Self-adjusting advertisement of cache indicators with bandwidth constraints

Self-adjusting Advertisement of Cache Indicators with Bandwidth Constraints

Itamar Cohen*, Gil Einziger†, and Gabriel Scalosub‡
*Politecnico di Torino, Italy
†Department of Computer Science, Ben-Gurion University of the Negev, Israel
‡School of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Israel

itamar.cohen@polito.it, gilein@bgu.ac.il, sgabriel@bgu.ac.il,

***Abstract*—Cache advertisements reduce the access cost by allowing users to skip the cache when it does not contain their datum. Such advertisements are used in multiple networked domains such as mobile ad-hoc networks, wide area networks, and information-centric networking. The selection of an advertisement strategy exposes a trade-off between the access cost and bandwidth consumption. Still, existing works mostly apply a trial-and-error approach for selecting the best strategy, as the rigorous foundations required for optimizing such decisions is lacking.**

Our work shows that the desired advertisement policy depends on numerous parameters such as the cache policy, the workload, the cache size, and the available bandwidth. In particular, we show that there is no ideal single configuration. Therefore, we design an adaptive, self-adjusting algorithm that periodically selects an advertisement policy. Our algorithm does not require any prior information about the cache policy, cache size, or workload, and does not require any apriori configuration. Through extensive simulations, using several state-of-the-art cache policies, and real workloads, we show that our approach attains a similar cost to that of the best static configuration (which is only identified in retrospect) in each case.

I. Introduction

Caching is a fundamental optimization technique where a small subset of the data is stored in a cache, which is cheaper to access than the regular storage. Caching is common to the point where it is present in some form in almost all computing environments and systems, ranging from micro-controllers, through PCs and servers, and onto distributed cloud services.

In large distributed systems, caches often further optimize performance by advertising their content. Such advertisements allow clients to bypass the cache when it is unlikely to contain the requested datum, thus reducing the total access cost, where "cost" can reflect bandwidth, access time, or energy [1]–[3]. Content advertisements are used in mobile ad-hoc networks [4], [5], content delivery networks (CDN) [3], [6], [7], information centric networking (ICN) [8], [9], and in wide-area networks [10].

Ideally, the advertisement policy would reflect the cached content at any given time, but such a solution is bandwidth-intensive. Content advertisement is often restricted to a bandwidth budget. Therefore, systems often compromise on advertising approximate *indicators* that approximate the cached content [11]–[13] to reduce the advertisement size, at the cost of some probability of generating false-positive errors [1], [6], [11], [13]–[17]. Such errors imply that indicators sometimes mistakenly assert that a datum is stored in the cache, causing redundant cache accesses.

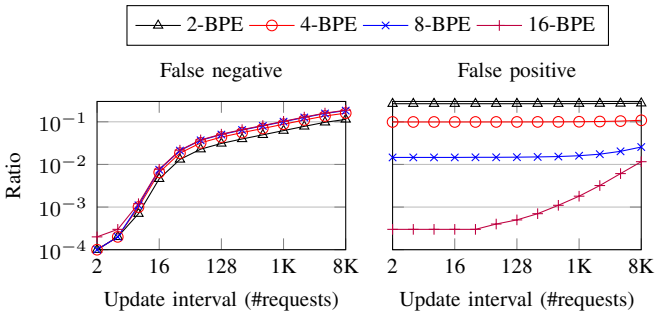

Fig. 1. Effect of the update interval on the false-negative errors (left) and false-positive errors (right) for optimally configured Bloom filter indicators. Both axis are in log-scale, and the cache size is 8K, policy is LRU, trace is F1 (described in Sec. VI).

When constantly sending indicator advertisements, the advertised content remains fresh in the sense that it accurately reflects the (approximate) state of the cached content, and we only experience errors due to hash collisions. In bandwidth-constrained environments, insisting on freshness mandates the usage of relatively small (and inaccurate) indicators to cope with the bandwidth *budget*, which would otherwise imply packet drop and increased error rate. An alternative is to send an indicator advertisement only *occasionally*. When using such an approach, the advertised content gradually becomes *stale*, in the sense that it takes time for the indicator available at the clients to reflect changes in the cached content, which again leads to an increased error rate.

To illustrate the above scenarios, consider an advertisement transmission, followed by having the cache admit a new item (x) and evict some item (y), such that these events are not (yet) advertised to the clients. When the client tests for y, the indicator falsely indicates that y is in the cache, resulting in a *false-positive* error. Similarly, a query for x is likely to falsely indicate that x is not in the cache (as it wasn't in the cache at the time the advertisement was sent), resulting in a *false-negative* error. Thus, staleness creates both false-positive and false-negative errors, and we expect fewer of these errors,

* The work was done while this author was with Ben-Gurion University.

the more frequently we refresh the advertisements.

Figure 1 shows the percentage of false indications as a function of the time between subsequent advertisements, referred to as the *update interval*. Here, the advertisement size is expressed by the number of *Bits Per cached Element* (BPE), while the update interval is expressed in terms of requests between subsequent updates. Notice that the X-axis and the Y-axis are in logarithmic scale. Figure 1 (left) shows that the false-negative ratio increases with the update interval, as we qualitatively explained above, and that it may reach non-negligible rates. Figure 1 (right) demonstrates that increasing the update interval also affects the false-positive errors, but the effect is less pronounced than it is on false-negatives.

The combination of approximate indicators and staleness makes selecting a cache advertisement policy a challenging task. Intuitively, the cache may send an advertisement of size x once every y requests, or a more accurate advertisement of size $\beta \cdot x$ ($\beta > 1$) once every $\beta \cdot y$ requests. Both options require similar bandwidth, but it is unclear which of them would do better. Note that although the probability of false-positive due to hash collisions is relatively well understood [11], the errors caused by stale advertisements are difficult to predict as they depend on the workload, the cache policy, and the cache size. Furthermore, existing works do not address the complex interplay between advertisement strategy, indicator size, update interval, and access cost [4], [10], [18], [19]. Instead, most works fix an advertisement policy by crude estimations and rules-of-thumb, rather than by optimizing it according to the system being used, and the workload being served [20], [21]. While such an approach may work in some scenarios, changes in either the cache policy, the workload, the cache size, or the budget, may deteriorate performance significantly, as we demonstrate in Sec. IV.

Our contribution: Our proposed solutions make advertisements easier to use in caching systems. Our first contribution is the formulation of the problem and a rigorous study of the problem domain. We perform a simulation-based study using state-of-the-art cache policies and real workloads, demonstrating that the best advertisement strategy depends on numerous factors such as the cache policy, the cache size, the workload, and the communication budget. Our study implies that using previously developed approaches require extensive testing under varying conditions, to optimize the advertisement policy. Worse yet, every change in the system parameters requires revisiting the previous decisions. Such optimization is rarely done in practice because it is time-consuming, and more importantly, because some affecting parameters are uncontrollable by the system designers. E.g., the workload may change dynamically, and the cache size may vary between deployments.

In light of this challenge, we suggest an adaptive, self-adjusting, algorithm that periodically updates its advertisement strategy. Through an extensive simulation study using real workloads, and state of the art cache policies, we show that our algorithm matches the performance of the best static strategy, that *can only be determined in retrospect*. Adopting our proposed solution implies that system designers are no longer required to optimize their advertisement policy. The algorithm adapts to the system configuration, as well as to the dynamic workload characteristics in runtime.

