Optimized Dynamic Cache Instantiation

Niklas Carlsson
Linköping University, Sweden

Derek Eager
University of Saskatchewan, Canada

***Abstract*—By caching content at geographically distributed servers, content delivery applications can achieve scalability and reduce wide-area network traffic. However, each deployed cache has an associated cost. As the request rate from a region varies (e.g., according to a daily cycle), there may be periods when the request rate is high enough to justify this cost, and other periods when it is not. Cloud computing offers a solution to problems of this kind, by supporting the dynamic allocation and release of resources. In this paper, we analyze the potential benefits from dynamically instantiating caches using resources from cloud service providers. We develop novel analytic caching models that accommodate time-varying request rates, transient behavior as a cache fills following instantiation, and selective cache insertion policies. Using these models, within the context of a simple cost model, we then develop bounds and compare policies with optimized parameter selections to obtain insights into key cost/performance tradeoffs. We find that dynamic cache instantiation can provide substantial cost reductions, that potential reductions strongly dependent on the object popularity skew, and that selective cache insertion can be even more beneficial in this context than with conventional edge caches.**

I. Introduction

The performance and scalability of content delivery systems benefit significantly from geographically distributed caches, and so these have been the subject of much research (Section VII). However, despite the emergence of distributed, regional, and edge cloud computing offering a completely new service paradigm – on-demand caching – surprisingly few works have taken into account on-demand cache provisioning [1]–[4], and, to our knowledge, no prior work has considered, rigorously modelled, and analyzed the problem of when to instantiate and release caches in such environments.

Content request rates typically differ between locations and vary over time (e.g., according to a relatively predictable daily cycle [5]). In systems where the service provider pays on an on-demand basis, we would, ideally, like to incur the cost of a local cache only when the local request rate is high enough to justify this cost.

In this paper, we take a first look at the potential benefits from dynamically instantiating and releasing caches. In particular, we develop novel analytic models of cache performance that accommodate the important challenges of taking into account (i) arbitrarily time-varying request rates and (ii) periods of transient behavior when a cache fills following instantiation, and apply these models within the context of a simple cost model to study cost/performance tradeoffs.

First, to accommodate time-varying request rates and periods of transient behavior, we develop a modelling approach based on what we term here "request count window" (RCW) caches. Objects are evicted from an RCW cache if not requested over a window consisting of the most recent L requests, where L is a parameter of the system. As we show here empirically, similarly as with "Time-to-Live" (TTL) caches [6]–[10] in scenarios with fixed request rates, the performance of an RCW cache closely approximates the performance of an LRU cache when the size of the window (for an RCW cache, measured in number of requests) is set such that the average occupancy equals the LRU cache size.

Second, we carry out analytic analyses of RCW caches for both indiscriminate *Cache on* 1^{st} *request* and selective *Cache on* k^{th} *request* cache insertion policies. Explicit, exact expressions are derived for key cache performance metrics under the independent reference model, including (i) the hit and insertion rates for permanently allocated caches, and (ii) the average rates over the transient period during which a newly instantiated cache is filling. Selective *Cache on* k^{th} *request* cache insertion policies are of particular interest here, since dynamically instantiated caches may be relatively small, and therefore cache pollution may be a particularly important concern. Our RCW analysis makes no assumptions regarding inter-request time distributions or request rate variations, ensuring that our RCW results (in contrast to prior TTL approximations [6]–[10]) can be used to approximate LRU cache performance under highly time-varying request volumes. In general, for time-varying workloads, the concept of RCW caches provides a more natural choice than TTL caches when approximating fixed-capacity LRU caches.

Third, in addition to the insertion policy, important design choices in a dynamic cache instantiation system include the cache size and the duration of the instantiation interval. We develop optimization models for these parameters for both *Cache on* 1^{st} *request* and *Cache on* k^{th} *request*. We also develop bounds on the best potentially achievable cost/performance tradeoffs, assessing how much room for improvement there may be through use of more complex caching policies.

Finally, we apply our analyses to obtain insights into dynamic cache instantiation and explore key system tradeoffs. We find that dynamic cache instantiation using *Cache on* k^{th} *request* is a promising approach for content delivery applications. Specifically: (1) Dynamic cache instantiation has the potential to provide significant cost reductions, sometimes more than halving the costs of (optimized) baselines that either use a cache or not, depending on which results in a lower cost. (2) The cost reductions are strongly dependent on the object popularity skew. When there is high skew, dynamic instantiation can work particularly well since a newly instantiated cache is quickly populated with frequently requested items that will capture a substantial fraction of the requests. (3) *Cache on* k^{th} *request* cache insertion policies can be even

ISBN 978-3-903176-28-7

more beneficial in this context than with conventional edge caches. When there is high popularity skew, there is likely only modest room for improvement in cost/performance through use of more complex cache insertion and replacement policies.

Roadmap: Section II describes our workload and system assumptions, the caching policies considered, and the metrics of interest. Section III analyzes RCW caches for the baseline case without use of dynamic instantiation. Section IV analyzes the transient period as an RCW cache fills. Optimization models and performance results for dynamic instantiation are presented in Sections V and VI. Section VII describes related work, before Section VIII concludes the paper.

II. System Description and Metrics

Workload Assumptions: We focus on a single cache location serving a sub-population of clients [11]. For this cache location, we consider a time period of duration T (e.g., one day), over which the total (aggregated over all objects) content request rate $\lambda(t)$ varies. We assume that these variations are predictable, and so for any desired cache instantiation duration $D < T$, it would be possible to identify in advance the interval of duration D with the highest average request rate over all intervals of duration D within the time period.

Short-term temporal locality, non-stationary object popularities, and high rates of new content creation make dynamic cache instantiation potentially more promising, since they reduce the value of old cache contents. Here, we provide a conservative estimate of the benefits of dynamic cache instantiation, assuming a fixed set of objects with stationary object popularities, and with requests following the independent reference model. We denote the number of objects by N, and index the objects such that $p_i \geq p_{i+1}$ for $1 \leq i < N$, where p_i denotes the probability that a request is for object i.

Cache Policies: We model a "request rate window" (RCW) cache from which objects are evicted if not requested over a window consisting of the most recent L requests, where L is a system parameter. As we empirically demonstrate, the performance of an RCW cache closely approximates the performance of an LRU cache when the value of L is set such that the average occupancy equals the size of the LRU cache.

Both indiscriminate, *Cache on* 1^{st} *request*, and selective *Cache on* k^{th} *request* cache insertion policies are considered. For *Cache on* k^{th} *request* with k>1, we assume that the system maintains some state information regarding uncached objects that have been requested at least once over a window consisting of the most recent W requests, where W is a policy parameter. Specifically, for each such "caching candidate", a count of how many requests are made for the object while it is a caching candidate is maintained. When a request is received for an uncached object that was not already a caching candidate, the object becomes a caching candidate with count initialized to one. Should this count reach k, the object is cached. Should no request be made to the object for W requests, the object is removed as a caching candidate.

For the dynamic instantiation, we assume that the cloud provider returns an empty cache when (re)instantiated. This does not require us to make any assumption regarding the type of cache (e.g., in memory vs disk-based storage, type of VMs, etc.). However, we note that the cloud provider that is not able to rent out the resources to serve other workloads may decide to only shut down disks/memory to save energy and in some of these cases therefore potentially could return part of the cache in its original state. For such a case, our analysis provides a pessimistic performance bound.

