relies on hICN [19], an Information-Centric Networking (ICN) protocol that uses the format of TCP/IP packets for backward compatibility. Using the P4 framework [20], request semantics are then efficiently accessible in hardware at the network layer (due to hICN's *name-based* forwarding with *fixed-sized* names), while relieving the AC from maintaining per-flow state by way of a *connection-less* approach. The implementation is shown to support a throughput of 16.7 Mpps (exceeding 10 GbE line-rate) with a packet processing

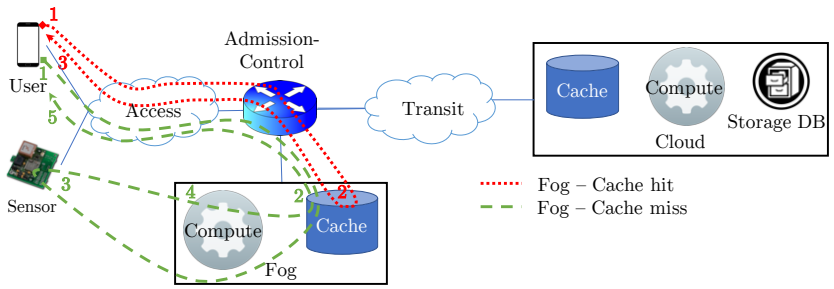


Fig. 1. Reference IoT, Fog and Cloud architecture

latency of 3 µs. Algorithmic performance of the ABF is evaluated and shown to produce QoS results that are consistent with the LRU-AC model.

The remainder of this paper is organized as follows: Section 2 states the Fog-AC problem in a principled way; Section 3 introduces the analytical model used to derive the performance of AC strategies; relevant popularity-based strategies are presented and partially evaluated in Section 4; in Section 5, the use of ABF to realize the LRU-AC, selected because it is both practical and effective, is proposed and justified; the LRU-AC hardware implementation is then introduced in Section 6, and is evaluated in Section 7.

2 PROBLEM DESCRIPTION

2.1 Reference Fog architecture

The reference IoT architecture assumed in this paper consists of three main components: (i) IoT networks, where sensors, actuators, and users are connected; (ii) an access network which connects these IoT networks together and with the Internet; and (iii) a Cloud platform, used for compute and storage. The Cloud platform is assumed to be a large-scale private or public data-center, with an infinite amount of resources. Workloads are assumed to be able to be automatically scaled up or down to handle variations in computing demands.

On top of this Cloud platform, a Fog deployment is available in the access network. Both the Fog and the Cloud are equipped with LRU caches. Each tenant of the Fog deployment is assumed to be allocated a *fixed* part of the computing resources. Therefore, without loss of generality, we adopt a per-application view and consider a single Fog node with computing and storage capabilities that receives requests from a single stateless application (e.g., lambda function). The Fog, either owned by the application developer or rented from its Internet Service Provider (ISP), is deployed for latency-critical applications. However, as Fog platforms are not elastic, the application developer wants to use the residual computing and memory capacities to perform latency-sensitive tasks. That residual capacity is considered to be constant, hence the Fog node cannot handle a high arrival rate for a prolonged period. It redirects part of its load to a Cloud platform, where the operator rents computing and storage resources, while still respecting the latency agreement.

An admission control module is in charge of forwarding incoming requests from the IoT networks to either Fog (accept) or Cloud (refuse). That architecture is summarized in Figure 1. We consider Fog applications working in the following way: (i) the application retrieves *raw data* from one or several sensor nodes (e.g., an image or a temperature from several sensors); (ii) it performs some computation to transform the raw data into *processed data* (e.g., JSON file indicating detected shapes, or the average of the measured temperatures); (iii) the processed data is retrieved by users or actuators which use it to make decisions. As security is paramount for Fog applications [1], both

processed and raw data are encrypted during network transmissions. In particular, we consider a pull-based model driven by client requests, of which we illustrate two of the possible paths in Figure 1. The user application starts by issuing requests (step 1), which reach the AC module. For the sake of illustration, only cases where requests are accepted for processing in the Fog are shown. In the Fog node, the request is matched against a cache (step 2) for the availability of processed data. In case of a *cache hit* (red dots · · ·), the processed data is sent back directly to the user. In case of a *cache miss* (green dashes - - -), the raw data must be retrieved from the sensor (step 3), before the computation can take place (step 4) and the processed data can then be served back to the user (step 5).

2.2 Fog vs Cloud admission control

To devise such strategies where the application runs both in the Cloud and in the Fog, costs for the Fog operator must be considered. In the Cloud, resources are elastic and the capacity can be increased as the incoming load grows [21]. Furthermore, the Cloud is assumed to always store the raw data for orthogonal archiving and monitoring purposes; the cost of raw data archival is thus not considered. This comes on top of the cache for processed data, whose size is defined by the amount of storage rented in the Cloud. Moving data to/from the Cloud through the transit network also has a cost. On the other hand, application deployment in the Fog comes at no extra monetary cost for the Fog operator (since they already own/rent the infrastructure). As Fog nodes have limited storage resources, the Fog cache is only used for processed data. The Fog node also has a finite amount of computing resources, which must be equally shared between all incoming requests. Thus, in case of a high load, the Fog node might have a high completion time or even start dropping requests, which may violate the agreement set up with the Fog application developer.

The need for a proper offloading function ϕ between the Fog and Cloud resources thus becomes clear: such a function should *minimize the cost* $\Pi(\phi)$ of renting Cloud resources while *respecting the agreed upon latency constraint*, which we outline below (and formalize in Section 3):

$$\begin{cases} \min. & \Pi(\phi) \\ \text{s.t.} & \mathbf{E}[T(\phi)] \leq \Delta \end{cases} \tag{1}$$

where T is the stochastic variable describing the system completion time. In particular, Π is a cost per second, in accordance with the pay-as-you-use model for Cloud pricing. We use a *statistical* latency constraint, which guarantees that the average request completion time is under Δ. The advantage of such formulation is clear considering that it enables us to express the constraint in closed form in a queueing model, which simplifies tractability.

3 AN ANALYTICAL MODEL

To evaluate the performance of AC functions, a queueing model for the response time of the IoT architecture is introduced. The variables necessary for the model are summarized in Table 1.

3.1 Application model and request distribution

Let us consider a single application allocated a fixed amount of resources from a multi-tenant Fog deployment. This application is described by its latency constraint Δ, its raw data size s_r, its processed data size s_p, and a service time X. In particular, s_r and s_p are assumed to be constant, while X is assumed to be a stochastic variable following an exponential distribution. Let now R be the total number of possible requests, and $\{r_1, \ldots, r_R\}$ these requests. Following previous work[1],

[1]Most query processes generated by actual demands from a set of distributed users have been found to follow a Zipf distribution [5, 22–25].

Table 1. Variables used in the model

Application specific		unit	Optimization variables		unit
Number of ≠ requests	R	-	Load-balancing function	$\phi(r)$	-
Cumulated arrival rate	λ	req/s	Cloud cache size	s_c	Objects
Service time	X	Cycles	Total request serving time	$T(\phi, s_c)$	s
Raw data size	s_r	B	Cost function	$\Pi(\phi, s_c)$	$/s
Processed data size	s_p	B	**Cloud characteristics**		
Latency constraint	Δ	s	Cloud compute capacity	p_c	Hz
Popularity distribution	$q(r)$	-	DB query delay	τ_d	s
Fog characteristics			Transit network capacity	κ_t	B/s
Fog compute capacity	p_f	Hz	Transit network delay	τ_t	s
Fog cache size	s_f	Objects	Handshake delay	τ_c	s
Access network capacity	κ_a	B/s	**Cloud pricing**		
Access network delay	τ_a	s	Compute price	c_c	$/s
Handshake delay	τ_f	s	Network price	c_n	$/s
Miscellaneous			Storage price	c_s	$/B
Cache hit probability	$h_f(r, s_f)$	-			

the request popularity distribution q is assumed to follow a Zipf distribution [26], i.e., for a request r arriving in the system, $q(k) = \mathbf{P}[r = r_k] = \gamma k^{-\alpha}$ where $\alpha > 0$ is the skew parameter and γ a normalization factor. In particular, requests arrivals are user-driven and are thus well modeled by a Poisson process of parameter λ. The arrival processes for each request r_k are assumed to be independent and thus follow a Poisson process of parameter $\lambda_k = q(k)\lambda$. The use of an independent requests model (IRM), wherein the arrival time of each request is independent from the arrival time of the previous one, is known to under-estimate the cache hit rate [27, 28], so we expect our results to be conservative.