II. Related work

Indicators are used to periodically advertise the cache content in multiple networking environments, including wide-area networks [10], content delivery networks [3], [6], [7], [22], information centric networking [8], [9], [23]–[25], and wireless networks [4], [5]. Indicators make use of randomized hash-based data structures such as Bloom filters [11], [12], [26], and fingerprint hash tables [13], [16].

While conceptually simple, there are numerous challenges in utilizing indicators. For example, the study in [1] shows that indicators may degrade the performance in some scenarios in what they refer to as the "Bloom Paradox". The work of [27], tackles a distributed scenario where multiple caches send indicators, and the client needs to formulate an access strategy that minimizes the access cost. The work of [28] suggests methods to reduce the transmission overheads of indicators, at the expense of larger local memory consumption. The works [29], [30] reduce the transmission overheads by accurately advertising important information, while allowing less important information to be stale, or less accurate. The work [15] surveys many optimizations to indicators, such as the support for removals and dynamically scaling. While such structures are used by our work, the exact construction is not a central part of our work.

Staleness is a major challenge for cache advertisements. The work of [31] suggests advertisement strategies that make the impact of staleness on the false-positive ratio and the false-negative ratio predictable. However, their proposed solution requires sending an update whenever sufficiently many bits of the indicator have changed. This might impose a hefty toll on the bandwidth consumption, and may well violate the available budget. The works [4], [10], [18], [19] perform simulation studies in order to identify "reasonable" advertisement policies for some concrete settings (workload, cache size, cache policy, miss penalty). Several other works [32]–[36] address the problem of maintaining a bandwidth budget when sending advertisements using a trial-and-error approach. Such an approach usually requires a lot of effort on the part of system designers, and as the workload may change, one might still end up exceeding (or under-utilizing) the budget. In comparison, our work includes an autonomous self-adjusting mechanism that utilizes the budget efficiently without resorting to trial-and-error.

In Squid cache [21], the update interval is fixed and defaults to sending an update once in a hour [20], [21]. Since such a solution is problematic, Squid's spec defines the problem of scaling the update interval as an "open issue" [21]. In comparison, our work provides a good solution that adapts the update interval, and the indicator size, to the current situation. To the best of our knowledge, our work is the first to automatically optimize the advertisement strategy, while efficiently utilizing a fixed bandwidth budget.

TABLE I
LIST OF NOTATION

Symbol	Meaning
C	Cache size [number of elements]
S_t	Set of data items in the cache at time t
I_t	Indicator at time t
$I_t(x)$	Indication for datum x
$\lvert I_{\min}\rvert, \lvert I_{\max}\rvert$	Minimal, maximal feasible indicator size [bits]
FP_t, FN_t	False-positive, false-negative estimate of indicator I_t
M	Miss penalty
$\lvert I\rvert$	Indicator size [bits]
u	Update interval [number of cache requests]
$u_{\min}, u_{\max}$	Minimal, maximal update interval [requests]
B	Bandwidth budget [bits/request]
T	Re-configuration interval [number of cache requests]
α	tradeoff parameter balancing accuracy / responsiveness / bandwdith variation
P	Cache policy
W	Workload (trace)

III. SYSTEM MODEL

This section formally defines our system model, as well as our notation (which is also summarized in Table I).

Cache and cost model: We consider a cache that contains, at any time t, some set of items S_t. The maximal number of items in the cache is C. Clients issue a sequence of requests/queries for data. The clients may request the data from the cache, or from some remote storage. Without loss of generality, we refer to each request as arriving in a unique *time slot* t. Yet, when clear from the context, we sometimes omit the subscript t.

Accessing the cache incurs some *access cost*, which we normalize to 1 without loss of generality. Cache access cost is due whenever the cache is accessed, even if the requested datum is not in the cache. If a cache access for datum d at time t results in a *cache miss*, i.e., $d \notin S_t$, then an additional *miss penalty* $M > 1$ is incurred. This miss penalty is also imposed whenever the cache is not accessed for a given request. The miss penalty reflects the cost of retrieving the datum from some remote storage. The cost M includes notifying the cache about the data access, in which case the cache may decide to admit the datum towards serving future requests, depending on the policy being applied for admitting and evicting items from the cache. The *service cost* is the sum of the cache access cost, and the miss penalty cost. To make a meaningful comparison of performance, we focus our attention on the *average service cost* of all the requests in the sequence, thus following similar cost models studied in previous works [27], [37].

Indicators, update intervals, and configurations: At any time t, the cache may advertise an *indicator* I_t that approximates S_t at time t. For any indicator I_t, given a datum x, a *positive indication* of I_t indicates that $x \in S_t$, while a *negative indication* of I_t indicates that $x \notin S_t$. I_t may generate false-positive and false-negative errors. A positive indication is said to be a *false-positive* when $x \notin S_t$. Similarly, a negative indication is said to be a *false-negative* when $x \in S_t$. We let FP_t and FN_t denote the estimates at time t of the false-positive probability, and the false-negative probability, of a cache request, respectively. We let $|I_t|$ denote the size of the indicator I_t in bits. To use only feasible sizes, the indicator size should be within some predefined range $[|I_{\min}|, |I_{\max}|]$.

Given some positive integer T, we consider a non-overlapping partitioning of time (or equivalently, the sequence of requests) into *segments* of length T. The *update interval* u_t is the number of requests between subsequent indicator updates that the cache sends to the users. At any time t, u_t represents the time between the last update that was sent, and the next update scheduled to be sent. When considering dynamic algorithms, we allow the value of u_t to be adjusted only at the end of a segment.

The update interval is at least $u_{\min}$. One could use $u_{\min} = 1$, but a slightly higher interval enables piggybacking indicator updates on packets carrying cached data payloads, to avoid transmission overheads [38]. We also use a maximal update interval denoted $u_{\max}$ (which we discuss in the sequel). We refer to the tuple $(|I_t|, u_t)$ as a *configuration*.

An advertisement that includes the full indicator I, is called a *full-indicator* update. Alternatively, an update that contains the list of bits in the indicator that have flipped since the previous advertisements is called a *delta* update. Specifying the location of each bit in the indicator requires $\log|I|$ bits. We assume that the cache uses a delta update whenever this consumes less bandwidth than sending a full indicator, namely, when the number of bits flipped in the indicator since the last update is less than $\frac{|I|}{\log|I|}$.

Itamar: Added a reviewer's comm when the cache s

Bandwidth constraints: To model the system's bandwidth constraint, we use the previously defined partitioning of time into segments. The transmitted *bandwidth cost* of configuration $(|I_t|, u_t)$ over a segment of length T ending at time t is the *average* number of update bits per request, being *sent* to the user during the segment. We denote this cost by BW_t. Since indicators are usually of size $\Theta(C)$, and since we would like to potentially allow the algorithm to transmit more than one update during a segment, we require that $T \geq \max\{u_{\max}, C\}$. In particular, we choose $T = \alpha \cdot \max\{u_{\max}, C\}$, for some positive integer α. Parameter α serves to control the tradeoff between (i) the variance of the statistics gathered during a segment, and (ii) the dynamic response of the algorithm across segments. I.e., if α is small, then statistics are gathered over a short interval, and may capture only very transient behavior which could be very different in the following segment. On the other hand, if α is large, then the algorithm maintains its current configuration longer, even though workload and system characteristics may change significantly during the segment.