Metrics and Cost Assumptions: The metrics of primary interest are the expected *fraction of requests served locally* from cache (over the entire time period), and the *cache cost*. With dynamic cache instantiation, the first of these two metrics is given by $\frac{\bar{H}_{t_a:t_d}\int_{t_a}^{t_d}\lambda(t)\mathrm{d}t}{\int_0^T\lambda(t)\mathrm{d}t}$, where t_a denotes the time at which the cache is allocated, t_d the time at which it is deallocated, and $\bar{H}_{t_a:t_d}$ the average hit rate over this interval. Note that the hit rate (probability) will vary over the interval, with the hit rate immediately after instantiation being zero (empty cache).

Implementations of dynamic cache instantiation could use a variety of technologies. One option would be to use dynamic allocation of a virtual machine, with main memory used for the cache. We assume here a simple cost model where the cost per unit time of a cache of capacity C objects is proportional to $C+b$, where the constant b captures the portion of the cost that is independent of cache size. The total cost over the period T is then proportional to $(t_d-t_a)(C+b)$. More complex cost models could be easily accommodated; the only issue being the computational cost of evaluating the cost function when solving our optimization models.

In addition to the above metrics, when analyzing RCW caches we evaluate the hit rate H, the cache insertion rate I (fraction of requests that result in object insertions into the cache) as well as the insertion fraction $I/(I+H)$, and the average number A of objects in the cache. The insertion fraction is an important measure of overhead; cache insertions consume node resources, but do not yield any benefit unless there are subsequent resulting cache hits. The average number of objects in the cache is used to match the cache capacity C of an LRU cache with similar performance.

III. RCW Cache Analysis

We consider first a permanently allocated RCW cache.

A. Exact "Always on" RCW Analysis

Cache on 1^{st} **Request:** The probability that a request for object i finds it in the cache is given by $1-(1-p_i)^L$, since object i will be in the cache if and only if at least one of the most recent L requests was to object i. The average number A of objects in the cache, as seen by a random request, the insertion rate I, and the hit rate H, are therefore given by

$$A=N-\sum_{i=1}^{N}(1-p_i)^L,\ I=\sum_{i=1}^{N}p_i(1-p_i)^L,\ H=1-\sum_{i=1}^{N}p_i(1-p_i)^L. \quad (1)$$

Cache on 2^{nd} **Request:** The expected value $E[\Theta_i]$ of the object i duration in the cache, measured in number of requests, is given by the average number of requests until there is a

sequence of L requests in a row that do not include a request for object i. Since requests follow the independent reference model and the probability of a request for object i is p_i, this is the same as the average number of flips of a biased coin that are required to get L heads in a row, with the probability of a head equal to $1-p_i$:

$$E[\Theta_i] = \sum_{r=1}^{L} \frac{1}{(1-p_i)^r} = \frac{1-(1-p_i)^L}{p_i(1-p_i)^L}. \quad (2)$$

With *Cache on 2^{nd} request*, the expected value $E[\Delta_i]$ of the object i duration out of the cache, measured in number of requests, satisfies the following equation:

$$\begin{aligned} E[\Delta_i] &= 1/p_i + (1-p_i)^W(W + E[\Delta_i]) \\ &\quad +(1-(1-p_i)^W)\left(1/p_i - \frac{(1-p_i)^W W}{1-(1-p_i)^W}\right) \quad (3) \end{aligned}$$

Here, the first term $(1/p_i)$ gives the expected number of requests until the first request for object i following its removal from the cache. The second term gives the expected number of additional requests until object i is added to the cache, conditional on the first request not being followed by another request within the window W, multiplied by the probability of this condition. The third term gives the expected number of additional requests until object i is added to the cache, conditional on the first request being followed by another request within the window W, multiplied by the probability of this condition. Solving for $E[\Delta_i]$ yields:

$$E[\Delta_i] = \frac{2-(1-p_i)^W}{p_i(1-(1-p_i)^W)}. \quad (4)$$

Expressions for A, H, and I now follow directly from (2) and (4), and $A = \sum_i \frac{E[\Theta_i]}{E[\Theta_i]+E[\Delta_i]}$, $H = \sum_i p_i \frac{E[\Theta_i]}{E[\Theta_i]+E[\Delta_i]}$, and $I = \sum_i \frac{1}{E[\Theta_i]+E[\Delta_i]}$.

Cache on kth Request: The analysis for general $k \geq 2$ differs from that for *Cache on 2^{nd} request* with respect to $E[\Delta_i]$, the expected value of the object i duration out of the cache, measured in number of requests. Denoting $E[\Delta_i]$ for *Cache on k^{th} request* by $E^k[\Delta_i]$, $E^k[\Delta_i]$ ($k \geq 2$) can be expressed as a function of $E^{k-1}[\Delta_i]$ as follows:

$$E^k[\Delta_i] = \frac{E^{k-1}[\Delta_i] + (1-p_i)^W W + (1-(1-p_i)^W)\left(\frac{1}{p_i} - \frac{(1-p_i)^W W}{1-(1-p_i)^W}\right)}{1-(1-p_i)^W}, \quad (5)$$

with $E^1[\Delta_i]$ defined as $1/p_i$. The numerator of the right-hand side of this equation gives the expected number of requests from when an object is removed from cache or removed as a caching candidate, until it next exits from the state in which it is a caching candidate with a count of $k-1$ (either owing to being cached because of a request occurring within the window W, or removed as a caching candidate if no such request occurs). The denominator is the probability of being cached when exiting from the state in which it is a caching candidate with a count of $k-1$, and therefore the inverse of the denominator gives the expected number of times the object will enter this state until it is finally cached. Simplifying yields

$$E^k[\Delta_i] = \frac{E^{k-1}[\Delta_i]}{1-(1-p_i)^W} + \frac{1}{p_i}, \quad (6)$$

implying

$$E^k[\Delta_i] = \frac{1}{p_i}\left(\frac{1-(1-(1-p_i)^W)^k}{(1-p_i)^W(1-(1-p_i)^W)^{k-1}}\right). \quad (7)$$

Expressions for A, H, and I now follow directly from (2) and (7), once A, H, and I are expressed in terms of $E^k[\Delta_i]$ and $E[\Theta_i]$ as in the analysis for *Cache on 2^{nd} request*. For W=L, these expressions reduce to:

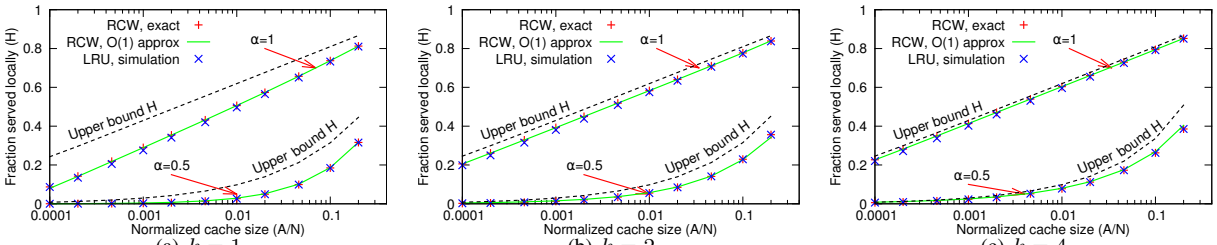
$$\begin{aligned} A = \sum_{i=1}^{N}(1-(1-p_i)^L)^k, \quad H = \sum_{i=1}^{N} p_i(1-(1-p_i)^L)^k, \\ I = \sum_{i=1}^{N} p_i(1-p_i)^L(1-(1-p_i)^L)^{k-1}. \quad (8) \end{aligned}$$

B. Validation and Performance Results

Figure 1 compares our exact RCW cache hit rate results (red '+' markers), using W=L, with the results from simulations of corresponding fixed-capacity LRU caches (blue '$\times$' markers), for N=100,000 objects, different k, and over a large range of cache sizes (A/N=0.0001 corresponds here to A=10). Also shown in the figure are the *upper bound* hit rate (dashed black lines), corresponding to when the cache is kept filled with the $\lfloor C \rfloor$ most popular objects, and $\mathcal{O}(1)$ *approximations* (green lines) that we have derived using Taylor series expansions (see appendix) for the special cases of Zipf distributions with α=1 and α=0.5. We note that popularity skew typically is intermediate between these two cases.