3.2 Queueing model

Whenever relevant, we follow seminal work to select the most appropriate queueing model to describe each resource. Particularly, the *Fog compute* is modeled by a M/M/1-PS queue [12, 29][2]: the processor-sharing policy models that the Fog has a fixed amount of resources that must be shared between all the incoming requests. On the other hand, as the *Cloud compute* is elastic, it is best represented by an M/M/∞ queue[3], and we represent *Cloud-database* access as a constant-time M/D/∞ queue. For *network resources*, the M/M/1-PS model is used [12, 30, 31]. As *admission control* decisions and cache lookup should be done at line-rate (see Section 6), their impact is considered minimal w.r.t. other queues; and since they do not impact the comparison between Cloud and Fog service time anyway, they are neglected in what follows. Finally, the security handshakes are modeled as M/D/∞ queues[4], to account for a constant network and compute delay. Several families of protocols (TCP/TLS, ICN) requiring a different number of RTTs to complete the handshake can be modeled by modulating the amount of time spent in these queues. For the sake of generality, in the model, a one-RTT handshake is assumed (e.g., similarly to the TLS 1.3 handshake), i.e.: $\tau_f = 2\tau_a$ and $\tau_c = 2(\tau_a + \tau_t)$. The resulting queueing system is depicted in Figure 2. One can note that the

[2] Arrivals are Markovian, processing times are Markovian, one processor can handle the jobs according to a processor-sharing policy

[3] Arrivals are Markovian, processing times are Markovian, and there is an infinite number of instances to treat the queries. The underlying assumption is that of a perfectly elastic autoscaling policy.

[4] Arrivals are Markovian, processing times are constant and do not depend on the presence of other queries

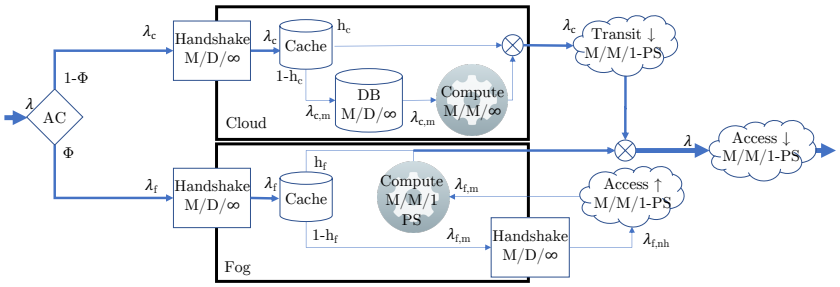


Fig. 2. Queueing network

Table 2. Arrival rate per queue in network

AC	AC module / Access downstream	λ
Fog	TLS Fog	$\lambda_f = \sum_{r \in R} \phi(r)\lambda q(r)$
	Access upstream / Fog Compute	$\lambda_{f,m} = \sum_{r \in R} \phi(r)(1 - h_f(r))\lambda q(r)$
Cloud	TLS Cloud / Transit downstream	$\lambda_c = \sum_{r \in R}(1 - \phi(r))\lambda q(r)$
	DB / Compute Cloud	$\lambda_{c,m} = \sum_{r \in R}(1 - \phi(r))(1 - h_c(r))\lambda q(r)$

request transmission time is uniquely taken into account in the handshake queue: since IoT requests have negligible size, their transmission time is indeed dominated by their propagation time.

The AC strategy can be expressed as ϕ:$\{1, \ldots, R\} \rightarrow [0, 1]$, a function that to each request associates a probability of being accepted in the Fog. In particular, given a popularity distribution q and an admission control strategy ϕ, the popularity distribution of requests arriving in the Fog is $q_f(r) = \gamma_f \phi(r) q(r)$, where γ_f is a normalization factor. For computing the hit probability in the Fog cache, the seminal approximation proposed by Che et al. [32] is used:

$$h_f(r) \approx 1 - e^{-q_f(r)t_s} \tag{2}$$

where t_s is the unique root of $\sum_{r=1}^{R}(1 - e^{-q_f(r)t}) = s_f$. A similar model applies for the Cloud cache, replacing the probability function ϕ by its complement $1 - \phi$.

3.3 Computing the statistical latency

First, let us point out that since processor-sharing queues are quasi-reversible processes, the exit distribution of an M/G/1-PS queue is a Poisson process (Theorem 3.6 of [33]). This is also true for the M/G/∞ queue [34], thus all the queues have a Markovian input. Thus, the expected service time for a job of size X and Poisson arrival rate λ of an M/G/1-PS queue with capacity C is given by:

$$\mathbf{E}[T] = \frac{1}{(\mu - \lambda)} \quad \text{where } \mu = \frac{C}{\mathbf{E}[X]} \tag{3}$$

The arrival rate at each queue can be derived from the offloading strategy ϕ and the cache hit probabilities h_f and h_c at the Fog and Cloud caches respectively, obtained using Equation (2). Per-queue arrival-rates are reported in Table 2. The expected queueing delay is thus:

$$\mathbf{E}[T] = \sum_r q(r)\Big[\phi(r)\mathbf{E}[T_f(r)] + (1 - \phi(r))\mathbf{E}[T_c(r)] + \mathbf{E}[T_{a,d}]\Big] \tag{4}$$

where the expected latency for the service time in the Fog $T_f(r)$ is:

$$\mathbf{E}[T_f] = \tau_f + (1 - h_f)\left(\tau_f + \mathbf{E}[T_{a,u}] + \mathbf{E}[T_{comp,f}]\right),$$

Table 3. An example application

Deployment		Application		Cloud pricing	
p_f	3 GHz	R	10^7	c_n	\$0.08 per GB
s_f	10^5 (1 GB)	λ	10 kHz	c_s	\$0.004446 per GB and hour
κ_a	10 Gbit/s	$\mathbf{E}[X]$	10^7 CPU cycles	c_c	\$0.033174 per vCPU and hour
τ_a	2 ms	s_r	1 MB		
p_c	2 GHz	s_p	10 kB		
τ_d	1 ms	α	1		
κ_t	1 Gbit/s	Δ	100 ms		
τ_t	20 ms				

the expected latency for the service time in the Cloud $T_c(r)$ is:

$$\mathbf{E}[T_c] = \tau_c + (1 - h_c)\left(\tau_d + \frac{\mathbf{E}[X]}{p_c}\right) + \mathbf{E}[T_{transit,d}],$$

and where $T_{a,u}, T_{a,d}, T_{comp,f}, T_{transit,d}$ respectively represent the time spent in the access upstream, in the access downstream, in the compute in the Fog, and in the transit downstream, and whose expected values can be computed with Equation (3), Table 1 and Table 2.

3.4 Computing the cost function

The cost per second consists of a network, a computing, and a storage term. The computing power rented in the Cloud is assumed to be synchronized with the incoming load (i.e., the Cloud spawns a container process at each new request). The cost of running the Cloud increases proportionally to the requested load: $p(c, s, \nu) = c_c c + c_s s + c_n \nu$, where c (resp. s) is the amount of computing (resp. storage) resources rented on the Cloud, and ν is the egress Cloud traffic in bytes per second.

Since the resource consumption in the Cloud is assumed to be elastic, if $Q_c(t)$ is the number of customers in the Cloud compute M/M/∞ queue, the instantaneous number of instantiated Cloud computing instances is: $c(\phi, s_c)_t = Q_c(t)$. According to [34], the expected value for $c(\phi, s_c)$ is thus:

$$\mathbf{E}[c(\phi, s_c)] = \frac{\lambda_{c,m}(\phi, s_c)}{p_c/\mathbf{E}[X]}$$

The storage cost is easier to compute as it depends on the cache size in the Cloud: $s(\phi, s_c) = s_c s_p$. Finally, for each incoming request, s_p is transferred downstream as a reply. Given the Cloud arrival rate λ_c, the average number of bytes per second on the Cloud downstream link is: $\mathbf{E}[\nu(\phi, s_c)] = \lambda_c(\phi) s_p$. Thus, the total cost function reads:

$$\Pi(\phi, s_c) = \frac{c_c \lambda_{c,m}(\phi, s_c)}{p_c/\mathbf{E}[X]} + c_s s_c s_p + c_n \lambda_c(\phi) s_p \tag{5}$$

3.5 An example application - Numerical parameters

All upcoming numerical evaluations use the characteristics described in Table 3. In particular, we select an application with a medium computing difficulty (10 ms on a 1 GHz processor) and medium processed data size, and a popularity parameter $\alpha = 1$[5]. For the Fog deployment, the application is assigned a fixed amount of resources from a computing platform, amounting to a 3 GHz CPU and 1 GB of cache. Finally, to make the evaluation more realistic, the public pricing of

[5]Zipf's α parameters are reported in [24, 25] to range in $[0.6, 2.5]$ across the literature. It is argued in [25] that $\alpha = 1$ is a reasonable value if no assumption is to be made as to where α lies in this spectrum.

the Google Compute infrastructure as of October 2017 is used (as per https://cloud.google.com/compute/pricing), the delay target is set to Δ=100 ms.

4 POPULARITY-BASED FOG ADMISSION

Request processing in the Fog-Cloud system can be decomposed in three modes: requests served (i) from the Fog cache, (ii) from the Fog compute, and (iii) from the Cloud (disregarding the Cloud cache vs the Cloud compute trade-off). An effective AC strategy should then maximize the proportion of traffic handled by the Fog. Since the Fog compute has a fixed capacity, increasing the amount of traffic handled by the Fog can only be done by increasing the cache hit-rate. The AC policy should then aim at maximizing the global Fog cache hit-rate (as defined by $\widetilde{h_f} = \sum_r \phi(r) h_r(r, \phi)$) while keeping the Fog-Compute arrival rate $\lambda_{f,m}$ bounded.