We target system configurations that satisfy budget constraints, defined by a *bandwidth budget* of B bits/request. The budget constraint requires that the bandwidth cost in each segment is at most B. We note that when sending a full indicator in each update we must have

$$\frac{|I_t|}{u_t} \leq B. \tag{1}$$

We use this equation for determining $u_{\max}$ as the minimal value satisfying Eq. 1, which implies that $u_{\max} = \left\lfloor \frac{|I_{\max}|}{B} \right\rfloor$.

A configuration $(|I|, u)$ is said to be *static* if for every time t, $|I_t| = |I|$ and $u_t = u$. Such a configuration is said to satisfy the budget constraint if in *every* segment of length T, the overall bandwidth cost of using $(|I|, u)$ is at most $B \cdot T$, i.e. $BW_t \leq B$ for every time t in which a segment ends. We note that due to the dynamic nature of caching environments, it may be impossible to verify *a-priori* that a specific static configuration does not violate the budget constraint. In particular, a configurations that uses delta updates might end up violating the budget if there are too many updates.

In our work, we are interested in *dynamic* configurations that may re-scale and adjust both $|I_t|$ and u_t over time. Such dynamic configurations may also occasionally end up oversubscribing the network. However, using dynamic configurations one can strive to satisfy the budget constraint over all segments, by adjusting to the current workload pattern, while (implicitly or explicitly) taking into account additional system parameters related to, e.g., the cache size, or the cache policy. Although dynamic configurations may sometimes violate the budget constraint, a careful adjustment of the configurations throughout the system's lifetime may reduce this violation significantly (e.g., compared to static configurations).

Since configurations (either static or dynamic) may end up violating the budget constraint, we apply a *network policing* mechanism that enforces the budget constraint as follow: At the beginning of each segment, the cache receives $B \cdot T$ tokens. Once the overall number of bits sent for indicator advertisement during the segment reaches $B \cdot T$, all further updates during the segment are dropped by the network policing mechanism. This model conforms to common network policing behaviour that may selectively drop packets when a user oversubscribes its allotted resources. In this sense, the transmitted bandwidth cost BW_t may indeed be larger than the budget, but in effect, the network will never forward more traffic than the amount prescribed by the budget B. Lastly, we note that α also serves to define the time horizon for which we enforce the budget violation. I.e., choosing a larger value for α implies that we allow larger fluctuations in bandwidth usage during a segment, as long as the overall bandwidth cost is maintained over the entire segment.

In what follows, we consider distinct system *scenarios*, where each scenario is defined by the cache size C, policy P, workload W, and budget B. We denote such a scenario by (C, P, W, B). We will be studying static advertisement configurations for a variety of scenarios, as well as dynamic advertisement configuration strategies that *adapt* to dynamically changing scenarios. Our work considers the problem of (dynamically) adjusting the configuration so as to minimize the (average) service cost within a given bandwidth budget. To simplify expressions throughout our work, all logarithms are of base 2.

IV. Motivation and Preliminaries

This section provides insights into the performance of static configurations to further motivate dynamic advertisement strategies. We present the results of several experiments, which use several real-life workloads and state-of-the-art cache policies. Our results show that there is no "one-size-fits-all" configuration and that using static configurations may lead to substantial performance degradation in highly dynamic systems.

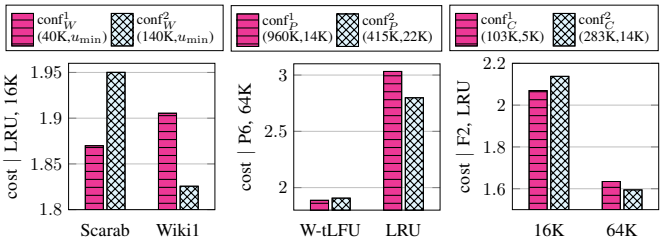

Fig. 2. Differences in cost for static configurations when varying workload (left), policy (center), and cache size (right). For each aspect being compared, one configuration is better for one value (workload / policy / cache size), whereas another configuration is better for the other value.

For each scenario (C, P, W, B) considered, we perform a grid-search of static configurations and find the best configuration that satisfies the bandwidth budget constraint for this scenario. We then compare the performance of these configurations when used for scenarios that differ by merely one aspect, where we focus here on changing either the cache size, the policy, or the workload, to exemplify the effect each of these system aspects has on system performance, given a specific configuration. In this evaluation, we set $u_{\min} = 10$, $|I_{\min}| = 2.5 \cdot C$, $|I_{\max}| = 15 \cdot C$, $M = 3$, $\alpha = 10$, and $B = 20$.[1] We demonstrate configurations that exhibit very good performance for some scenarios, but changing merely one aspect in the scenario results in significant performance degradation.

Figure 2 shows the results of several such experiments. In Figure 2 (left) we consider two workloads W_1, W_2, where W_1 is the Scarab trace, and W_2 is the Wiki1 trace, with conf_1^W and conf_2^W being their best static configurations, respectively. The policy is LRU, and the cache size is $C = 16$K. One can note that for each workload W_i, using conf_j^W $j \neq i$ incurs a toll as large as 5% compared to using conf_i^W. In Figure 2 (center), one can see similar results hold for varying the cache policies. Here we consider policies P_1, P_2, where P_1 is W-tLFU and P_2 is LRU, with conf_1^P and conf_2^P being their best static configurations, respectively. The workload is P6, and the cache size is $C = 64$K. In Figure 2 (right), one can see the same effect is manifested for the case where we vary the cache size. Here we consider cache sizes $C_1 = 16K$ and $C_2 = 64K$, with conf_1^C and conf_2^C being their best static configurations, respectively. The cache policy, in this case, is LRU, and the workload is F2. In all plots, the configurations are specified in the legend (up to rounding to the nearest K).

V. Algorithm CAB

In this section, we introduce the Cache-indicators Advertisement with Budget constraint (CAB) algorithm. The pseudo-code of CAB is provided in Algorithm 1. We begin with a

[1] We describe the specific workloads and cache policies, as well as the methodology of our grid search and the choice of parameters, in Sec. VI.

high-level description of our algorithmic concepts and then detail the optimization it employs in its decisions.

A. High-Level Overview

CAB has two challenges, one for each of the following two *regimes*: (i) when sending full indicators, and (ii) when sending delta updates.

1) Full-Indicator Regime: When sending full indicators, the problem is to find the right balance between the update interval and the accuracy of the indicator. In this regime, a configuration $(|I_t|, u_t)$ sends an advertisement of size $|I_t|$ once in u_t requests. Recall that by Eq. 1, we must satisfy $\frac{|I_t|}{u_t} \leq B$. Here, our approach is to choose, among all the possible configurations that satisfy Eq. 1, a configuration that equalizes the additional cost caused by false-negative errors (controlled by the update interval) and false-positive errors (mainly controlled by the indicator size).

2) Delta Regime: When only sending the bits that have changed since the previous advertisement, increasing the update interval would usually have very little effect on the consumed bandwidth. Intuitively, updating about a single change in the cache once every u requests consumes a similar bandwidth as updating about x changes in the cache once in every $x \cdot u$ requests. The only exception to this rule-of-thumb happens due to hash collisions, e.g., when some of the x changes in the cache occasionally flip and re-flip the same bit in the indicator. However, when u is small, this effect is negligible. Hence, we favor sending updates as soon as possible, i.e., we set the update interval to $u_{\min}$.