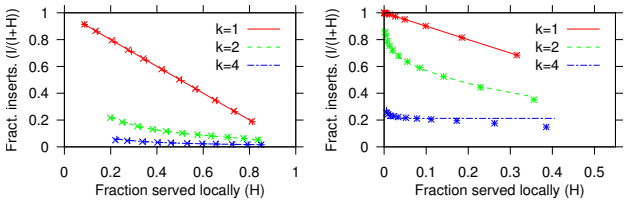
For the simulations, we set the LRU cache size C to equal A. To match use of W=L in the case of the RCW caches, we assume an implementation of LRU with *Cache on k^{th} request* in which, when the cache is full (as it is in steady state), W is dynamically set to the number of requests since the "least recently requested" object currently in the cache was last requested. Similar to an RCW cache with W=L, this choice ensures that an object remains a "caching candidate" as long as it is requested at least as recently as the least recently requested object in the cache. For the simulation results reported here and in subsequent sections, each simulation was run for six million requests, with the statistics for the initial two million requests removed from the measurements.

The following observations stand out. First, for all cases (including larger k), the exact RCW results closely match the LRU simulation results. This shows that our RCW analysis can be used as an effective method to approximate the performance of an LRU cache. Second, the $\mathcal{O}(1)$ approximations are accurate both for α=1 (caching effective) and α=0.5 (caching largely ineffective). For the insertion fraction $I/(I+H)$ (not shown), the $\mathcal{O}(1)$ approximations diverge somewhat more, but errors remain within 10% for all cases except for (i) small cache sizes ($A/N < 0.001$) when k=4 and α=0.5, and (ii) large cache sizes ($A/N > 0.1$) when k=4 and α=0.5. For α=1, the insertion fraction errors remain within 5%, and for k=1 (regardless of α) the errors are within 0.3%. Third, the gap in hit rate between the policies and the upper bound is substantial with k=1 (regular LRU), narrows with k=2, and is almost eliminated with k=4, leaving little room for further hit rate improvements.



(a) $k = 1$ (b) $k = 2$ (c) $k = 4$

Fig. 1. Performance of *Cache on* k^{th} *request* (N=100, 000); dashed lines show the hit rate when the cache is kept filled with the $\lfloor C \rfloor$ most popular objects.



(a) $\alpha = 1$ (b) $\alpha = 0.5$

Fig. 2. Tradeoff curves for *Cache on* k^{th} *request* policies. Here, N=10^5 and we use markers: "RCW, exact" (+), "RCW, approx" (line), and LRU ($\times$).

Although increasing k beyond k=2 yields only small additional improvements in hit rate, substantial improvements in the insertion fraction $I/(I+H)$ continue as k is increased. To illustrate this, Figure 2 shows the tradeoff between hit rate (on x-axis) and insertion fraction (y-axis) for different k. Note that with selective insertion policies (i.e., larger k), the same hit rate can be achieved with a much lower insertion fraction.

IV. Dynamic Instantiation Analysis

A. Cache on 1st Request

Consider now the case where the cache is allocated for only a portion of the time period, and is initially empty when instantiated. With *Cache on* 1^{st} *request*, after the first L requests following instantiation, the cache will have the occupancy probabilities derived earlier for the "always-on" case in Section III, and so for requests following the first L requests the analysis in Section III can be used. The average insertion rate over the first L requests (the transient period) is given by the expression for the average number A of objects in cache from (1), divided by L. Denoting the average hit rate during the transient period by $\bar{H}_{\text{transient}}$, this gives:

$$\bar{H}_{\text{transient}} = 1 - \frac{A}{L} = 1 - \frac{N - \sum_{i=1}^{N}(1-p_i)^L}{L}, \qquad (9)$$

and from equation (1) for H, assuming that $\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t \geq L$,

$$\bar{H}_{t_a:t_d} = \frac{L\bar{H}_{\text{transient}} + \left(\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t - L\right)\left(1 - \sum_{i=1}^{N} p_i(1-p_i)^L\right)}{\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t}. \qquad (10)$$

Finally, for *Cache on* 1^{st} *request*, $\bar{I}_{\text{transient}}$ and $\bar{I}_{t_a:t_d}$ are given simply by $1-\bar{H}_{\text{transient}}$ and $1-\bar{H}_{t_a:t_d}$.

B. Cache on kth Request ($k \geq 2$)

As described in Section II, *Cache on* k^{th} *request* requires maintenance of state information regarding "caching candidates" and all currently cached objects. We assume that when a cache using *Cache on* k^{th} *request* is deallocated, the state information of both types is transferred to the upstream system to which requests will now be directed. The upstream system maintains and updates this state information when receiving requests that the cache would have received had it been allocated, and transfers it back when the cache is instantiated again. Therefore, although the cache is initially empty when instantiated, it can use the acquired state information to selectively cache newly requested objects, caching a requested object not present in the cache, whenever that object should be in (or be put in) the cache according to its state information. Note that after the first L requests following instantiation, the cache will have the cache occupancy probabilities derived earlier for the "always-on" case, and so for requests following the first L requests the analysis in Section III can be used.

Note that over the transient period consisting of the first L requests, no objects are removed from the cache. The average insertion rate during the transient period $\bar{I}_{\text{transient}}$ is therefore given by the average number A of objects in cache (e.g. from (8) for the case of $W = L$) divided by L. Under the assumption that $\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t \geq L$, it is then straightforward to combine $\bar{I}_{\text{transient}}$ with the always-on insertion rate from Section III to obtain $\bar{I}_{t_a:t_d}$.

The average hit rate during the transient period is given by one minus the average transient period insertion rate, minus the average probability that a requested object is not present in the cache and should not be inserted. Recall that the cache receives up-to-date state information when instantiated, and a requested object is cached according to this state information. Therefore, a requested object is not present in the cache and should not be inserted, if and only if it would not be in the cache and would not be inserted into the cache on this request with an always-on cache. The probability of this case is equal to one minus the hit rate for an always-on cache minus the insertion rate for an always-on cache. The above implies that the average hit rate during the transient period, $\bar{H}_{\text{transient}}$, is given by the always-on cache hit rate plus the always-on cache insertion rate (e.g. from (8) for the case of $W = L$) minus the average transient period insertion rate $\bar{I}_{\text{transient}}$; i.e., $\bar{H}_{\text{transient}} = H + I - A/L$. Under the assumption that $\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t \geq L$, it is then straightforward to combine $\bar{H}_{\text{transient}}$ with the always-on hit rate to obtain $\bar{H}_{t_a:t_d}$.