To this end, an effective approach is to admit only popular content in the Fog [15, 35]. Indeed, this method increases the hit-rate of the cache policy effectiveness by artificially reinforcing the skewness towards the most popular pieces of content in the cache input distribution. The AC must then be able to (i) identify the content targeted by each request and (ii) extract popularity patterns. Content identification (i) is an architectural problem, which we discuss in Section 6.1. To illustrate the limit of solutions without content awareness, a "*blind*" AC is introduced in Section 4.1. Extracting popularity patterns (ii) is a similar problem to caching policies: detecting the most popular pieces of content. Thus, using a virtual cache (caching only *identifiers* rather than content) that follows traditional admission and eviction policies is a natural solution. In this case, a hit in the virtual cache identifies popular requests which should be handled in the Fog, while a miss hints that the request is not popular and should be offloaded to the Cloud. This solution is similar to multi-layered caching systems such as the 2Q-LRU [36], differing mainly in that the first layer (the identifier cache) is completely independent from the actual cache. Indeed, in 2Q the virtual layer only governs cache insertion (but not cache retrieval), whereas the Fog-AC might refuse requests even though the corresponding answer is available in the Fog cache. In particular, the LFU (Section 4.2) and LRU (Section 4.3) policies are investigated.

4.1 Blind AC

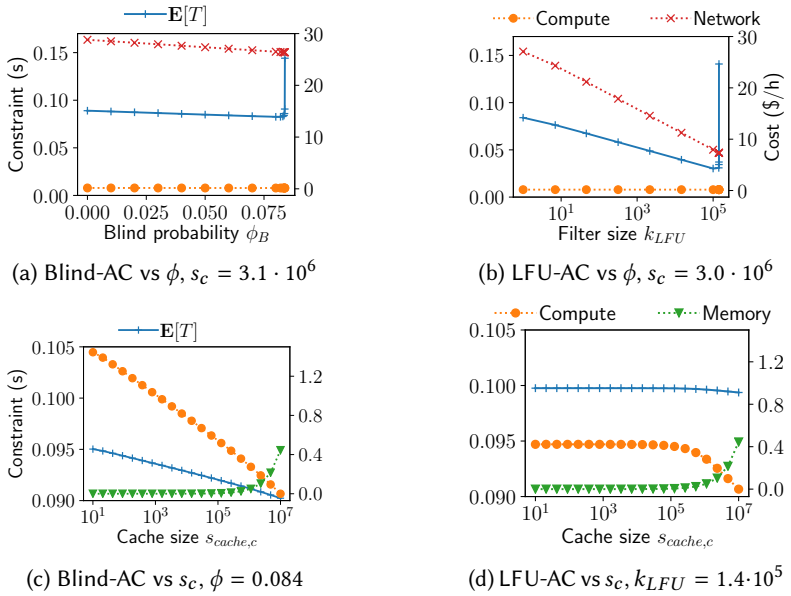
A request-oblivious AC strategy, called *Blind-AC*, is used as a baseline. It admits all requests with i.i.d. probability: $\phi(r) = \phi_B, \forall r$. In particular, Equation (1) can be rewritten as:

$$\begin{cases} \min. & \Pi(\vec{\phi}_B, s_c) \\ \text{s.t.} & \mathbf{E}[T(\vec{\phi}_B, s_c)] \leq \Delta \end{cases}$$

with $\vec{\phi}_B = (\phi_B, \ldots, \phi_B)$. In this particular case, since $(\phi_B, s_c) \in [0, 1] \times [0, R]$, the problem is easy to optimally solve numerically. We next evaluate the respective importance of the two optimization variables using the parameters reported in Table 3 for numerical evaluation. In particular, the variation of the costs and constraint functions depending on either ϕ_B or s_c are represented in respectively Figure 3a and Figure 3c.

Figure 3a shows the variation of the constraint function (+, left side) and of the compute (•, right side) and network (×, right side) costs for a fixed cache size $s_c = 3.1 \cdot 10^5$. The storage cost is ignored as it is constant (since s_c is fixed). The first takeaway is that, as expected from the costs in Table 3, the network cost is dominant w.r.t. the compute and memory cost. We also notice that the constraint function diverges towards $+\infty$ when ϕ_B grows close to 0.45, as the Fog compute queue becomes unstable and cannot handle the request rate.

In Figure 3c, we next vary the cache size s_c for a fixed load-balancing probability $\phi_B = 0.42$ (the sweet spot in Figure 3a). We show the value of the constraint function (+, left side), and the



(a) Blind-AC vs ϕ, $s_c = 3.1 \cdot 10^6$

(b) LFU-AC vs ϕ, $s_c = 3.0 \cdot 10^6$

(c) Blind-AC vs s_c, $\phi = 0.084$

(d) LFU-AC vs s_c, $k_{LFU} = 1.4 \cdot 10^5$

Fig. 3. Constraint value (left) and cost (right) of the system vs ϕ and s_c for the Blind- and LFU-AC

compute (●, right side) and memory (▼, right side) costs. The network cost is now ignored since it is constant when ϕ is constant. First, we note that varying the cache size has a limited impact on both the cost and the constraint function. Furthermore, in this case, given that the compute is almost one order of magnitude more expensive than the memory and the Cloud popularity distribution is sufficiently skewed, it is interesting to cache highly popular requests.

4.2 LFU-AC strategy

Let us first consider a perfect LFU virtual cache, which deterministically identifies the k_{LFU} most popular requests for processing in the Fog. In particular, the AC function ϕ can be expressed as:

$$\phi(r) = \delta_{r \leq k_{LFU}} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } r \leq k_{LFU} \\ 0 & \text{otherwise.} \end{cases}$$

Equation (1) then becomes:

$$\begin{cases} \min.\ \Pi(\vec{\delta}_{k_{LFU}}, s_c) \\ \text{s.t. } \mathbf{E}[T(\vec{\delta}_{k_{LFU}}, s_c)] \leq \Delta \end{cases}$$

with $\vec{\delta}_{k_{LFU}} = (\delta_{1 \leq k_{LFU}}, \ldots, \delta_{R \leq k_{LFU}})$. Again, this problem is two-dimensional and can easily be solved numerically.

Figure 3b (resp. Figure 3d) shows the evolution of the cost and constraint functions while setting $s_c = 0$ (resp. $k_{LFU} = 6.1 \cdot 10^5$) for the setup in Table 3. At a first glance, Figure 3b indicates that a proper choice of k_{LFU} allows decreasing the network cost at levels unreachable with the Blind-AC while respecting the constraint. Furthermore, as in Section 4.1, the dominant factor in terms of cost is the number of offloaded requests. Both of these insights point towards LFU as a good strategy for Fog/Cloud AC. Additionally, Figure 3d shows that for small values of s_c, the compute cost stays constant. This is due to the popularity distribution at the Cloud cache, which only contains the

Table 4. Optimal costs and parameters per AC

Method	$\mathbf{E}[\phi]$	s_c	Π
Blind	0.084	$3.1 \cdot 10^6$	27 \$/h
LFU	0.75 ($k_{LFU} = 1.4 \cdot 10^5$)	$3.0 \cdot 10^6$	7.6 \$/h
LRU	0.71 ($k_{LRU} = 2.8 \cdot 10^5$)	$2.9 \cdot 10^6$	8.6 \$/h

long tail of the Zipf distribution. Thus, for small cache sizes, the hit probability is low and the cache almost useless.

4.3 The LRU-AC strategy

While the LFU-AC is efficient, it is an ideal policy, difficult to realize in practice if the popularity distribution is not known in advance (as deriving the exact popularity distribution is difficult and slow[6]). To derive a practical AC policy, we argue that the AC does not need to learn the popularity of each specific request. It only needs to flag whether a request is popular, acting as a low-pass filter. In a second step, the LRU policy is thus considered for the AC virtual-cache. Compared to the aforementioned counter solution for the LFU-AC, the LRU-AC has four main advantages: (i) it does not require prior knowledge of the application; (ii) it keeps memory constrained to the size of the filter, instead of the size of the catalogue; (iii) it is flexible w.r.t. changes in the popularity distribution; (iv) it requires minimal effort for integration in ICN forwarders as the LRU structure is already used for caches.