Once the update interval is fixed, the remaining challenge is to dynamically-scale the indicator size $|I|$ to utilize all the budget without exceeding it. However, sending changes may become infeasible (e.g., when the hit ratio drops and more items are admitted to the cache), and in that case, we might need to return to the full indicator regime. We trigger such a transition when we exceed the budget while using the minimal indicator size, $|I_{\min}|$. To do so, we increase the update interval to the "safe zone" of sending full indicators, i.e., satisfying Eq. 1, in which case we can assure compliance with the budget constraint. Such a step allows us to search for better configurations in subsequent segments (as also demonstrated in Sec. VI).

Co-similarity and system lifetime: Our algorithm implicitly assumes that the behaviour of the cache in the next segment will be similar to its behaviour in the current segment. While such an assumption is not always correct, many underlying caching algorithms make similar assumptions. E.g., adaptive caches [39], [40] assume that the past access pattern provides a good indication of the future access pattern. Thus, our assumptions are reasonable in workloads where adaptive caching works well [39]–[41]. It should be noted that in Sec. VI we demonstrate the effectiveness of our approach in a variety of scenarios and workloads. CAB is oblivious of the cache policy that manages the evictions and admissions, as we show in Sec. VI. Instead, CAB uses only the information of false-positive and false-negative errors, as well as the bandwidth cost, to adapt its advertisement strategy. Such information indirectly includes some details about the cache policy, e.g., when the cache policy rapidly changes the cached content, then a large update interval is likely to cause plenty of false-negative errors. Alternatively, when the cache policy hardly changes the cached content, the same (long) update interval results in few false-negative errors.

Algorithm 1 CAB(B)

1: $|I_0| = |I_{\min}|, u_0 = \left\lfloor \frac{|I_0|}{B} \right\rfloor$
2: $T = \alpha \cdot \max\{u_{\max}, C\}$
3: **for** every time slot $t = T, 2T, 3T, \ldots$ **do**
4: **if** $\exists$ full indicator update during $[t-T, t)$ **then**
5: $|I_{t+1}| = \text{FitToRange}\left(\left\lfloor |I_t| \sqrt{\frac{FP_t}{(M-1)\cdot FN_t}} \right\rfloor\right)$
6: $u_{t+1} = \left\lfloor \frac{|I_{t+1}|}{B} \right\rfloor$
7: **else** ▷ all updates are delta-updates
8: **if** $|I_t| > |I_{\min}|$ **or** $BW_t \leq B$ **then**
9: $u_{t+1} = u_{\min}$
10: $|I_{t+1}| = \text{FitToRange}\left(\left\lfloor e^{W\left(\frac{B|I_t|\log|I_t|}{BW_t}\right)} \right\rfloor\right)$ ▷ Lambert W function
11: **else** ▷ $|I_t| = |I_{\min}|$ **and** $BW_t > B$
12: $|I_{t+1}| = |I_t|$ ▷ indicator size remains $|I_{\min}|$
13: $u_{t+1} = \left\lfloor \frac{|I_{t+1}|}{B} \right\rfloor$
14: **end if**
15: **end if**
16: **end for**

17: **procedure** FitToRange(size)
18: **return** $\max\{\min\{\text{size}, |I_{\max}|\}, |I_{\min}|\}$
19: **end procedure**

B. Detailed Description

The CAB algorithm (formally defined in Algorithm 1) is an implementation of the approach outlined above. The algorithm begins with an arbitrary configuration that satisfies the budget constraint (line 1). The algorithm then sets the segment size T (line 2), such that the configuration may be updated at the end of each segment (at times $t = T, 2T, \ldots$). T is set to ensure that sufficient statistics can be obtained during a segment, for determining the configuration to be used towards the following segment. This is controlled by the value of α, as described in Sec. III. We note that it usually suffices to set α to a small constant number (e.g., throughout our evaluation in Sec. VI we use $\alpha = 10$). At every time $t = T, 2T, \ldots$, we let FP_t and FN_t denote the false-positive ratio and the false-negative ratio during the segment ending at t, and we recall that BW_t denotes the bandwidth cost (i.e., number of bits being sent by CAB, divided by T) during this segment. Whenever a configuration is chosen, the procedure FitToRange (lines 17-19) ensures that the indicator size is within the prescribed bounds, i.e., in $[|I_{\min}|, |I_{\max}|]$. For determining the indicator's size and the update interval, CAB distinguishes between three cases (marked by different shaded colors in Algorithm 1). In what

follows, we discuss the algorithmic design criteria for each of these cases.

1) Full indicator updates: The first case (lines 4-6) is when a full indicator is sent in (at least) one of the updates during the segment ending at t. We refer to this case as having the algorithm work in *Mode 1* (shaded red). In this operation mode, we expect to have $FP_t > 0$ due to using indicators (which by nature provide merely an approximate representation of the cached content), and $FN_t > 0$ due to staleness. CAB adjusts its indicator size and update interval in an attempt to strike a balance between the loss of performance caused by false-negatives, and false-positives. This approach makes the reasonable assumption that false-negatives increase when increasing the update interval (as exhibited, e.g., in Fig. 1), and false-positives increase when decreasing the indicator size. Due to the budget constraint, the indicator size and the update interval are positively correlated.

For understanding the choice made in line 5, one should note that (i) a false-positive indication incurs an unwarranted extra cost of 1, whereas (ii) a false-negative indication incurs an unwarranted extra cost of $(M-1)$ (since we could have incurred a cost of 1 by merely accessing the cache). It follows that targeting having $(M-1)$ false-positives (which are relatively cheap) for every single false-negative (which is relatively expensive) would balance the unwarranted extra costs. I.e., we would like to have $(M-1)\cdot FN = FP$, or equivalently, $\frac{FP}{(M-1)\cdot FN} = 1$. When considering FP_t and FN_t, if $(M-1)\cdot FN_t < FP_t$, we would like to decrease the number of false-positives, even at the cost of some additional false-negatives, which translates to increasing the indicator size (and in turn also increasing the update interval). If, on the other hand, $(M-1)\cdot FN_t > FP_t$, we would like to do the converse. We use the term $\sqrt{\frac{FP_t}{(M-1)\cdot FN_t}}$ as the step size (and direction) for updating the indicator size. This step size implies the same factor for adjusting the update interval (for maintaining the budget constraint, as verified by line 6). By this, we effectively distribute the required change of $\frac{FP}{(M-1)\cdot FN}$ equally across the indicator size (governing the behavior of false-positives) and the update interval (governing the behavior of false-negatives). The combined effect brings us closer to having $\frac{FP}{(M-1)\cdot FN} = 1$. The algorithm may adjust the indicator size to ensure that it is within the allowed range, and then adjusts the update interval to satisfy Eq. 1, and avoid violating the budget constraint.

2) Delta-updates, no budget violation or non-minimal indicator size: Lines 8-10 describe a case where either the budget constraint during the segment ending at t was satisfied (i.e., $BW_t \leq B$), or the indicator size can still be reduced (i.e., $|I_t| > |I_{\min}|$). We refer to this case as having the algorithm work in *Mode 2* (shaded blue).