C. Transient Period Performance Results

Figure 3 shows sample results for the transient period when using *Cache on* k^{th} *request* with different $k = 1, 2, 4$ and Zipf with $\alpha = 0.5$ or 1. In all experiments, we used W=L and show results only for the transient period itself. For the analytic expressions, we used the $\mathcal{O}(1)$ approximations for

each metric (see appendix). For the simulations, we start with an empty cache, and simulate the system until the system reaches steady-state conditions. At that time, we empty the cache and begin a new transient period. This is repeated for 2,000 transient periods or until we have simulated 6,000,000 requests, whichever occurs first, and statistics are reported based on fully completed transient periods. With these settings, each data point was calculated based on at least 17 transient periods. (This occurred with A/N=0.2, k=4, and α=1.) To improve readability, as in prior figures, confidence intervals are not included. However, in general, the confidence intervals are tight (e.g., ± 0.0016 for the data point mentioned above).

For the RCW simulations, the system maintains the same state information and operates in the same way as described for our analysis assumptions. As in the steady-state simulations, for the corresponding LRU cache, the capacity C was set equal to A, and when the cache is full, W is dynamically set to the number of requests since the "least recently requested" object currently in the cache (and with at least k requests within W of each other) was last requested. To reach steady state conditions, the cache must be filled completely with objects requested at least k times during that period.

The transient results very much resemble the steady-state results. For example, the tradeoff curves in Figure 3 are very similar to those observed in Figure 1, and the analytic approximations again nicely match the simulated RCW values for most instances (which themselves nicely match the exact analysis results). Most importantly, there is a very good match for all hit rate results (red curves/markers: RCW approximations, RCW simulations, and LRU simulations); the metric that we will use in the optimization models (Section V) and the evaluation thereof (Section VI). Substantive differences between the RCW simulations and analytic approximations are observed only for the insertion fraction metric (shown in blue) when using very small cache sizes (e.g., A/N less than 0.001) when α=0.5. When k=4 and α=0.5, we also observe some noticeable differences in the insertion fraction between RCW and LRU. This may suggest that when k is large, RCW is a worse approximation for LRU (as we compare them) during transient periods than during steady state. Again, in following sections, we use the (more accurate) hit rate results.

V. Optimization Models for Dynamic Instantiation

Consider now the problem of jointly optimizing the capacity C of a dynamically instantiated cache, and the interval over which the cache is allocated, so as to minimize the cache cost subject to achieving a target fraction of requests $H_{\min}$ ($0 < H_{\min} < 1$) that will be served locally:

$$\begin{aligned} &\text{minimize} \quad (t_d - t_a)(C+b), \\ &\text{subject to} \quad \frac{\bar{H}_{t_a:t_d} \int_{t_a}^{t_d} \lambda(t)\mathrm{d}t}{\int_0^T \lambda(t)\mathrm{d}t} \geq H_{\min}. \end{aligned} \tag{11}$$

Note that a smaller cache has the advantages of a shorter transient period until it fills and lower cost per unit time, while a larger cache has the advantage of a higher hit rate once filled. For convenience, in the following we assume that $\lambda(t) > 0, \forall t$.

A. Lower Bound

A lower bound on cost can be obtained by using an upper bound for the average hit rate over the cache allocation interval. One such bound can be obtained by assuming that there is a hit whenever the requested object is one that has been requested previously, since the cache was allocated. We apply this bound to obtain a lower bound on the duration of the cache allocation interval. Another bound is the hit rate when the $\lfloor C \rfloor$ most popular objects are present in the cache. We apply this bound to the more constrained optimization problem that results from our use of the first bound.

Denote by $\bar{H}_R$ the average hit rate over the first R requests after the cache has been allocated. At best, request r, $1 \leq r \leq R$ is a hit if and only if the requested object was the object requested by one or more of the $r-1$ earlier requests, giving:

$$\bar{H}_R \leq \frac{1}{R}\left(\sum_{r=1}^{R}\sum_{i=1}^{N} p_i(1-(1-p_i)^{r-1})\right) = 1-\frac{1}{R}\left(N-\sum_{i=1}^{N}(1-p_i)^R\right). \tag{12}$$

Since this is a concave function of R, we can bound the average hit rate over the cache allocation interval by setting $R = \int_{t_a}^{t_d} \lambda(t)\mathrm{d}t$, the expected value of the number of requests within this interval. Applying this bound to the hit rate constraint in (11) yields

$$\left(\frac{\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t}{\int_0^T \lambda(t)\mathrm{d}t}\right)\left(1-\frac{1}{R}\left(N-\sum_{i=1}^{N}(1-p_i)^R\right)\right) \geq H_{\min}, \tag{13}$$

implying that

$$\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t + \sum_{i=1}^{N}(1-p_i)^{\int_{t_a}^{t_d}\lambda(t)\mathrm{d}t} \geq \left(\int_0^T \lambda(t)\mathrm{d}t\right) H_{\min} + N. \tag{14}$$

Given that we choose t_a and t_d as the beginning and end, respectively, of a time interval with the largest average request rate, the left-hand side is a strictly increasing function of $t_d - t_a$, as can be verified by taking the derivative with respect to $\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t$, noting that this derivative is minimized for minimum $\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t$ (which is at least one, in the region of interest), and using the fact that $-\ln(x)$ is a convex function. Therefore, for any particular workload this relation defines a lower bound D_l for the interval duration $t_d - t_a$.

Applying now the upper bound on hit rate from when the $\lfloor C \rfloor$ most popular objects are present in the cache, gives the following optimization problem:

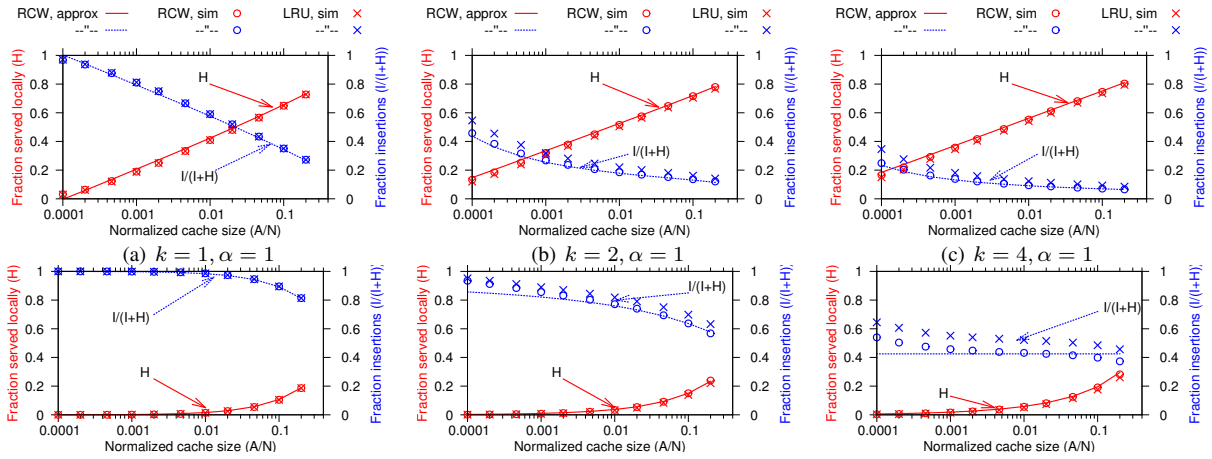
$$\begin{aligned} &\text{minimize} \quad (t_d - t_a)(C+b), \\ &\text{subject to} \quad \left(\frac{\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t}{\int_0^T \lambda(t)\mathrm{d}t}\right)\sum_{i=1}^{\lfloor C \rfloor} p_i \geq H_{\min}, \quad D_l \leq t_d - t_a \leq T. \end{aligned} \tag{15}$$

Solution of this problem yields a lower bound on cost.