To incorporate LRU-AC in the model, we must compute the load-balancing function ϕ depending on the filter size k_{LRU}. Since the AC behaves like an LRU cache, $\phi(r) = h_{k_{LRU}}(r)$ where $h_{k_{LRU}}(r)$ is the hit probability for the request r in an LRU cache of size k_{LRU} with input distribution q, which can be derived straightforwardly from Equation (2). Integrating this in Equation (1) yields a constraint and a cost function that depends only on k_{LRU} and s_c:

$$\begin{cases} \text{min. } \Pi(\vec{h}_{k_{LRU}}, s_c) \\ \text{s.t. } \mathbf{E}[T(\vec{h}_{k_{LRU}}, s_c)] \leq \Delta \end{cases}$$

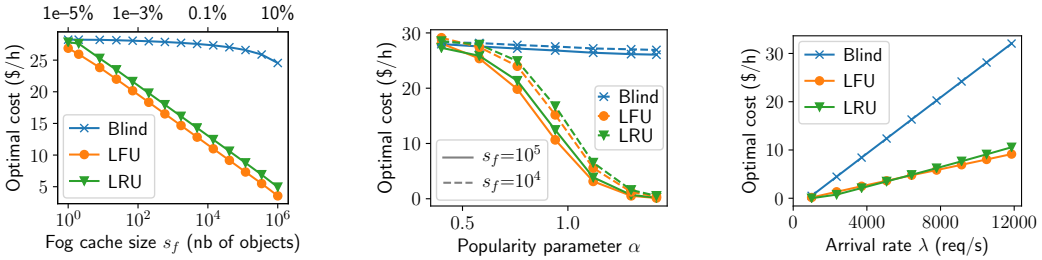
with $\vec{h}_{k_{LRU}} = (h_{k_{LRU}}(1), \ldots, h_{k_{LRU}}(R))$. However, the interaction between the LRU meta-cache and Fog LRU cache introduces some correlation effects, as shown by Garetto et al., for the LRU-2Q cache [37]. Following their example, a discrete time Markov chain is used to model the interdependence between hits in the filter and in the Fog cache. Details of the derivation are provided in [38]. Compared to the LFU-AC, optimizing the LRU-AC only requires knowing the popularity skewing factor α and the arrival rate λ instead of the actual per-content popularity distribution.

4.4 Preliminary evaluation of the AC strategies

In this section, a first evaluation of the AC strategies based on the model introduced in Section 3 provided. In particular, the Method-of-moving-asymptotes [39] (in its NLopt implementation [40]) is used to solve the optimization problem (Equation (1)). A Jupyter notebook is available for reproducing the results of this section or to experiment with different parameters [41].

In a first step, we show in Table 4 the optimized values for our example application. We first note that using the LRU and LFU-AC allows the Fog to handle more than twice as many requests as with the Blind-AC. This results in a decrease in offload cost of more than 70%. Furthermore, it

[6]This requires either offline analysis of the popularity distribution, or to keep counters of incoming requests. Both solutions are not flexible to popularity changes and are difficult to implement efficiently.



(a) Fog cache size s_f (b) Zipf parameter α (c) Arrival rate λ

Fig. 4. Cost of the Blind-, LFU-, and LRU-AC letting the application-independent parameters vary

shows that the LRU-AC has similar performances to the LFU-AC, with a 3% relative difference in offload cost w.r.t. the Blind-AC.

In Figure 4a, the influence of the Fog cache size s_f on the optimal Cloud cost for the Blind-, LFU-, and LRU-AC strategies is shown. In particular, this figure shows that both the LFU- and LRU-AC strategies are much better at exploiting an increasing cache size that the Blind-AC. Indeed, for the Blind-AC, increasing the Fog cache size only slightly decreases the optimal cost. On the other hand, both the LFU- and the LRU-AC provide an exponential decrease of the cost as the cache size increases, showing the effectiveness of popularity-based AC. Furthermore, this graph confirms that the LRU-AC is a good approximation of the ideal LFU-AC, this regardless of the Fog cache size.

Figure 4b then investigates the influence of the popularity distribution skew (represented by the Zipf parameter α). For small α values, the popularity distribution converges towards a uniform distribution, thus diminishing the impact of popularity-based ACs. However, for typical values of α found in the literature ($\alpha \in [0.5, 1.1]$), the LFU- and LRU-AC strategies allow for a largely reduced optimal cost for both Fog cache sizes that we tested. Furthermore, the LFU- and LRU-AC strategies with $s_f = 10^4$ are much more efficient than the Blind-AC strategy with $s_f = 10^5$. This indicates that the strategies also allow for more efficient provisioning of Fog resources. Once again, the performance of the LRU-AC is close to the LFU-AC, varying by at most 6%.

Finally, the impact of the arrival rate on the efficiency of the strategies is shown in Figure 4c. Interestingly enough, the optimal cost increases linearly w.r.t. the arrival rate for all three cases, with slopes at $2.9{\cdot}10^{-3}$ \$/Hz for the Blind-AC, $8.2{\cdot}10^{-4}$ \$/Hz for the LFU-AC, and $1.0{\cdot}10^{-3}$ \$/Hz for the LRU-AC. This confirms that the LFU- and LRU-AC strategies cope better with increased loads (e.g., flash crowds) than the Blind-AC. This is typically due to the improved hit rate at the Fog cache, which absorbs a large part of the increased arrival rate. Particularly, if the cost of the Blind-AC diverges with respect to the LFU-AC and the LRU-AC for an increasing arrival rate, their ratio stays however constant. The ratio between the absolute costs for the LFU-AC (LRU-AC) over the Blind-AC is of 3.5× (2.9×). Thus, when the arrival request rate increases, the relative gain of using the LFU-AC (LRU-AC) over the Blind-AC also increases, which shows the LRU- and LRU-AC to be quite robust to high arrival rates.

This preliminary evaluation shows that the LRU-AC combines tractability, flexibility, and effectiveness. It is thus selected as the default admission policy for the remainder of this paper.

5 AGEING BLOOM FILTERS FOR A HARDWARE-ACCELERATED LRU-AC

As the AC strategies were designed to provide predictable completion time under high-load, their implementation should offer the same virtues. Thus, implementing the LRU-AC in hardware seems

Algorithm 1 The ABF insertion algorithm

```
if item ∉ A_1 then
    A_1.INSERT(item)
    cnt ← cnt + 1
    if cnt = n_a then
        A_2.FLUSH(); SWAP(A_1,A_2); cnt ← 0
    end if
end if
```

particularly profitable. However, the LRU-AC raises many realization challenges, particularly on hardware-accelerated platforms. The first and obvious one is memory usage. As shown in Section 5.4, the total memory footprint of the LRU-AC can overtake the high-speed BRAM (Block RAM [42]) available in the hardware – notwithstanding the BRAM that needs to be allocated for, e.g., packet queues. Furthermore, the access time to an LRU element is not constant (as a hash map provides only amortized constant time for read operations), which renders it undesirable for hardware implementation [16]. Thus, in this section, we explore Ageing Bloom Filters (ABF) as an alternative data-structure to implement the LRU-AC. After a brief description of the ABF behavior (Section 5.1), a predictive model similar to Che's approximation (Equation (2)) for the hit-rate of the ABF is presented (Section 5.2). The model is verified by way of simulation (Section 5.3) and memory gains compared to standard LRU implementations are evaluated (Section 5.4).

5.1 Ageing Bloom filters

A popular and natural structure for storing sets in programmable hardware are Bloom filters [43–45]. They provide a compact and efficient way to store set membership with a controllable error probability as the only trade-off. In particular, Bloom filters have been proposed to store network identifier for high-performance packet forwarding [45, 46]. Thus, using a Bloom filter to store the content of the LRU-AC seems like a promising step towards hardware implementation. The LRU-AC, however, requires the ability to evict old content from the cache, which standard Bloom filters do not support. To replicate this behavior, Yoon introduced Ageing Bloom Filters (ABF) [17].

An ABF consists of two parallel Bloom filters: the *active* Bloom filter A_1, used to learn the most recent items, and the *passive* Bloom filter A_2, which holds older items in memory. It has two parameters: n_a, the maximum number of items that each filter holds, and f_p, a target false positive rate. The functioning of the ABF is summarized in Algorithm 1. In steady state, the passive filter A_2 holds exactly n_a different items. The active filter A_1 holds $0 \leq cnt < n_a$ items, some redundantly with A_2. An item is said to be in the ABF if it belongs to either A_1 or A_2. Insertion is done only in the active filter A_1 until it holds n_a unique items. At this point, A_2 is swapped with A_1, and the active filter is reinitialized. This ensures that the ABF contains at least the last n_a different items received, and at most $2n_a$. However, there are redundancies between A_1 and A_2, the exact number of objects in the ABF is a stochastic process.

5.2 Hit-rate approximation for the ABF

To replace the LRU-AC with an ABF, the expected hit-rate of the ABF must be evaluated. Indeed, as shown in Section 4.4, the hit-rate distribution is the major factor that explains the performance of the LRU-AC. While for brevity reasons, only the major ideas behind the model derivation are presented here, a complete explanation is provided in [38].

As in Section 3, Zipf arrivals under IRM are assumed. Let us consider $|A_1 \cup A_2|(k)$ the stochastic process representing the number of unique elements in an ABF after k arrivals. First, as depicted in

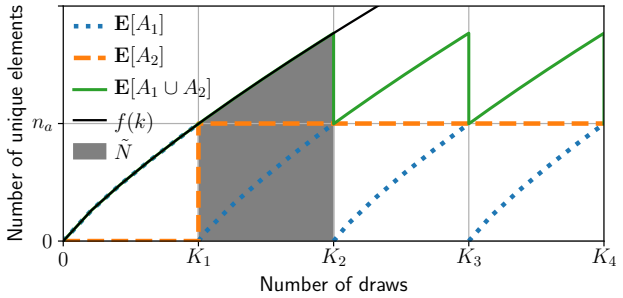


Fig. 5. Number of distinct elements in the active filter A_1 (··), in the passive filter A_2 (- -) and in the full filter $A_1 \cup A_2$ (–), as well as the function $f(k)$ (–) used to compute them and the stochastic average $\tilde{N}$ of $|A_1 \cup A_2|$.