Assume first that there is no budget violation. By the discussion presented in Sec. V-A, when sending delta-updates, and when there is no budget violation, it is advisable to send updates as fast as possible, i.e., using the minimal update interval $u_{\min}$. This approach also implies that there will be no (or very few) false-negatives, since we update the indicator with the shortest allowed interval, keeping it (almost) up to date. For determining the indicator size, we note that the overall bandwidth available per request can be increased by a factor of $\frac{B}{BW_t}$. It follows that we would like to utilize the entire budget to minimize the number of false-positives. If, on the other hand, there is a budget violation, but $|I_t| > |I_{\min}|$, this implies that we may remain in the delta regime, but will be forced to reduce the indicator size to stay within budget.

To determine the ratio by which we should adjust the indicator size, it is instructive to consider the effect of changing the indicator size $|I_t|$ by some factor $\beta > 0$. Such an adjustment implies that every change in the cache will cause β times more/less (depending on whether $\beta > 1$ or not) changed bits in the indicator. Furthermore, adjusting the indicator size by a factor of β implies that specifying each index in the indicator would now requires $\log(\beta \cdot |I_t|)$ bits instead of $\log|I_t|$. It follows that the overall number of bits sent for each change in the cache would increase/decrease by a factor of $\beta \cdot \frac{\log(\beta\cdot|I_t|)}{\log|I_t|}$. We would like the overall change in the number of bits sent to be equal to $\frac{B}{BW_t}$, to match the budget. Formally, we seek a new indicator size $|I|$ s.t:

$$\frac{|I|\log|I|}{|I_t|\log|I_t|} = \frac{B}{BW_t}, \tag{2}$$

where we replace β by $\frac{|I|}{|I_t|}$. The solution to this equation is obtained by using the Lambert W function [42], implying that the new indicator size should be set to $\left\lfloor e^{W\left(\frac{B|I_t|\log|I_t|}{BW_t}\right)}\right\rfloor$.[2] The algorithm then ensures that the best indicator size in this case falls within the allowed range.

3) Delta-updates, budget violation, minimal indicator size: The third and last case is when we use the minimal indicator size, but we still violate the budget. We refer to this case as having the algorithm work in *Mode 3* (shaded green).

In such a case, the only way to ensure feasibility is to increase the update interval, as done in lines 11-13, in order to satisfy Eq. 1. Such a scenario indeed occurs in practice (as we show in Sec. VI), and handling this case ensures that the algorithm can return to the configurations covered by the previous two cases. Mode 3 allows the algorithm to transcend to a considerably different state where we may prefer to send full indicators. Without it, CAB cannot leave the delta regime.

VI. Performance Evaluation

In this section, we present the results of our simulation study.

Setup and system parameters: We focus on scenarios handled by general-purpose caching libraries such as Caffeine [43], Ristretto [44], Guava Cache [45], and the likes. In particular, we use Caffeine for the evaluation of our proposed solution.[3] The cache has a split *get/put* interface where *get* tests the cache, and *put* updates the cache. We extended Caffeine's simulator [43] to simulate the access cost with cache advertisements. In

[2]The Lambert W function is the inverse of the function $f(w) = we^w$. It can be used for solving the equation $x\ln x = a$ by substituting $y = \ln x$, resulting in $ye^y = a$, implying that $x = e^y = e^{W(a)}$.

[3]Caffeine is arguably the most popular Java libraries and is used in tens of large open-source projects such as Cassandra, Corfu, and Infinispan.

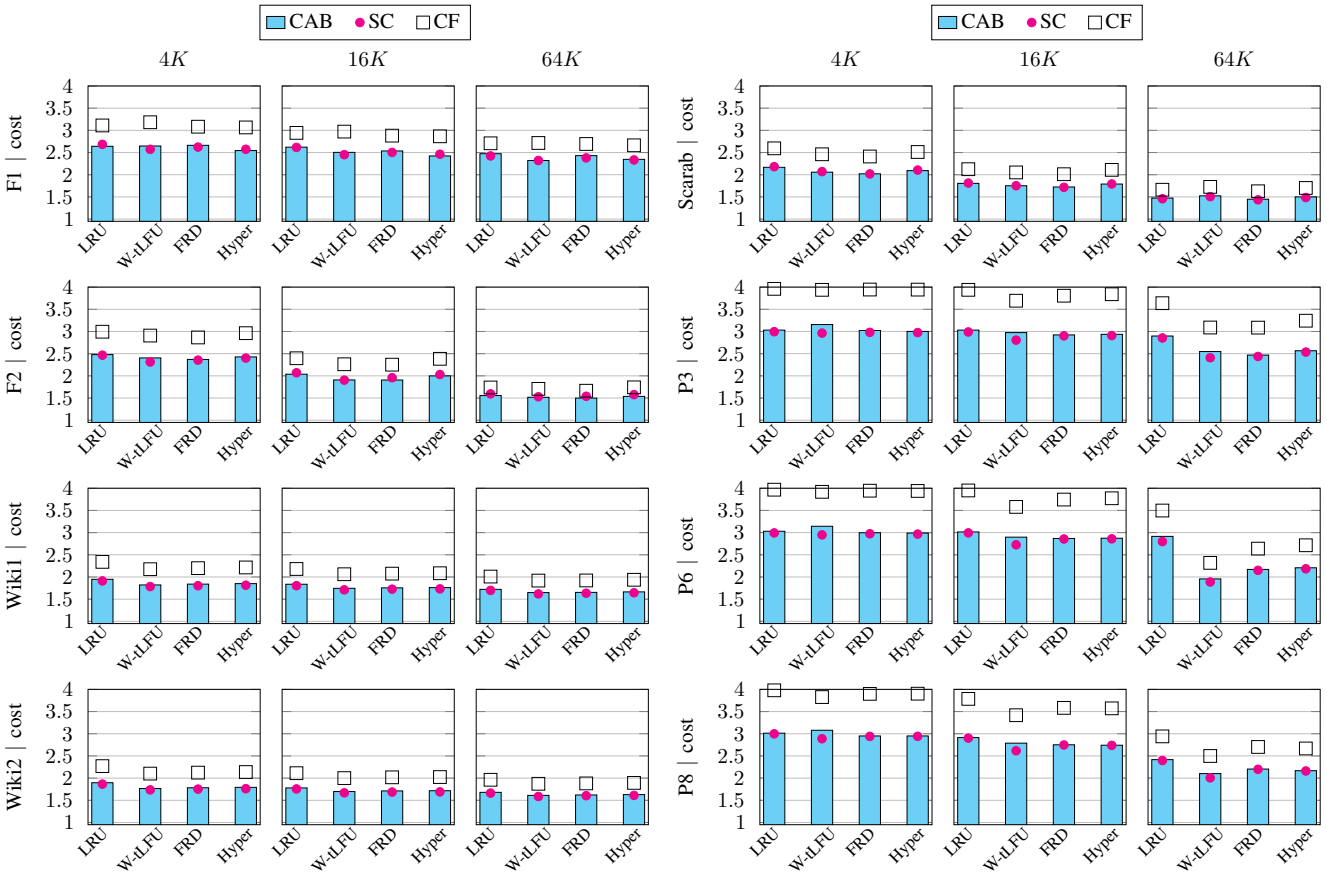

Fig. 3. Access cost for CAB the best feasible static configuration (SC), and always accessing the cache (CF).

our implementation, we issue a *get* request upon a positive indication, and we issue a *put* after handling the request regardless of indications. Our advertisement mechanism uses the Orestes Bloom filters library [46]. The cache maintains a four-bit Counting Bloom Filter (CBF) [12]. However, before sending an update, the cache compresses the CBF to a simple Bloom Filter (BF) [11], where a bit in the BF is set iff the respective counter in the CBF is (strictly) positive. We optimally configure the number of hash functions to minimize the false-positive rate [14].