B. Policy-based Cost Optimizations

Cache on 1^{st} request: For an LRU cache using this policy, equating the cache capacity C to the average occupancy A of an RCW cache and applying (10) yields:

$$\text{minimize} \quad (t_d - t_a)(C+b), \tag{16}$$



(a) $k = 1, \alpha = 1$ (b) $k = 2, \alpha = 1$ (c) $k = 4, \alpha = 1$

(d) $k = 1, \alpha = 0.5$ (e) $k = 2, \alpha = 0.5$ (f) $k = 4, \alpha = 0.5$

Fig. 3. Transient period results. Performance of *Cache on k^{th} request* (N = 100,000) during transient period.

$$\text{subject to} \quad C = N - \sum_{i=1}^{N}(1-p_i)^L, \quad L \le \int_{t_a}^{t_d} \lambda(t)\mathrm{d}t,$$

$$\frac{L - C + \left(\int_{t_a}^{t_d} \lambda(t)\mathrm{d}t - L\right)\left(1 - \sum_{i=1}^{N} p_i(1-p_i)^L\right)}{\int_0^T \lambda(t)\mathrm{d}t} \ge H_{\min}.$$

Cache on k^{th} request: An analogous optimization problem is obtained for *Cache on k^{th} request* by equating the capacity C to the average occupancy A of an RCW cache and applying the Section IV-B analysis.

VI. Dynamic Instantiation Performance

For an initial model of request rate variation, we use a single-parameter model in which the request rate increases linearly from a rate of zero at the beginning of the time period to a rate λ_{high} half-way through, and then decreases linearly such that the request rate at the end of the period is back to zero. Default parameter settings (each used unless otherwise stated) are T = 1440 min. (24 hours), $\lambda_{\text{high}} = 20$ req./min., $b = 500$ (and so for a cache capacity of 1000 objects, for example, the size-independent portion of the cache cost contributes half of the total), $H_{min} = 0.4$, $N = 100,000$, and a Zipf object popularity distribution with $\alpha = 1$.

Figures 4(a), (b), and (c) show the ratio of the minimal cost for a dynamically instantiated cache using different cache insertion policies (using $W{=}L$ for the *Cache on k^{th} request* policies) to the cost lower bound, as obtained from numerically solving the optimization models of Section V, as a function of the cost parameter b, the hit rate constraint $H_{\min}$, and the peak request rate λ_{high}, respectively. Also shown are the cost ratios for *Cache on 1^{st} request* and *Cache on 2^{nd} request* for the baseline case of a permanently allocated cache with hit rate $H_{\min}$. In each figure, all other parameters are set to their default values. Note that in these results: (1) unless b is very small (in which case, it is most cost-effective to permanently allocate a small cache), $H_{\min}$ is large, or λ_{high} is too small for a dynamically instantiated cache to fill, dynamic cache instantiation can yield substantial cost savings; (2) *Cache on k^{th} request* for $k \ge 2$ provides a better cost/performance tradeoff curve compared to *Cache on 1^{st} request*; and (3) there is only modest room for further improvement through use of more complex cache insertion and replacement policies.

The potential benefits of dynamic cache instantiation (and of caching itself) are strongly dependent on the popularity skew. When object popularities follow a Zipf distribution with $\alpha{=}0.5$, with our default parameters it is not even possible to achieve the target fraction of requests $H_{\min}$ to be served locally, using dynamic cache instantiation. This is partly due to the fact that for $\alpha{=}0.5$, caching performance is degraded much more severely when in the transient period than for $\alpha{=}1$ (e.g., results in Section IV), and partly due to the fact that a larger cache is required to achieve a given hit rate. The impact of the popularity skew can be clearly seen by comparing the results in Figures 5(a) and (b), which use $N{=}10,000$ instead of the default value of 100,000 so as to allow the hit rate constraint to be met over a significant range of values, even when $\alpha{=}0.5$. In addition to the poorer performance of dynamic cache instantiation that is seen in Figure 5(a), note also the increased gap with respect to the lower bound, and the poorer performance of *Cache on 2^{nd} request* relative to *Cache on 1^{st} request* (compared to the relative performance seen in Figure 5(b)). (Results for *Cache on k^{th} request* for $k = 3, 4$ are not shown in Figure 5(a), since the required value of L becomes too large for all but the smallest cache sizes.)

The significant impact of N can be seen by comparing Figures 4(b) and 5(b), which both use $\alpha{=}1$ and differ only in the value of N. Finally, the extent of rate variability also has a substantial impact. This is illustrated in Figure 5(c), for which our model of request rate variation is modified so that the minimum rate is λ_{low}, $0{<}\lambda_{\text{low}}{<}\lambda_{\text{high}}$, rather than zero, and with linear rate increase/decrease occupying only a fraction $1{-}|h|$ of the time period, where h is a parameter between -1 and 1. When $h{>}0$, the request rate is λ_{low} for the rest of the time period, while when $h{<}0$, the request rate is λ_{high} for the rest of the time period (and so during the fraction $1{-}|h|$ of the time period the rate first decreases linearly to λ_{low} and then increases linearly back to λ_{high}), giving a peak to mean request rate ratio for $-1{<}h{<}1$ of $2/(1{-}h{+}(1{+}h)\lambda_{\text{low}}/\lambda_{\text{high}})$.

Results are shown for varying h, with λ_{low} fixed at 10% of λ_{high}, and λ_{high} scaled for each value of h so as to maintain the same total request volume as with the default single-parameter model. Note that $h{=}1$ and $h{=}{-}1$ correspond to the same scenario, since in both cases the request rate is constant throughout the period. Note also that the lower bound becomes overly optimistic for h around 0.7; in this case the requests are highly concentrated, and the solution to the lower bound optimization problem is a large cache allocated for a short period of time (for which the upper bound on hit rate when the $\lfloor C \rfloor$ most popular objects are present in the cache becomes quite loose). Most importantly, observe that when the pattern of request rate variation is such that there is a substantial "valley" ($h{>}0$) within the time period during which the request rate is relatively low, the benefits of dynamic instantiation are much higher than when there is a substantial "plateau" ($h{<}0$).

VII. Related Work

Caching policies typically either evict objects from the cache when the cache becomes full (i.e., capacity-driven policies [12]–[17]) or based on the time since each individual object was last accessed or entered the cache (i.e., timing-based policies [16], [17]). In practice, use of capacity-based policies such as LRU have dominated. Unfortunatly, these policies are hard to analyze exactly (e.g., [18], [19]). This prompted the development of approximations [14], [20], [21] and asymptotic analyses [22], [23]. Most recent work relies on accurate approximation of the performance of capacity-driven policies using TTL-based caches [6]–[10], [24]. TTL-based models have also been used to analyze networks of caches [9], [10], [25], [26], to derive asymptotically optimized solutions [27], for optimized server selection [28], for utility maximization [29], and for on-demand contract design [30].