Figure 5, one can observe that the behavior of an ABF is cyclic (except for the bootstrap, during which A_2 is empty). At a given time, its behavior only depends on the arrivals which occurred after the second-to-last flush-and-swap operation, as the memory of events that occurred before that point has been flushed. Therefore (and since arrivals are memoryless due to the IRM assumption), it suffices to analyze the behavior of the filter during the second cycle. Let us denote by K_1 (respectively K_2) the number of steps necessary to complete the first (respectively second) cycle:

$$K_1 = \min\{k \geq 0 : |A_1|(k) = n_a\}$$
$$K_2 = \min\{k \geq K_1 : |A_1|(k) = n_a\}$$

We will thus study the behavior of the filter in $[K_1, K_2)$.

Approximating K_1 and K_2. For $k \leq K_2$, the probability $h_k(r)$ that content r is in the filter after $k \leq K_2$ draws is simply the probability that it was selected at least once from the k draws:

$$h_k(r) \stackrel{\text{def}}{=} \mathbf{P}[r \in (A_1 \cup A_2)(k) | K_2 \geq k] = 1 - (1 - q(r))^k$$

Thus, the expected number $f(k)$ of items in the filter after k draws is:

$$f(k) \stackrel{\text{def}}{=} \mathbf{E}[|A_1 \cup A_2|(k) \ | K_2 \geq k] = \mathbf{E}[\sum_{r=1}^{R} \mathbf{1}_{r \in (A_1 \cup A_2)(k)} | K_2 \geq k] = \sum_{r=1}^{R} h_k(r) = \sum_{r=1}^{R} [1 - (1 - q(r))^k]$$

It is possible to prove, using Chernoff bounds (see [38]) that K_1 (the number of steps necessary for there to be n_a items in A_1) deviates little from $\widehat{k_1}$, the number of steps necessary for there to be n_a items *in average* in A_1, defined as:

$$\widehat{k_1} \stackrel{\text{def}}{=} f^{-1}(n_a) \tag{6}$$

Due to the cyclic behavior of the filter, K_2 is then naturally approximated as $K_2 \approx 2\widehat{k_1}$.

It is then possible to efficiently find $\widehat{k_1}$ by using an approximation for $f(k)$, rather than inverting a sum with R elements. Assuming a Zipf catalogue with a parameter $\alpha > 1/2$ (*i.e.*, $q(r) = \frac{r^{-\alpha}}{H_{R,\alpha}}$ with $H_{R,\alpha} = \sum_{i=1}^{R} i^{-\alpha}$), we have:

$$f(k) \approx \begin{cases} Ak - Bk \log k & \text{if } \alpha = 1 \\ Ak - Bk^{1/\alpha} & \text{if } \alpha \in (1/2, 1) \cup (1, +\infty) \end{cases} \tag{7}$$

where:

$$\begin{cases} A = \dfrac{R^{1-\alpha}}{(1-\alpha)H_{R,\alpha}} \text{ if } \alpha > \dfrac{1}{2}, \alpha \neq 1; \quad 1 + \dfrac{1 + \log\log R - 2\gamma}{\log R + \gamma} \text{ if } \alpha = 1 \\ B = \dfrac{\Gamma(-1/\alpha)}{\alpha H_{R,\alpha}^{1/\alpha}} \quad \text{if } \alpha > \dfrac{1}{2}, \alpha \neq 1; \quad \dfrac{1}{\log R + \gamma} \qquad\qquad \text{if } \alpha = 1 \end{cases} \tag{8}$$

where $\Gamma(z)$ is the gamma function and $\gamma \approx 0.577$ the Euler-Mascheroni constant. The derivation is provided in [38].

Approximating the hit rate. As introduced in the previous section, the hit rate $h_k(r)$ for content r after $k \leq K_2$ draws from the catalogue is: $h_k(r) = 1 - (1 - q(r))^k$. Thus, the (stochastic) hit rate $H(r)$ for content r when observed at an instant K_{obs} drawn uniformly during the cycle $[K_1, K_2)$ is:

$$H(r) \overset{\text{def}}{=} \mathbf{E}[\mathbf{1}_{r \in (A_1 \cup A_2)(K_{obs})} | K_1, K_2] = \frac{1}{K_2 - K_1} \sum_{K_1 \leq k < K_2} \left[1 - (1 - q(r))^k\right]$$

and the corresponding average hit rate $h(r)$ is:

$$h(r) \overset{\text{def}}{=} \mathbf{E}[H(r)] = \mathbf{E}\left[\frac{1}{K_2 - K_1} \sum_{K_1 \leq k < K_2} \left[1 - (1 - q(r))^k\right]\right]$$

Since (as argued in the previous section) K_1 and K_2 can be approximated by $\widehat{k_1}$ and $2\widehat{k_1}$, respectively, we can further provide an approximation $h_a(r)$ of $h(r)$ as:

$$h(r) \approx \frac{1}{\widehat{k_1}} \sum_{\widehat{k_1} \leq k < 2\widehat{k_1}} \Big[\underbrace{1 - (1 - q(r))^k}_{\approx 1 - e^{-kq(r)}}\Big] \approx \frac{1}{\widehat{k_1}} \int_{\widehat{k_1}}^{2\widehat{k_1}} \left[1 - e^{-kq(r)}\right] \mathrm{d}k \overset{\text{def}}{=} h_a(r)$$

Using $t_C(r) = \widehat{k_1} + \frac{1}{q(r)} \log \frac{\widehat{k_1} q(r)}{1 - e^{-\widehat{k_1} q(r)}} \in [\widehat{k_1}, \frac{3}{2}\widehat{k_1}]$, simple algebra then yields:

$$h_a(r) = 1 - e^{-q(r) t_C(r)} \tag{9}$$

A Che-like approximation. For large values of r, $t_C(r)$ exhibits very little variation: a Taylor expansion shows that $t_C(r) \approx \frac{3}{2}\widehat{k_1} - \frac{q(r)}{24}\widehat{k_1}^2$. For small values of r, $t_C(r)$ varies more, but its contribution to $h(r)$ can be neglected as $h(r) \approx 1$ in those cases, as argued in [24] (figures depicting this phenomenon are available in [38]). Therefore, it is possible to make the approximation that $t_C(r)$ is a constant t_C, yielding:

$$h_a(r) \approx 1 - e^{-q(r) t_C} \tag{10}$$

In that case, by summing over $r \in \{1, \ldots, R\}$, t_C can be computed by finding the root of:

$$\sum_{r=1}^{R} [1 - e^{-q(r) t_C}] = \sum_{r=1}^{R} h_a(r) \overset{\text{def}}{=} \tilde{N} \tag{11}$$

where $\tilde{N}$ represents the average number of items in the filter, and can be computed with: $\tilde{N} \approx \frac{1}{\widehat{k_1}} \int_{\widehat{k_1}}^{2\widehat{k_1}} f(k) \mathrm{d}k$ using the approximation of $f(k)$ provided in the previous section and straightforward integration. This is similar to the Che approximation (see Equation (2)), with the (fixed) size of the LRU cache replaced by the average over time of the "size" (number of distinct items) of the ABF. Functions for computing $\widehat{k_1}$ and n_a depending on k_{LRU} are provided in the Jupyter Notebook associated with this paper [41]. For instance, k_{LRU}=2.8·10^5 (as in Table 4) yields n_a=2.0·10^5.

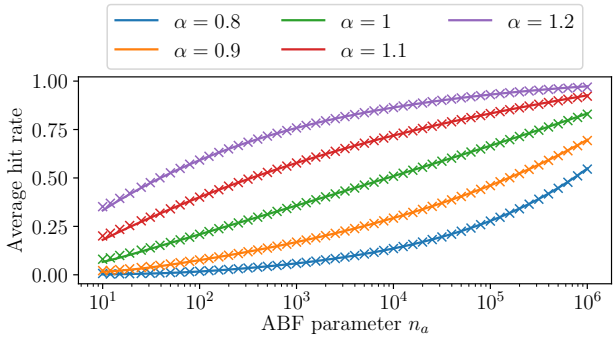


Fig. 6. Simulation results (×) and model results (plain lines) for the expected hit-rate of the ABF, as a function of n_a (R=10^7 and $\alpha \in \{0.8, 0.9, 1, 1.1, 1.2\}$). Each simulation point consists of the average value on 10^8 arrivals.

5.3 Model verification

To verify the model, a software implementation of the ABF was subjected to a Poisson arrival process with Zipf popularity distribution, for $\alpha \in \{0.8, .9, 1, 1.1, 1.2\}$. In particular, the accuracy of Equation (10) is shown in Figure 6, which compares the average hit-rate $\sum_r q(r)h_a(r)$ computed from the approximation of Equation (10) with the measured average hit-rate in simulation runs. It shows that the model fits almost perfectly with the experiments, with a relative difference $< 0.5\%$ for $n_a \geq 1000$.