We set $u_{\min} = 10$ to minimize the transmission overheads. We set $|I_{\min}|$ to $2.5 \cdot C$ which suffices for a false-positive ratio of 30%, and $|I_{\max}|$ to $15 \cdot C$ that implies a false-positive ratio of 0.07% [14]. Our default budget is $B = 20$, which sets $u_{\max}$ to $\frac{|I_{\max}|}{B} = 0.75 \cdot C$. We assume a miss penalty $M = 3$, which is typical for edge computing, where the delay from a cloud processing unit to the memory in the CDN is three times higher than the delay from the cache located at the edge [47].

Benchmarks: The static configuration benchmark (SC) is the best static configuration across a grid of configurations satisfying the budget constraints in all segments; We consider a set of possible indicator sizes $\left\{|I_{\min}| \cdot (1.1)^i | i = 0, \ldots, 18\right\} \subseteq [|I_{\min}|, |I_{\max}|]$,[4] and also the maximal indicator size $|I_{\max}|$. The update intervals considered are taken from the range $\left\{u_{\min} \cdot (1.15)^j | j = 0, \ldots, \frac{\log(u_{\max}/u_{\min})}{\log(1.15)}\right\}$. For example, for cache sizes $4K$, $16K$, and $64K$, the number of possible update intervals considered in the grid are 41, 51, and 61, respectively. We note that SC can only be determined in retrospect. We also evaluate the CacheFirst (CF) policy that always accesses the cache (without indicators) to quantify the access cost reduction from using advertisements.

Traces: We use the following real workloads, which are commonly used when evaluating caching systems: (i) *Scarab*: A trace from Scarab Research, a personalized recommendation system for e-commerce sites [48]. (ii) F1, F2: Traces taken from a financial transaction processing system [49]. (iii) *P3, P6, P8*: Traces of disk accesses in Windows servers [40]. (iv) *Wiki1, Wiki2*: Read requests to Wikipedia pages [50].

Cache policies: We simulated the classic LRU policy, along with three highly competitive policies, including W-TinyLFU [51] (also denoted by W-tLFU)), FRD [52], and Hyperbolic [41]. For completeness, we now very briefly outline these policies.

The Least Recently Used (LRU) policy evicts the least recently accessed item. It assumes that recently accessed items would be accessed again. The W-tLFU policy combines LRU with a frequency-based cache. Items are only admitted to a frequency-based cache if they are more frequent in a long

[4] $|I_{\min}| \cdot (1.1)^i \leq |I_{\max}|$ implies in this case that $i \leq \lfloor \frac{\log 6}{\log 1.1} \rfloor = 18$.

history, which is represented as a CBF for space efficiency. FRD varies the time it retains admitted items according to their past access pattern. First-timers are admitted for a short duration, while previously encountered items are admitted for a longer duration. FRD uses extensive metadata about past accesses to distinguish first-timers from recurring items. Finally, Hyperbolic caching is an adaptive cache policy that changes its eviction policy according to the workload.

A. Competitive Evaluation:

CAB across workloads and policies: Our first experiment compares the access cost for the four cache policies when varying the cache size and the workload. Figure 3 shows the results of these experiments. First, observe that increasing the cache size reduces the access costs, as expected by cache policies. Further, observe that for most traces, the differences between the cache policies are not very large, and are smaller than the differences between the CF and SC policy. This implies that the potential benefit from advertising cache content may be higher than the benefit from changing the cache policy. Finally, observe that the performance of CAB is very similar to that of the SC benchmark. In some cases, (e.g., F1 4K LRU) CAB is even slightly better than the best static configuration. Such a result implies that conditions change during the trace and that CAB manages to adjust itself according to these changes, thus reducing cost. In other cases, CAB is slightly worse than the best configuration, but the difference is always small. CAB operates in *real time*, without prior knowledge of the system configuration, or the workload (other than knowing the cache size, the budget, and the bounds on the indicator sizes).

CAB across budgets and policies: We now repeat the experiment for varying budgets. Fig. **??** illustrates these results. As expected, in all policies CAB and SC improve when the budget increases. As in our previous experiment, CAB matches the performance of SC regardless of budget. However, for W-tLFU CAB is not as good as SC for a small budget (10). The reason for this is that W-tLFU contains a very small and rapidly-changing Window cache, which forces CAB into short update intervals, which are bad for the larger and less dynamic Main cache (consuming 99% of the cache space). Alternatively, notice that CAB is slightly better than SC for a budget of 40 across all policies, implying that it manages to adapt to the changes within the trace.

B. CAB Under the Hood:

We now turn to highlight the performance and behavior of CAB in dynamic settings, where the workload changes. In this experiment, we run several traces one after another. Since each workload's characteristics are slightly different, such an experiment allows us to follow the dynamic change of configuration performed by CAB and the system behavior as it interacts with these changes.

Figure 4 shows various aspects of the execution of CAB for a combination of traces F1 (dark shaded area) and F2 (light shaded area), concatenated as F1→F2→F1→F2. Note that both F1 and F2 are typical to the same application (financial

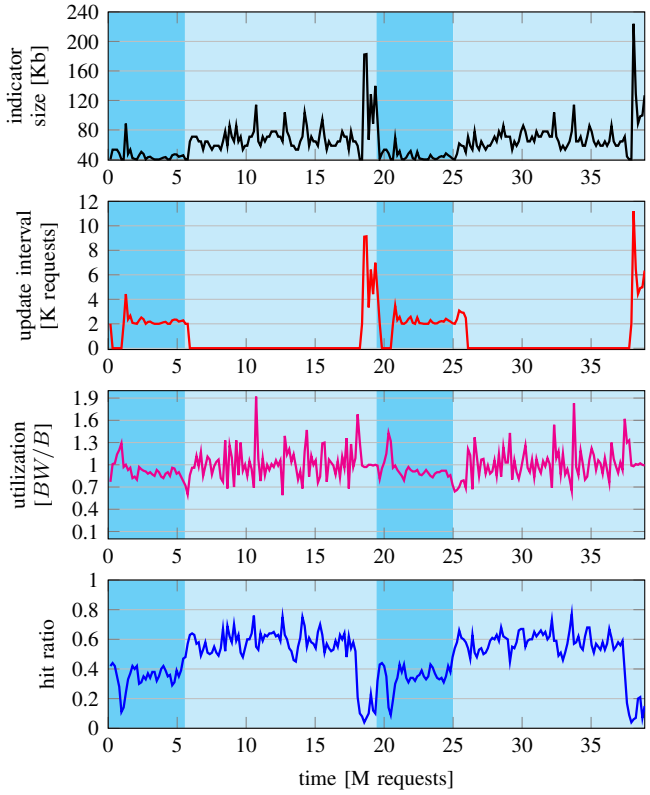

Fig. 4. The dynamics of CAB on a concatenated trace F1→F2→F1→F2. F1 requests are dark shaded, and F2 requests are light shaded. The cache size is $16K$, the policy is LRU, and the budget is 20 bits/request.

transactions), and therefore such a concatenated workload may indeed happen in practice. The total request count is ∼40M. The experiment is performed using $C = 16$K, the LRU policy, $M = 3$, and a budget of 20 bits/req. As in earlier experiments, we take $u_{\min} = 10$, and $\alpha = 10$. This implies $T = 160$K, resulting in ∼250 configuration segments during the entire simulation. Also, we set $|I_{\min}| = 2.5 \cdot C = 40$K, and $|I_{\max}| = 15 \cdot C = 240$K.