Few papers (regardless of replacement policy) have modeled discriminatory caching policies such as *Cache on k^{th} request*. In our context, these policies are motivated by the risk of cache pollution in small dynamically instantiated caches, and more generally by the long tail of one-timers (one-hit wonders) observed in edge networks [5], [31]–[33]. Recent works include trace-based evaluations of *Cache on k^{th} request* policies [32], [33], simple analytic models for hit and insertion probabilities that (in contrast to us) ignore cache replacement [33], or works that have used TTL-based recurrence expressions to model variations of *Cache on k^{th} request* [10], [24], [34]–[36]. Of these works, none consider dynamic cache instantiation.

Other related works have adapted the individual TTL values of each object so to achieve some objective [37], [38]. For example, Carra et al. [1] demonstrate how the individual TTL values of each object can be adapted (with constant overhead) so to closely track the optimal cache configurations. However, these works only consider *Cache on 1^{st} request* policies.

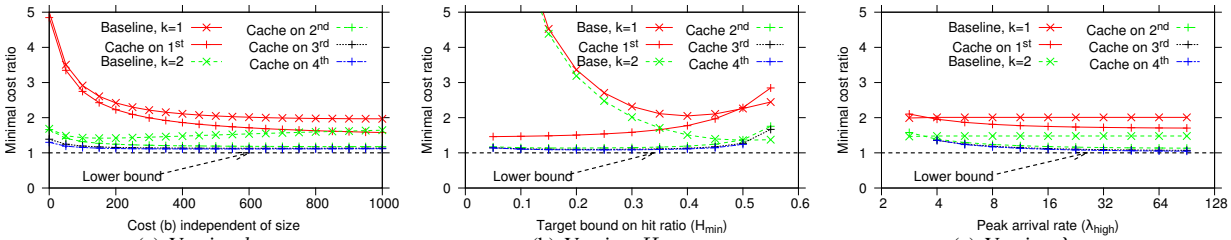
To simplify analysis, TTL-based approximations of LRU-based caches typically leverage the general idea of a cache characterization time [6], [7], which in the simplest case corresponds to the (approximate) time that the object stays in the cache if not requested again. This time corresponds closely to our parameter L, with the important difference that the RCW caches use a *request count window* rather than a *time window*. This subtle difference makes our approach favorable when modeling fixed capacity caches in systems with substantial request rate variations (e.g., according to a diurnal cycle). Furthermore, our RCW approach allows us to derive (i) exact expressions for general *Cache on k^{th} request* policies, popularity distributions, and transient periods, and (ii) corresponding $\mathcal{O}(1)$ computational cost approximations. Such results, which we need for our optimization models, are not found in the TTL-based modeling literature.

For the case of an infinite *Cache on 1^{st} request* cache with a finite request stream, Breslau et al. [20] provide exact hit rate expression for general popularity distributions, as well as approximate scaling properties for Zipf distributions. However, we did not find these scaling relationships (focusing on orders) sufficient for our analysis and therefore developed more precise expressions for Zipf with $\alpha = 1$ and $\alpha = 0.5$.

The (general) idea of dynamically adapting the amount of dedicated resource based on time-varying workloads is not new [39], [40]. In the context of cloud-based caching, in addition to the work by Carra et al. [1] (discussed above), Sundarrajan et al. [2] use discriminatory caching algorithms together with partitioned caches to save energy during off-peak hours, and Dan and Carlsson [3] optimize what objects to push to cloud storage based on diurnal demands and a basic cost model. Others have considered how CDNs best collaborate with ISPs making available microdatacenters [41], how to build virtual CDNs on-the-fly on top of shared third-party infrastructures [4], or optimized a cloud caching hierarchy under the assumption that request rates are known and stationary (ignoring time-varying request loads) [42]. None of these works consider the problem of optimized cache instantiation.

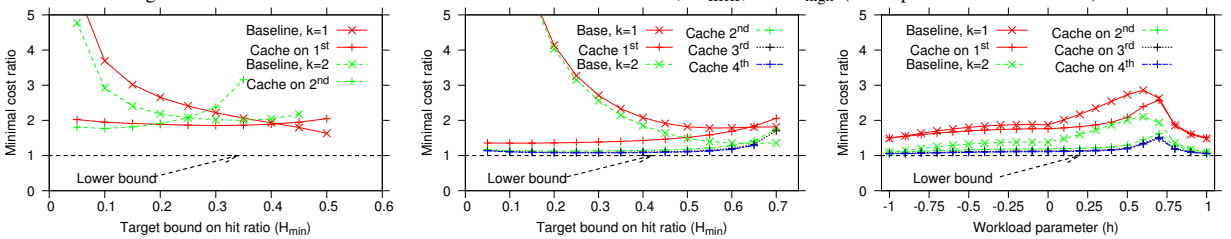
VIII. Conclusions

In this paper we have taken a first look at dynamic cache instantiation. For this purpose, we have derived new analysis results for what we term "request count window" (RCW) caches, including explicit, exact expressions for cache performance metrics for *Cache on k^{th} request* RCW caches for general k, and new $\mathcal{O}(1)$ computational cost approximations for cache performance metrics for Zipf popularity distributions with $\alpha{=}1$ and $\alpha{=}0.5$. These results are of interest in their own right, especially as the performance of RCW caches are shown to closely match the performance of the corresponding *Cache on k^{th} request* LRU caches. We then applied our analysis results to develop optimization models for dynamic cache instantiation and derived a cost lower bound that holds for *any* caching policy. Our results show that dynamic cache instantiation using a selective cache insertion policy such as *Cache on 2^{nd} request* may yield substantial benefits compared to a permanently allocated cache, and is a promising approach for content delivery applications. Finally, we note that the use of the independent reference model (used here) provides conservative estimates of the potential improvements using dynamic instantiation, since in practice short-term temporal



(a) Varying b (b) Varying $H_{\min}$ (c) Varying λ_{high}

Fig. 4. Ratio of the minimal cost to the lower bound versus b, $H_{\min}$, and λ_{high} (other parameters at defaults).



(a) Varying $H_{\min}$, $\alpha = 0.5$, $N = 10,000$ (b) Varying $H_{\min}$, $\alpha = 1$, $N = 10,000$ (c) Varying h; λ_{low} = 0.1, λ_{high}, λ_{high} scaled so req. vol. unchanged

Fig. 5. Impact of popularity skew, number of objects, and request rate variability.

locality, non-stationary object popularities, and high rates of addition of new content typically would make yesterday's cache contents less useful.

Acknowledgements: This work was partially funded by the Swedish Research Council (VR) and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

APPENDIX

We have derived approximate expressions of $\mathcal{O}(1)$ computational cost for the cases of Zipf object popularities with parameter α=1 and α=0.5. Table I summarizes the key approximations derived and used in this paper, and in the following we present a simple example illustrating the general methodology used to obtain these approximations. (The full derivations will be provided in an extended technical report.)