5.4 ABF - memory usage vs LRU

LRU memory. A simple LRU implementation consists of a hash map pointing to a doubly-linked list. Each LRU entry has a memory footprint at least equal to the size of 3 pointers (one in the hash map and two in the list). The minimal pointer size is $\lceil \log_2 k_{LRU} \rceil$. To compare the ABF to the LRU and according to Equation (11), $k_{LRU} = \tilde{N}$ is used. Thus, the memory used by the LRU is:

$$m_{LRU} = 3\tilde{N}\lceil \log_2 \tilde{N} \rceil \tag{12}$$

ABF memory. Let f_p be the wished false-positive rate of the ABF. The corresponding false-positive rate of A_1 and A_2 is $f_a = 1 - \sqrt{1 - f_p}$ [17]. The corresponding number of hash-functions is $n_h = \lceil -\log_2(1 - \sqrt{1 - f_p}) \rceil$, and the corresponding memory consumption of the ABF is $m_{ABF} = \frac{2n_a n_h}{\log 2}$, yielding:

$$m_{ABF} = \frac{2n_a \lceil -\log_2(1 - \sqrt{1 - f_p}) \rceil}{\log 2} \tag{13}$$

Numerical results. For $k_{LRU} = 2.8 \cdot 10^5$ (as in Table 4), the minimal pointer size is 19 bits (since $\log_2(k_{LRU}) \approx 18.1$). The required memory per element adds up to 57 bits. In total, this amounts to $m_{LRU} = 16$ Mbit. By comparison, the corresponding ABF has for parameter $n_a = 2.0 \cdot 10^5$. For a false positive rate of $f_p = 1\%$, Equation (13) yields $m_{ABF} = 4.6$ Mbit, dividing the LRU-AC memory footprint by 4. Furthermore, let us note that while the computation of m_{ABF} is exact, the computation of m_{LRU} is conservative. Indeed, implementing a hash-table requires a significant memory overhead and creates a memory vs operation-latency trade-off.

6 HARDWARE-IMPLEMENTATION OF THE LRU-AC

6.1 Using hICN as the underlying network layer

To guarantee line-rate performances, the hardware-accelerated AC must be able to access the request identifier easily. While extracting application-level semantics from packets (deep-packet inspection) is possible in hardware using frameworks such as P4 [20], it is costly in terms of cycles. Therefore, the performance of the AC hardware module would profit from a network framework that exposes application semantics at a low-level, but that also provide a deterministic packet format (through fixed-size headers with fixed field locations).

In that regard, hICN [19] is an ideal candidate. hICN is an implementation of ICN [47] that transparently uses the TCP/IPv6 packet format for backward-compatibility. In particular, while hICN is a name-based protocol (i.e., forwarding is performed based on named-content identifiers rather than locators), names are encoded using an IPv6 address for a *name prefix* (64 bits of routable prefix and 64 bits of data identifier) and a 32-bit integer for the *name suffix*. An Interest (resp. Data) packet then carries the name prefix in the IPv6 destination (resp. source) address field, while the name suffix is placed in the TCP segment number field. As such, hICN holds the desirable characteristics for implementing the LRU-AC in hardware: request semantics accessible in *fixed-size* fields in *fixed locations* of the *network and transport headers*.

6.2 Hardware-implementation of the LRU-AC

To demonstrate implementability of the LRU-AC in hardware, the NetFPGA-SUME [18], a state-of-the-art academic programmable network cards, was used. To provide a modular and easily-modifiable implementation, the prototype is implemented in P4 [20], allowing packet parsing to be performed in a high-level language. The P4→NetFPGA framework [48] is then used to translate that high-level representation into a NetFPGA-SUME implementation.

The prototype comprises two parts: (i) a Bloom filter *atom*[7] written in Verilog, that implements a single Bloom filter; and (ii) a P4 data-plane, which performs packet parsing, processing, and deparsing, and whose processing part implements an ABF by using two Bloom filters atoms. Using a Bloom filter atom and implementing the ABF logic in P4 rather than directly implementing the ABF as a black-box Verilog module provides greater modularity and expressiveness in the high-level language and simplifies the engineering effort by having to focus on optimizing only a single and simpler low-level module. The Bloom filter atom has been upstreamed to the P4-NetFPGA project.

Bloom filter atom. The Bloom filter atom takes a fixed-length key of size s_{key} in argument, as well as an operand specifying the operation to be performed on that key (read or insert), and returns a single bit specifying whether the key was found in the filter – in case of an insert operation, whether the key was found before insertion. Additionally, a third operand allows resetting the filter (*i.e.*, clearing all its bits). This allows exporting a very simple API:

```
void bloom_filter(in bit<2> opcode, in bit<KEY_SIZE> index, out bit<1> result);
```

The Bloom filter is parameterized by the number n_h of hash functions, and the size s_{hash} of their output – governing the number of addressable objects. Each hash function h^i ($i \in \{1, \ldots, n_h\}$) is implemented using universal hashing [50]. Indeed, universal hashing relies only on multiplication, XOR and shift operations, and can thus be efficiently implemented on the NetFPGA-SUME (using multiplier blocks), with a latency of one cycle.

To optimize throughput and latency, the Bloom filter is implemented with a pipelined approach, as depicted in Figure 7. The idea is to use each cycle to query one bit (at the address dictated by

[7] Atoms are low-level modules that can handle state and perform a simple operation, while interfacing with a higher-level packet processing language [49].

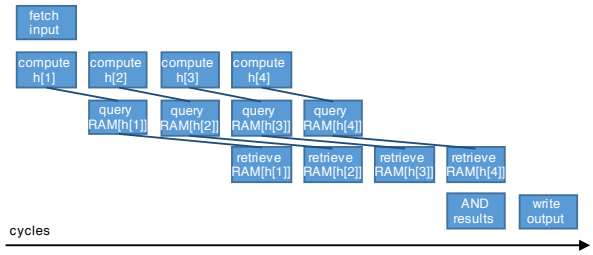


Fig. 7. Bloom filter pipeline illustration with $n_h = 4$: latency is 7 cycles, throughput is one operation per 8 cycles.

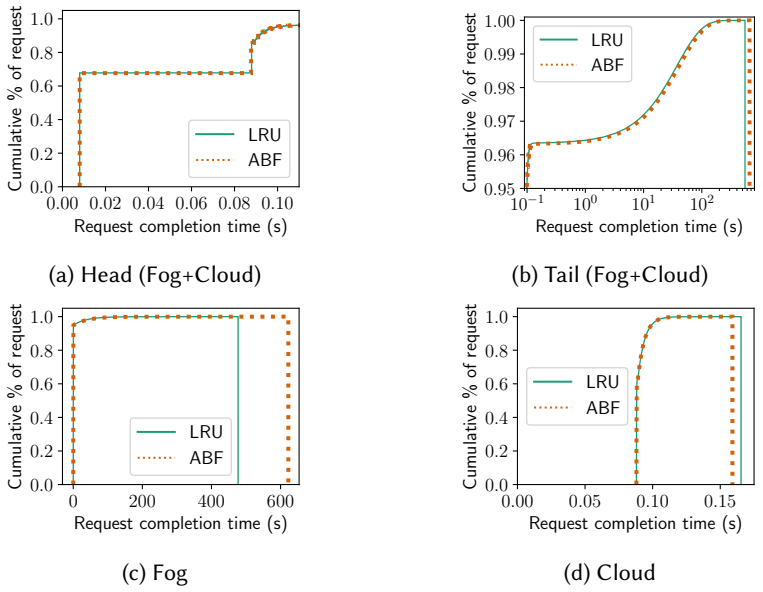
Table 5. Resource usage of a Bloom filter atom

s_{hash}	s_{key}	Logic (LUTs)	Registers	BRAM	Multipliers
20	24	185	134	33	2
21	24	246	152	65	2
22	24	357	186	129	2
20	48	281	178	33	4
21	48	317	196	65	4
22	48	431	230	129	4

one of the n_h hash functions), for a total of approximately n_h cycles. Since querying the BRAM has a 2-cycle latency and hash function computation uses a full cycle, the i-th hash function is computed at cycle i and the corresponding bit is queried at cycle $i + 1$ for a result retrieved at cycle $i + 3$. The final result is then output at cycle $n_h + 4$, after ANDing all the intermediary results. In sum, the Bloom filter has a latency of $n_h + 3$ cycles and a throughput of 1 operation every $n_h + 4$ cycles. In terms of spatial complexity, in addition to the LUTs (Look-Up Tables, the fundamental reconfigurable logic blocks in FPGAs) required to implement the logic, the module uses $2^{s_{hash}-15}$ blocks of BRAM (of size 32 Kbit). Table 5 reports the resource usage of a single Bloom filter after synthesis, for different values of s_{key} and s_{hash}.