The top subfigure shows the evolution of the indicator size. The second topmost subfigure shows the evolution of the update interval. These two figures provide a glimpse into the evolution of the configurations used by CAB. The bottom subfigure shows the evolution of the hit ratio of the cache. We stress that the hit ratio captures the ratio of requests that are actually found in the cache, and is therefore an artifact of the settings (workload, cache size, and cache policy). The advertisement policy doesn't impact the hit ratio. However, CAB's advertisement policy implicitly *reacts* to the hit ratio. The second bottom-most subfigure shows the bandwidth utilization (normalized to the bandwidth budget). Notice that the utilization is calculated before network policing; hence CAB may try to exceed the bandwidth (but the network policing prevents that).

These figures show the effect of the algorithm's choices and the properties of the workload. We now turn to explain and describe the algorithm's performance along time t (measured by the request counts).

- $t \in [0, 1\text{M}]$: Soon after the beginning, CAB identifies that

Itamar: In the ver
tiny bug in the ca
run all the simula
results. However,
may observe the
CABinfocom21su
this project). I ch
that Fig. accordin

the cache uses delta-updates, and thus sets $u_t = u_{\min}$, while adjusting the indicator size so as to comply with the budget constraint (Mode 2).

- $t \sim 1$M: The hit ratio (bottom subfigure) sharply decreases. This translates to a substantial change in the cached content, resulting in much larger delta updates that violate the budget constraint (second-bottom subfigure). Notice that CAB cannot exceed the budget, as its advertisements are dropped once the budget is exhausted (leading to additional errors). The algorithm fails to comply with the budget constraint with a minimal update interval, even when shrinking the indicator size to $|I_{\min}|$. Hence, CAB increases the update interval, to ensure adhering to the budget restriction (Mode 3).
- $t \in [1\text{M}, 5.5\text{M}]$: CAB constantly works in Mode 1, mostly sending full indicators (as it is cheaper than sending the mere changes). During this time, the algorithm constantly satisfies the budget constraint, and merely makes small adjustments to the indicator size, and the update interval, to optimize its usage of bandwidth while balancing the extra costs incurred by false-positives and false-negatives.
- $t \in [5.5\text{M}, 18\text{M}]$: the F1 trace ends, and the F2 trace begins. The hit ratio significantly increases, thus allowing for using a much larger indicator, and more frequent updates. The algorithm hence switches to Mode 2, and persistently uses the minimal update interval. CAB occasionally violates the budget constraint, and then shrinks the indicator again, to stay within limits.
- $t \sim 18$M: A sudden drop in the hit ratio does not allow the algorithm to keep a minimal update interval anymore, even when the indicator size is $|I_{\min}|$. Hence, CAB switches to mode 3, and significantly increases the update interval to satisfy the budget constraint.
- $t \in [18\text{M}, 20\text{M}]$: The algorithm works in Mode 1, constantly sending full indicators, and adjusting the indicator size and the update interval to remain within budget. This operation occurs alongside the general increase in hit ratio, allowing the algorithm to settle for smaller indicator size and update intervals.
- $t \in [20\text{M}, 25.5\text{M}]$: The algorithm again behaves as it did when initially handling F1.
- $t \in [25.5\text{M}, 40\text{M}]$: The F2 trace arrives, and the algorithm exhibits similar behavior when transitioning again to handling F2. However, a closer look shows that the indicator size and the update interval slightly differ from those chosen by the algorithm in the previous run of F2, which occurred in the interval $t \in [5.5\text{M}, 18\text{M}]$. These changes further exemplify how the algorithm adapts even to minor changes stemmed from differences in the cache's content at the beginning of the different runs of F2.

VII. Discussion and Conclusions

Many systems use cache advertisements, which highlight the need for *efficient* cache advertisement strategies. Yet, prior to our work, the literature lacked a rigorous method to configure the advertisement strategy. Therefore, system designers turn to design decisions based on rules-of-thumb and ad-hoc benchmarks of typical workloads. Additionally, these approaches are mostly limited to selecting a static advertisement policy, which means that changes in system and workload parameters might degrade the quality of their choice.

Our work surveys the possible modes of operations that an advertisement policy can utilize. We empirically show that there is no one-size-fits-all policy and that advertisement policies' performance depends on the cache policy, the cache size, and the workload. Worse yet, static policies are ill-suited for adaptive cache policies that change their behavior during run-time [39]–[41].

We designed the novel CAB algorithm that adjusts the advertisement policy according to the current conditions. CAB reaches its decision by monitoring its bandwidth footprint, false-positive, and false-negative errors. It is indifferent to the cache size, the workload, and the cache policy (beyond their indirect effect on the false-positive and false-negative rates).

We performed an extensive evaluation that uses eight real workloads and tested the classic LRU policy and three other leading cache management policies. We performed our work under a strict network model that drops messages if transmitting them would violate the bandwidth budget. Under these conditions, CAB exhibits an overall cost comparable (and sometimes superior) to the best static advertisement policy for all cache sizes, workloads, and cache policies. CAB is a game-changer as developers no longer need to optimize their advertisement policy manually. Instead, they can use CAB to optimize the advertisement strategy, shorten development time, and attain good performance in a variety of scenarios and system configurations. More so, CAB solves problems that were encountered by many works [4], [10], [18], [19], [31]–[36], and solved only in an ad-hoc manner.

Next, we run multiple traces one after another and show that CAB successfully adapts to the changes in the workload, varying its advertisement strategy according to the conditions, and effectively transitioning between the delta and full-indicator updates regimes. To the best of our knowledge, CAB is the first to combine these options seamlessly. We note that CAB changes its transmission policy quite often during the system's lifetime, which incurs computation costs at the cache. E.g., we need to prepare a new indicator every time we change the indicator size. While we do not evaluate the CPU usage of the cache, we sized the reconfiguration intervals to be ten times the cache size. Thus, the amortized cost of computing a new indicator is one indicator operation per 10 cache accesses. We believe that such a configuration makes the additional overheads manageable (if not negligible) since current indicators, such as Bloom filter implementations [43], [46], reach over 20 million ops per second on a single thread and are embarrassingly parallel.

Another takeaway from our research is that false-negatives cannot be neglected when advertising the cached content. Thus, the works that neglect them [1], [27] limit their analysis to a single mode of operations and are therefore incomplete. Looking into the future, we plan to develop access strategies

that cope well with false-negatives, and then use such strategies alongside CAB on a distributed network. Such a work may borrow some ideas from [27], [31].