First, let us derive an approximation for the $\sum_{i=1}^{N}(1-p_i)^L$ for the case α=1. In this case, $p_i = 1/(i\Omega)$, where $\Omega = \sum_{i=1}^{N} 1/i$ is a normalization constant, and for large N, $\Omega \approx \ln N + \gamma$, where γ denotes the Euler-Mascheroni constant (≈ 0.577). For large N, L, and $g \to \infty$, we then have:

$$\sum_{i=1}^{N}(1-p_i)^L \approx \sum_{i=1}^{N} e^{-L/(i\Omega)} \approx \int_{L/(g(\ln N+\gamma))}^{N} e^{-L/(x(\ln N+\gamma))}\mathrm{d}x. \tag{17}$$

Next, using a Taylor series expansion for e^y gives:

$$\int_{L/(g(\ln N+\gamma))}^{N} e^{-L/(x(\ln N+\gamma))}\mathrm{d}x = \int_{\frac{L}{g(\ln N+\gamma)}}^{N} \sum_{j=0}^{\infty} \frac{(\frac{-L}{x(\ln N+\gamma)})^j}{j!}\mathrm{d}x$$

$$= N - \frac{L}{\ln N+\gamma}\left(\ln N + \frac{L}{2N(\ln N+\gamma)} - \sum_{j=2}^{\infty}\frac{(\frac{-L}{N(\ln N+\gamma)})^j}{(j+1)!j}\right)$$

$$- \frac{L}{\ln N+\gamma}\left(1/g - \ln(L/(\ln N+\gamma)) + \ln(g) + \sum_{j=1}^{\infty}\frac{(-g)^j}{(j+1)!j}\right). \tag{18}$$

Now, note that $\ln(g) + \sum_{j=1}^{\infty}\frac{(-g)^j}{j!j} = \mathrm{Ei}(-g) - \gamma$, where Ei is the exponential integral function, and that $\sum_{j=1}^{\infty}\frac{(-g)^j}{(j+1)!j} - \sum_{j=1}^{\infty}\frac{(-g)^j}{j!j} = \frac{1}{g}\sum_{j=1}^{\infty}\frac{(-g)^{j+1}}{(j+1)!} = \frac{1}{g}\left(e^{-g}+g-1\right)$, which tends to 1 as $g \to \infty$. Also, for $g \to \infty$, $\mathrm{Ei}(-g) \to 0$. Therefore, for $g \to \infty$,

$$1/g + \ln(g) + \sum_{j=1}^{\infty}\frac{(-g)^j}{(j+1)!j} \to 1-\gamma. \tag{19}$$

Substituting this result into (18), and neglecting the terms in the summation on the second line of (18) under the assumption that L is substantially smaller than $N(\ln N+\gamma)$, yields the result. Similar derivation steps yielded the approximations for the other three sums (in the first two rows of the table).

Now, applying these summation approximations to the equation for A, H, and I in (1), we obtain the approximations for *Cache on 1st request*. Similarly, for *Cache on k^{th} request* and $W = L$, we applied the summation approximations to (8). With respect to the range of values for L for which the *Cache on 2nd request* approximations are accurate when $W = L$ and $\alpha = 1$, note that the equations in (8) with $k = 2$ include both $(1-p_i)^L$ and $(1-p_i)^{2L}$ terms. Therefore, when L is substantially smaller than $N(\ln N+\gamma)$, but $2L$ is not, the accuracy of these approximations is uncertain *a priori*, and requires experimental assessment. A similar issue arises in the case of $\alpha = 0.5$, and for *Cache on k^{th} request* with $k > 2$.

Finally, the transient approximations (three last rows) are obtained by inserting the appropriate steady-state approximations for A, H and I (prior rows in the table) into the equation $\bar{H}_{\text{transient}} = H + I - A/L$, and simplifying. Note that the ratio of the average cache hit rate over the transient period to the hit rate once the cache has filled can yield substantial insight into the impact of the transient period on performance. For example, for the case with $k = 1$ and $\alpha = 1$, the ratio is approximately $1-(1-L/(2N(\ln N+\gamma)))/(\ln N+\gamma)$ and for $k = 1$ and $\alpha = 0.5$, the ratio can be shown to be between 0.5 and 0.7 (for $0 < L < 2N$). Similarly, for $k = 2$, the ratio with $\alpha = 1$ is $1-(\ln 2)/(\ln(L/(\ln N+\gamma)) + 2\gamma - \ln 2)$ and with $\alpha = 0.5$ it is between about 0.64 and 0.72 (considering here $0 < L < N$). These results show that the ratio typically is substantially smaller with $\alpha = 0.5$ than with $\alpha = 1$.

TABLE I
$\mathcal{O}(1)$ APPROXIMATIONS.

	Policy (or sum)	Zipf, $\alpha = 1$	Zipf, $\alpha = 0.5$
Sums	$\sum_{i=1}^{N}(1-p_i)^L$	$N - \frac{L}{\ln N+\gamma}\left(\ln N - \ln\left(\frac{L}{\ln N+\gamma}\right) + 1 - \gamma + \frac{L}{2N(\ln N+\gamma)}\right)$	$N - L + \frac{L^2}{4N}\left(\ln((2N)/L) + \frac{L}{6N} + \frac{3}{2} - \gamma\right)$
	$\sum_{i=1}^{N} p_i(1-p_i)^L$	$1 - \frac{\ln(L/(\ln N+\gamma))+2\gamma-L/(N(\ln N+\gamma))}{\ln N+\gamma}$	$1 - \frac{L}{2N}\left(\ln((2N)/L) + \frac{L}{4N} + 1 - \gamma\right)$
Always on (steady state)	Cache on 1^{st}	$A \approx \frac{L}{\ln N+\gamma}\left(\ln N - \ln\left(\frac{L}{\ln N+\gamma}\right) + 1 - \gamma + \frac{L}{2N(\ln N+\gamma)}\right)$ $I \approx 1 - \frac{\ln(L/(\ln N+\gamma))+2\gamma-L/(N(\ln N+\gamma))}{\ln N+\gamma}$ $H \approx \frac{\ln(L/(\ln N+\gamma))+2\gamma-L/(N(\ln N+\gamma))}{\ln N+\gamma}$	$A \approx L\left(1 - \frac{L}{4N}\left(\ln((2N)/L) + \frac{L}{6N} + \frac{3}{2} - \gamma\right)\right)$ $I \approx 1 - \frac{L}{2N}\left(\ln((2N)/L) + \frac{L}{4N} + 1 - \gamma\right)$ $H \approx \frac{L}{2N}\left(\ln((2N)/L) + \frac{L}{4N} + 1 - \gamma\right)$
	Cache on 2^{nd}, $W = L$	$A \approx \frac{(2\ln 2 - L/(N(\ln N+\gamma)))L}{\ln N+\gamma}$ $H \approx \frac{\ln(L/(\ln N+\gamma))+2\gamma-\ln 2}{\ln N+\gamma}$ $I \approx \frac{\ln 2 - L/(N(\ln N+\gamma))}{\ln N+\gamma}$	$A \approx \frac{L^2}{2N}\left(\ln(N/(2L)) + \frac{L}{2N} + \frac{3}{2} - \gamma\right)$ $H \approx \frac{L}{N}\left(\ln 2 - \frac{L}{4N}\right)$ $I \approx \frac{L}{2N}\left(\ln(N/(2L)) + \frac{3L}{4N} + 1 - \gamma\right)$
	Cache on k^{th}, $k \geq 3$, $W = L$	$A \approx \frac{\left(\sum_{j=2}^{k}(-1)^j\binom{k}{j} j\ln(j)\right)L}{\ln N+\gamma}$ $H \approx \frac{\ln(L/(\ln N+\gamma))+2\gamma-\sum_{j=2}^{k}(-1)^j\binom{k}{j}\ln(j)}{\ln N+\gamma}$ $I \approx \frac{\sum_{j=2}^{k}(-1)^j\binom{k-1}{j-1}\ln(j)}{\ln N+\gamma}$	$A \approx \begin{cases} (9\ln 3 - 12\ln 2 - L/N)\,L^2/(4N) & k=3, \\ \left(\sum_{j=2}^{k}(-1)^{j+1}\binom{k}{j} j^2\ln(j)\right)L^2/(4N) & k \geq 4. \end{cases}$ $H \approx \frac{L}{2N}\sum_{j=2}^{k}(-1)^j\binom{k}{j} j\ln(j)$ $I \approx \begin{cases} (L/(2N))\left(3\ln 3 - 4\ln 2 - \frac{L}{2N}\right) & k=3, \\ (L/(2N))\sum_{j=1}^{k-1}(-1)^j\binom{k-1}{j}(j+1)\ln(j+1) & k \geq 4. \end{cases}$
Transient	Cache on 1^{st}	$\bar{H}_{\text{transient}} \approx \frac{\ln(L/(\ln N+\gamma))+2\gamma-1-L/(2N(\ln N+\gamma))}{\ln N+\gamma}$	$\bar{H}_{\text{transient}} \approx \frac{L}{4N}\left(\ln((2N)/L) + \frac{L}{6N} + \frac{3}{2} - \gamma\right)$
	Cache on 2^{nd}, $W = L$	$\bar{H}_{\text{transient}} \approx \frac{\ln(L/(\ln N+\gamma))+2\gamma-2\ln 2}{\ln N+\gamma}$	$\bar{H}_{\text{transient}} \approx \frac{L}{N}\left(\ln 2 - \frac{L}{8N} - \frac{1}{4}\right)$
	Cache on k^{th}, $k \geq 3$, $W = L$	Insert above into $\bar{H}_{\text{transient}} = H + I - \frac{A}{L}$	Insert above into $\bar{H}_{\text{transient}} = H + I - \frac{A}{L}$