P4 data plane. The P4 data plane leverages the simplicity of having defined an external Bloom filter atom to provide a simple implementation. It comprises four components: (i) a parser, which extracts Ethernet and IPv6 headers and stores the hICN object key, (ii) an action, which implements the ABF logic to determine the egress interface for the packet, (iii) a match-action table, which maps that interface to an Ethernet address, and (iv) a deparser, which reconstructs the output packet. (i), (iii) and (iv) are straightforward to implement in P4. (ii) is implemented through four external atoms: two Bloom filters, a flag that keeps track of the active Bloom filter, and a counter that keeps track of the number of requests since the last swapping event. Predicated-read-add-write registers available from the P4-NetFPGA framework [49] are used to implement the counter and the flag. The counter is incremented if its value is smaller than $\widehat{k_1}$ and reset when it reaches $\widehat{k_1}$[8]. The flag is swapped when the counter has just been reset. Depending on the value of the flag and on whether it has just been swapped, suitable (read, write or clear) operations are sent to the two Bloom filters,

[8] $\widehat{k_1}$ is used as a threshold on the number of steps rather than n_a on the number of elements in the filter because the `reg_ifElseRaw` atom can only be accessed once per packet in the P4-NetFPGA workflow. To use n_a, the counter would have to be read before querying the filters (to decide whether to swap) and updated after the queries have completed (to count the number of active elements). The validity of using $\widehat{k_1}$ comes from Equation (6).



(a) Head (Fog+Cloud)

(b) Tail (Fog+Cloud)

(c) Fog

(d) Cloud

Fig. 8. Cumulative distribution functions of the measured request response time for the LRU- and ABF-AC

along with the key extracted from the packet. Finally, the output port is chosen, depending on the OR of the result of the two queries.

7 EVALUATION

In this section, an evaluation of the FPGA-based LRU-AC (named ABF-AC in this section to distinguish it from the LRU-AC implemented with an actual LRU filter) is presented. First, packet-level simulation is used to compare the ABF-AC to the LRU-, LFU- and Blind-AC. Then, the throughput and processing speed of the FPGA implementation is evaluated.

7.1 Packet-level simulation

To provide insights about the fine-grained behavior of the schemes introduced in Section 4, packet-level simulations of the queueing network introduced in Section 3 using the different AC strategies. The simulator, written in Rust and available in open-source [41], is a general-use queueing simulator designed to allow quick specification and testing of queueing networks. In this section, it is set up with the values presented in Table 3 and the AC modules with the values computed in Table 4. The ABF filter is configured with $\widehat{k_1} = 5.2{\cdot}10^5$ (computed through Equation (6)) and $f_p = 1\%$. The interested reader can use the Jupyter notebook provided with the simulator to find the filter parameters tailored to their own scenario and set up the simulator accordingly.

In a first step, the per-packet response time of the LRU-AC is considered. Figure 8 shows the cumulated distribution function (CDF) for both the ABF- and LRU-AC. For the sake of clarity, the head and the tail of the distribution are represented in resp. Figure 8a and Figure 8b, while individual CDFs for the Fog and Cloud paths are represented in Figure 8c and Figure 8d. The only visible difference between the LRU- and the ABF-based implementations is the spread of the distribution, which concerns only 0.1% of requests, thus justifying the validity of the ABF-AC. Of further note is the length of the queue, which goes up to 600s. As shown by Figure 8, this is due to the effect of the tri-modal nature of the distribution on the problem formulated in Equation (1). Indeed, a large

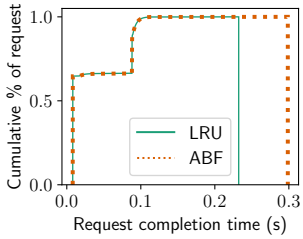


Fig. 9. Cumulative distribution function of the measured response time for conservative settings of the LRU- and ABF-AC

Table 6. Request breakdown and threshold excess rate for the Blind-, LFU-, LRU-, and ABF-AC

	Request breakdown			Threshold excess rate
	Fog cache	Fog compute	Cloud	
Blind	5.3%	3.0%	92%	4.0%
LFU	71%	3.6%	25%	3.2%
LRU	68%	3.8%	29%	4.9%
ABF	67%	3.9%	29%	4.9%

percentage of requests are processed with a latency $\ll \Delta$. Thus, the constraint $\mathbf{E}[T] \leq \Delta$ can tolerate a long tail, which might be a problem for real-life applications (even if 99% of the requests are completed under a minute). For operators with stricter latency constraints, the length of the queue can be reduced by slightly modifying the value of k_{LRU} and $\widehat{k_1}$. For instance, artificially reducing the load on the Fog by 5 percentage points by setting k_{LRU}=1.3·10^5 (resp. $\widehat{k_1}$=2.3·10^5) allows reducing the distribution spread to about 0.3 s and the threshold excess rate (i.e., % of requests with service time $\geq \Delta$) to 1% for a 16% cost increase for both implementations, as depicted in Figure 9.

In a second step, Table 6 shows the request path repartition and the threshold excess rate for the Blind-, LFU-, and both implementations of the LRU-AC. It highlights that the threshold excess rate is of the same magnitude between all schemes, with probabilistic LRU-AC just slightly higher. Furthermore, it illustrates again the strong difference between on one side the LFU- and LRU-AC, which handle about 70% of the requests in the Fog, and the Blind-LB, which sends more than 90% of the requests in the Cloud. Popularity-based AC thus seems an appropriate approach to take maximum advantage of edge resources.

7.2 Implementation evaluation

The P4 data plane introduced in Section 6.2 has been synthesized for the NetFPGA-SUME platform with the P4→NetFPGA framework. The targeted false positive rate is $f_a = 1\%$, yielding $n_h = 8$. The maximum number of elements in the ABF is taken to be $n_a = 9.2{\cdot}10^4$ (corresponding to an equivalent LRU filter size $k_{LRU} = 1.3{\cdot}10^5$, i.e., the conservative settings introduced in Section 7.1), yielding $\frac{m_{ABF}}{2} = 2.1{\cdot}10^6$ bits in each filter. Thus, one must take $s_{hash} = \lceil \log_2 \frac{m_{ABF}}{2} \rceil = 21$ to be able to address all elements in each filter. Finally, we take a key of $s_{key} = 48$ bits, allowing to address up to $280 \cdot 10^{12}$ objects, way above the catalogue size $R = 10^7$.

Resource usage on the FPGA board is reported in Table 7. As can be seen by comparing these results to those in Table 5, most of the logic (LUT) is consumed by the NetFPGA framework (MAC for the network interfaces and packet processing). However, an important part of the BRAM resources is consumed by the Bloom filter modules. In total, the BRAM usage is 47%, confirming

Table 7. NetFPGA Dataplane performance

FPGA resource usage				**Forwarding performance**	
	LUTs	BRAM	Power (W)	Latency (μs)	2.62
2 Bloom filters	715	130	0.089	Throughput (Mpps)	16.7
P4	103424	564	8.550		
Total	104139	694	8.639		
Available resources	433200	1470	—		
Resource consumption	24.0 %	47.2 %	—		

that the limits of the platform are reached and that a standard LRU filter (which would consume 4× more memory as shown in Section 5.4) could not be implemented.

The performance of the P4 data plane is evaluated by injecting in the FPGA simulator a batch of 4096 packets (directly after the Ethernet interface, so as to outreach the 10 Gbps limit), and measuring the corresponding latency and throughput. Results are reported in Table 7, showing that packets can be forwarded over the 10 Gbps line-rate (14.4 Mpps) while providing low latency. The obtained throughput results are consistent with the throughput of the Bloom filter atom (one operation every $n_h + 4 = 12$ cycles at 200 MHz).

8 RELATED WORK

The importance of locating computing resources topologically close to users has been put forward under diverse forms in the community: Fog computing [1], Mobile-Edge-Computing [51], hybrid Cloud [12]. In particular, Niu et al. [12] explore a similar problem to ours: the use of a local Cloud infrastructure to handle sudden bursts of traffic. They also use a Markov-chain based model for computing the expected completion time and devise a scheduling algorithm between hybrid and public Cloud using an optimization problem under budget constraints. However, they do not exploit any knowledge of the request popularity, thus falling under the hard limit that we exposed for the Blind-AC. Malawski et al., study costs optimization between a hybrid Cloud and multiple public Clouds with different pricing models under a deadline constraint [52]. However, they focus on task optimization, looking at a model closer to scheduling for scientific computing rather than live optimization of user requests. Du et al. [13] formulate a joint resource allocation and offloading between user devices, Fog networks and Cloud networks, so as to minimize energy consumption and request processing delay.

Using popularity to load-balance content in ICN networks has already been explored. In [22], the authors propose to count incoming packets and use exponential smoothing. As argued in Section 4.3, this approach is not flexible to popularity changes and requires knowledge of the application. Furthermore, the authors aim at load-balancing packets over homogeneous paths, whereas the Fog/Cloud offload problem is essentially heterogeneous. Similarly, the authors of [23] propose to use a k-LRU filter to learn popularity for load-balancing ICN interests over multiple paths. They then measure per-name latency to optimize the distance to the next object. However, the authors do not specify the settings of the k-LRU filter, and only consider the effect of their load-balancing on the data creation process. Finally, in [6], the authors use the ICN-Fog node as a classifier between static and dynamic data, thus preventing upstream caches to store dynamic data. They do not, however, consider the data processing that happens in many Fog applications.