REFERENCES

[1] O. Rottenstreich and I. Keslassy, "The bloom paradox: When not to use a bloom filter," *IEEE/ACM Trans. Netw.*, vol. 23, no. 3, pp. 703–716, 2015.

[2] X. Guo, T. Wang, and S. Wang, "Joint optimization of caching and routing strategies in content delivery networks: A big data case," in *IEEE ICC*, 2019.

[3] B. Maggs and R. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Comp. Comm. Rev.*, vol. 45, no. 3, pp. 52–66, 2015.

[4] I.-W. Ting and Y.-K. Chang, "Improved group-based cooperative caching scheme for mobile ad hoc networks," *J. Parallel. and Distrib. Comp.*, vol. 73, no. 5, pp. 595–607, 2013.

[5] T. Le, Y. Lu, and M. Gerla, "Social caching and content retrieval in disruption tolerant networks (dtns)," in *2015 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2015, pp. 905–910.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," *SIGPLAN Not.*, vol. 35, no. 11, pp. 190–201, 2000.

[7] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *NSDI*, 2017, pp. 483–498.

[8] M. Bilal and S. G. Kang, "A cache management scheme for efficient content eviction and replication in cache networks," *IEEE Access*, vol. 5, pp. 1692–1701, 2017.

[9] I. Psaras, W. K. Chai, and G. Pavlou, "Probabilistic in-network caching for information-centric networks," in *ICN*, 2012, pp. 55–60.

[10] 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, 2000.

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

[12] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *ESA*, 2006, pp. 684–695.

[13] G. Einziger and R. Friedman, "Tinyset: An access efficient self adjusting bloom filter construction," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2295–2307, 2017.

[14] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Comm. Surv. & Tut.*, vol. 14, no. 1, pp. 131–155, 2012.

[15] L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Comm. Surv. & Tut.*, vol. 21, no. 2, pp. 1912–1949, 2018.

[16] G. Einziger and R. Friedman, "Counting with tinytable: Every bit counts!" in *ICDCN*, 2016, p. 27.

[17] Y. Kanizo, D. Hay, and I. Keslassy, "Access-efficient balanced bloom filters," *Comput. Comm.*, vol. 36, no. 4, pp. 373–385, 2013.

[18] W. Shi and Y. Mao, "Performance evaluation of peer-to-peer web caching systems," *J. of Sys. and Soft.*, vol. 79, no. 5, pp. 714–726, 2006.

[19] M. Tortelli, L. A. Grieco, and G. Boggia, "CCN forwarding engine based on bloom filters," in *CFI*, 2012, pp. 13–14.

[20] Squid Cache, "Squid-cache wiki." [Online]. Available: https://wiki.squid-cache.org/SquidFaq/CacheDigests#Would_it_be_possible_to_stagger_the_timings_when_cache_digests_are_retrieved_from_peers.3F

[21] ——, "Squid digest spec, v5." [Online]. Available: http://www.squid-cache.org/CacheDigest/cache-digest-v5.txt

[22] X. Guo *et al.*, "Joint optimization of caching and routing strategies in content delivery networks: A big data case," in *IEEE ICC*, 2019.

[23] R. Hou, L. Zhang, T. Wu, T. Mao, and J. Luo, "Bloom-filter-based request node collaboration caching for named data networking," *Clust. Comp.*, vol. 22, no. 3, pp. 6681–6692, 2019.

[24] M. Zhang, H. Luo, and H. Zhang, "A survey of caching mechanisms in information-centric networking," *IEEE Comm. Surv. & Tut.*, vol. 17, no. 3, pp. 1473–1499, 2015.

[25] G. Zhang, Y. Li, and T. Lin, "Caching in information centric networking: A survey," *Comp. Net.*, vol. 57, no. 16, pp. 3128–3141, 2013.

[26] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Trans on Knowl. and Data Eng.*, vol. 22, no. 1, pp. 120–133, 2009.

[27] I. Cohen, G. Einziger, R. Friedman, and G. Scalosub, "Access strategies for network caching," in *IEEE INFOCOM*, 2019, pp. 28–36.

[28] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Trans. Netw.*, vol. 10, no. 5, pp. 604–612, 2002.

[29] S. Z. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, "Bloom filter with a false positive free zone," in *INFOCOM*, 2018, pp. 1412–1420.

[30] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "HBA: Distributed metadata management for large cluster-based storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 147, pp. 204–220, 2018.

[31] Y. Zhu and H. Jiang, "False rate analysis of bloom filter replicas in distributed systems," in *ICPP*, 2006, pp. 255–262.

[32] Hong Tang and Tao Yang, "An efficient data location protocol for self.organizing storage clusters," in *ACM/IEEE Supercomputing*, 2003, pp. 53–53.

[33] J. Ledlie, L. Serban, and D. Toncheva, "Scaling filename queries in a large-scale distributed file system," 2002.

[34] M. Ripeanu and I. Foster, "A decentralized, adaptive, replica location service," 01 2002.

[35] P.-H. Hsiao, "Geographical region summary service for geographical routing," in *Mobihoc*, 2001, p. 263–266.

[36] H. Cai and J. Wang, "Foreseer: A novel, locality-aware peer-to-peer system architecture for keyword searches," in *ACM/IFIP/USENIX Middleware*, 2004, p. 38–58.

[37] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's time to revisit LRU vs. FIFO," in *HotStorage*, 2020.

[38] A. Rousskov and D. Wessels, "Cache digests," *Comp. Net. and ISDN Sys.*, vol. 30, no. 22-23, pp. 2155–2168, 1998.

[39] G. Einziger, O. Eytan, R. Friedman, and B. Manes, "Adaptive software cache management," in *ACM Middleware*, 2018, pp. 94–106.

[40] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *Fast*, no. 2003, 2003.

[41] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *USENIX ATC*, 2017, pp. 499–511.

[42] R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth, "On the Lambert W function," *Adv. Comput. Math.*, vol. 5, no. 1, pp. 329–359, 1996.

[43] B. Manes, "Caffeine: A high performance caching library for java." [Online]. Available: https://github.com/ben-manes/caffeine

[44] Dgraph Labs, Inc., "Ristretto: A high performance memory-bound go cache." [Online]. Available: https://github.com/dgraph-io/ristretto

[45] Google, "Guava: Google core libraries for java." [Online]. Available: https://github.com/google/guava

[46] Baqend GmbH, "Orestes: Bloom filter library for java." [Online]. Available: https://github.com/Baqend/Orestes-Bloomfilter

[47] T. X. Tran and D. Pompili, "Octopus: A cooperative hierarchical caching strategy for cloud radio access networks," in *IEEE MASS*, 2016, pp. 154–162.

[48] "Caffeine's simulator cache traces." [Online]. Available: https://github.com/ben-manes/caffeine/tree/master/simulator/src/main/resources/com/github/benmanes/caffeine/cache/simulator/parser

[49] M. Liberatore and P. Shenoy, "Umass trace repository," 2016. [Online]. Available: http://traces.cs.umass.edu/

[50] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Comp. Net.*, vol. 53, no. 11, pp. 1830–1845, 2009.

[51] G. Einziger, R. Friedman, and B. Manes, "Tinylfu: A highly efficient cache admission policy," *TOS*, vol. 13, no. 4, pp. 35:1–35:31, 2017.

[52] S. Park and C. Park, "FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance," in *MSST*, 2017.