REFERENCES

[1] D. Carra, G. Neglia, and P. Michiardi, "TTL-based cloud caches," in *Proc. IEEE INFOCOM*, 2019.

[2] A. Sundarrajan, M. Kasbekar, and R. Sitaraman, "Energy-efficient disk caching for content delivery," in *Proc. ACM e-Energy*, 2016.

[3] G. Dan and N. Carlsson, "Dynamic content allocation for cloud-assisted service of periodic workloads," in *Proc. IEEE INFOCOM*, 2014.

[4] S. Kuenzer *et al.*, "Unikernels everywhere: The case for elastic CDNs," *ACM SIGPLAN Notices*, vol. 52, no. 7, pp. 15–29, 2017.

[5] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "YouTube traffic characterization: A view from the edge," in *Proc. IMC*, 2007.

[6] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE JSAC*, vol. 20, 2002.

[7] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for LRU cache performance," in *Proc. ITC*, 2012.

[8] G. Bianchi, A. Detti, A. Caponi, and N. B. Melazzi, "Check before storing: What is the performance price of content integrity verification in LRU caching?" *ACM CCR*, vol. 43, no. 3, pp. 59–67, 2013.

[9] D. S. Berger, P. Gland, S. Singla, and F. Ciucu, "Exact analysis of TTL cache networks," *Performance Evaluation*, vol. 79, pp. 2–23, 2014.

[10] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM ToMPECS*, vol. 1, 2016.

[11] F. Chen *et al.*, "End-user mapping: Next generation request routing for content delivery," in *Proc. ACM SIGCOMM*, 2015.

[12] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys*, vol. 35, no. 4, pp. 374–398, 2003.

[13] G. Barish and K. Obraczke, "World wide web caching: trends and techniques," *IEEE Comm. Mag.*, vol. 38, no. 5, pp. 178–184, 2000.

[14] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," in *Proc. ACM SIGMETRICS*, 1990.

[15] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3–11, 2000.

[16] J. Jung, A. W. Berger, and H. Balakrishnan, "Modeling TTL-based Internet caches," in *Proc. IEEE INFOCOM*, 2003.

[17] O. Bahat and A. M. Makowski, "Measuring consistency in TTL-based caches," *Performance Evaluation*, vol. 62, no. 1, pp. 439–455, 2005.

[18] W. F. King III, "Analysis of demand paging algorithms," in *Proc. IFIP Congress*, 1971, (Also IBM Research Report, RC 3288, Mar., 1971.).

[19] E. Gelenbe, "A unified approach to the evaluation of a class of replacement algorithms," *IEEE Trans. on Computers*, vol. 22, 1973.

[20] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE INFOCOM*, 1999.

[21] E. J. Rosensweig, J. Kurose, and D. Towsley, "Approximate models for general cache networks," in *Proc. IEEE INFOCOM*, 2010.

[22] P. R. Jelenkovic and A. Radovanovic, "Asymptotic insensitivity of least-recently-used caching to statistical dependency," in *Proc. IEEE INFOCOM*, 2003.

[23] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance evaluation of the random replacement policy for networks of caches," *Performance Evaluation*, vol. 72, pp. 16–36, 2014.

[24] N. Carlsson and D. Eager, "Worst-case bounds and optimized cache on m-th request cache insertion policies under elastic conditions," *Performance Evaluation*, vol. 127-128, pp. 70–92, 2018.

[25] N. C. Fofack, P. Nain, G. Neglia, and D. Towsely, "Analysis of TTL-based cache networks," in *Proc. VALUETOOLS*, 2012.

[26] N. C. Fofack *et al.*, "On the performance of general cache networks," in *Proc. VALUETOOLS*, 2014.

[27] A. Ferragut, I. Rodriguez, and F. Paganini, "Optimizing TTL caches under heavy-tailed demands," in *Proc. ACM SIGMETRICS*, 2016.

[28] N. Carlsson, D. Eager, A. Gopinathan, and Z. Li, "Caching and optimized request routing in cloud-based content delivery systems," *Performance Evaluation*, vol. 79, pp. 38–55, 2014.

[29] M. Dehghan, L. Massoulie, D. Towsley, D. S. Menasche, and Y. C. Tay, "A utility optimization approach to network cache design," in *Proc. IEEE INFOCOM*, 2016.

[30] R. T. B. Ma and D. Towsley, "Cashing in on caching: on-demand contract design with linear pricing," in *Proc. ACM CoNEXT*, 2015.

[31] M. Zink, K. Suh, Y. Gu, and J. Kurose, "Characteristics of YouTube network traffic at a campus network - measurements, models, and implications," *Computer Networks*, vol. 53, no. 4, pp. 501–514, 2009.

[32] B. Maggs and K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM CCR*, vol. 45, no. 3, pp. 52–66, 2015.

[33] N. Carlsson and D. Eager, "Ephemeral content popularity at the edge and implications for on-demand caching," *IEEE Trans. on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1621–1634, 2017.

[34] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," in *Proc. INFOCOM*, 2014.

[35] N. Gast and B. Van Houdt, "Transient and steady-state regime of a family of list-based cache replacement algorithms," in *Proc. ACM SIGMETRICS*, 2015.

[36] N. Gast and B. V. Houdt, "Asymptotically exact TTL-approximations of the cache replacement algorithms LRU(m) and h-LRU," in *ITC*, 2016.

[37] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi, "Elastic caching," in *Proc. ACM-SIAM SODA*, 2019.

[38] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman, "Adaptive TTL-based caching for content delivery," in *Proc. ACM SIGMETRICS (abstract)*, 2017.

[39] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1378–1391, 2013.

[40] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[41] B. Frank *et al.*, "Pushing CDN-ISP collaboration to the limit," *ACM CCR*, vol. 43, no. 3, pp. 34–44, 2013.

[42] P. Marchetta, J. Llorca, A. M. Tulino, and A. Pescape, "Mc3: A cloud caching strategy for next generation virtual content distribution networks," in *Proc. IFIP Networking*, 2016.