9 CONCLUSION

In this paper, methods for guaranteeing response time in Fog deployments based on popularity-aware AC were introduced. Two specific AC schemes were introduced: the oracle-based LFU-AC

and the probabilistic LRU-AC. Their effectiveness was demonstrated using an analytical model for various application parameters. An implementation of the LRU-AC on state-of-the-art FPGA hardware using ABF was then proposed and its soundness was demonstrated through analytical modeling. This implementation is shown to provide AC at 10 GbE line-rate throughput with a 3 µs latency. It increases the Fog acceptance rate by almost 10× w.r.t. content-blind approaches while maintaining the latency excess rate stable. A future research question which remains open by this work is the study of a concrete application scenario with real traffic traces, for instance in a smart city setup.

REFERENCES

[1] F. Bonomi *et al.*, "Fog computing and its role in the internet of things," in *Proc. 1st Edition Workshop Mobile Cloud Computing*. ACM, 2012.

[2] K. Hong *et al.*, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proc. 2nd SIGCOMM Workshop Mobile Cloud Computing*. ACM, 2013.

[3] M. Taneja and A. Davy, "Resource aware placement of IoT application modules in Fog-Cloud computing paradigm," in *2017 IFIP/IEEE Symp. Integrated Network and Service Manage. (IM)*, May 2017, pp. 1222–1228.

[4] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Latency-aware application module management for Fog computing environments," *ACM Trans. Internet Technology*, vol. 19, no. 1, Nov. 2018.

[5] J. Khan, C. Westphal, and Y. Ghamri-Doudane, "A content-based centrality metric for collaborative caching in information-centric fogs," in *IFIP-Networking - ICFC*, 2017.

[6] M. Wang *et al.*, "Fog computing based content-aware taxonomy for caching optimization in information-centric networks," in *IEEE Conf. Comput. Commun. Workshops*, May 2017.

[7] Z. Chen *et al.*, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proc. 2nd ACM/IEEE Symp. Edge Computing*. ACM, 2017.

[8] M. Chiang, B. Balasubramanian, and F. Bonomi, *Fog for 5G and IoT*. John Wiley & Sons, 2017.

[9] "Aws greengrass," https://aws.amazon.com/greengrass.

[10] A. Rauniyar *et al.*, "Crowdsourcing-based disaster management using fog computing in internet of things paradigm," in *2016 IEEE 2nd international conference on collaboration and internet computing (CIC)*. IEEE, 2016, pp. 490–494.

[11] Z. Su *et al.*, "A secure content caching scheme for disaster backup in fog computing enabled mobile social networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4579–4589, 2018.

[12] Y. Niu *et al.*, "Handling flash deals with soft guarantee in hybrid cloud," in *Proc. INFOCOM*. IEEE, 2017.

[13] J. Du *et al.*, "Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee," *IEEE Trans. Commun.*, vol. 66, no. 4, pp. 1594–1608, 2018.

[14] R. B. Miller, "Response time in man-computer conversational transactions," in *Proc. AFIPS Fall Joint Comput. Conf.*, 1968.

[15] M. Enguehard, G. Carofiglio, and D. Rossi, "A popularity-based approach for effective cloud offload in fog deployments," in *2018 30th Int. Teletraffic Congr. (ITC 30)*, vol. 1. IEEE, 2018, pp. 55–63.

[16] M. Blott *et al.*, "Achieving 10gbps line-rate key-value stores with fpgas." in *HotCloud*, 2013.

[17] M. Yoon, "Aging bloom filter with two active buffers for dynamic sets," *IEEE Trans. Knowledge and Data Eng.*, vol. 22, no. 1, pp. 134–138, 2010.

[18] N. Zilberman *et al.*, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[19] L. Muscariello *et al.*, "Hybrid Information-Centric Networking," Internet Engineering Task Force, Internet-Draft draft-muscariello-intarea-hicn-00, Jun. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-muscariello-intarea-hicn-00

[20] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Communication Review*, vol. 44, no. 3, pp. 87–95, jul 2014.

[21] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. of 2011 Int. Conf. for High Performance Computing, Networking, Storage and Anal.* ACM, 2011.

[22] T. Janaszka, D. Bursztynowski, and M. Dzida, "On popularity-based load balancing in content networks," in *Proc. 24th Int. Teletraffic Congr.*, 2012, p. 12.

[23] G. Carofiglio, L. Mekinda, and L. Muscariello, "Focal: Forwarding and caching with latency awareness in information-centric networking," in *Globecom Workshops*. IEEE, 2015, pp. 1–7.

[24] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," in *Proc. 24th Int. Teletraffic Congr.*, 2012.

[25] G. Rossini and D. Rossi, "Evaluating ccn multi-path interest forwarding strategies," *Computer Communications*, vol. 36, no. 7, pp. 771–778, 2013.
[26] L. Breslau *et al.*, "Web caching and zipf-like distributions: Evidence and implications," in *Proc. INFOCOM*, vol. 1. IEEE, 1999.
[27] C. Imbrenda, L. Muscariello, and D. Rossi, "Analyzing cacheable traffic in isp access networks for micro cdn applications via content-centric networking," in *Proc. 1st ACM SIGCOMM Conf. Inform.-Centric Networking*, Sep 2014.
[28] S. Traverso *et al.*, "Temporal locality in today's content caching: why it matters and how to model it," *ACM SIGCOMM Comput. Communication Review*, vol. 43, no. 5, 2013.
[29] B. Urgaonkar *et al.*, "An analytical model for multi-tier internet services and its applications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 291–302, 2005.
[30] M. Nabe, M. Murata, and H. Miyahara, "Analysis and modeling of world wide web traffic for capacity dimensioning of internet access lines," *Performance evaluation*, vol. 34, no. 4, 1998.
[31] J. Boyer *et al.*, "Heavy tailed m/g/1-ps queues with impatience and admission control in packet networks," in *Proc. INFOCOM*, vol. 1. IEEE, 2003.
[32] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *J. Select. Areas in Commun.*, vol. 20, no. 7, 2002.
[33] F. P. Kelly, *Reversibility and stochastic networks*. Cambridge University Press, 2011.
[34] G. F. Newell, "The m/g/∞ queue," *J. Appl. Math.*, vol. 14, no. 1, 1966.
[35] Y. Desmouceaux *et al.*, "A content-aware data-plane for efficient and scalable video delivery," in *Proc. 16th IFIP/IEEE Int. Symp. Integrated Network Manage.*, 2019, to appear.
[36] D. Shasha and T. Johnson, "2q: A low overhead high performance buffer management replacement algoritm," in *Proc. 20th Int. Conf. Very Large Databases*, 1994.
[37] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM Trans. Modeling and Performance Evaluation of Computing Syst.*, vol. 1, no. 3, p. 12, 2016.
[38] M. Enguehard, Y. Desmouceaux, and G. Carofiglio, "Efficient latency control in Fog deployments via hardware-accelerated popularity estimation (technical report)," https://enguehard.org/papers/lru-ac-techrep2019.pdf, 2019.
[39] K. Svanberg, "The method of moving asymptotes—a new method for structural optimization," *Int. J. Numerical Methods in Eng.*, vol. 24, no. 2, 1987.
[40] S. G. Johnson, "The NLopt nonlinear-optimization package," http://ab-initio.mit.edu/nlopt.
[41] M. Enguehard and Y. Desmouceaux, "marceleng/queueing-network-simulator: a simulator for queueing networks," https://github.com/marceleng/queueing-network-simulator, Jan 2019.
[42] J.-L. Brelet, "Using block ram for high performance read/write cams," *Xilinx Inc., Application Notes*, vol. 204, 2000.
[43] S. Dharmapurikar *et al.*, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, jan 2004.
[44] H. Song *et al.*, "Fast hash table lookup using extended bloom filter: an aid to network processing," *ACM SIGCOMM Comput. Communication Review*, vol. 35, no. 4, pp. 181–192, 2005.
[45] ——, "IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards," in *Proc. 28th IEEE Conf. Comput. Commun. (INFOCOM)*, IEEE. IEEE, apr 2009, pp. 2518–2526.
[46] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *Proc. 5th Int. Conf. Emerging Networking Experiments and Technologies - CoNEXT'09*, ACM. ACM Press, 2009, pp. 313–324.
[47] G. Xylomenos et al., "A survey of information-centric networking research," *IEEE Commun. Surveys and Tutorials,*, vol. 16, no. 2, pp. 1024–1049, Jul. 2014.
[48] S. Ibanez *et al.*, "The P4 → NetFPGA workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 1–9.
[49] A. Sivaraman *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proc. 2016 ACM SIGCOMM Conf.* ACM, 2016, pp. 15–28.
[50] M. Dietzfelbinger *et al.*, "A reliable randomized algorithm for the closest-pair problem," *J. of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997.
[51] Y. C. Hu *et al.*, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, 2015.
[52] M. Malawski, K. Figiela, and J. Nabrzyski, "Cost minimization for computational applications on hybrid cloud infrastructures," *Future Generation Comput. Syst.*, vol. 29, no. 7, 2013.