Practical Bounds on Optimal Caching with Variable Object Sizes

DANIEL S. BERGER, Carnegie Mellon University
NATHAN BECKMANN, Carnegie Mellon University
MOR HARCHOL-BALTER, Carnegie Mellon University

Many recent caching systems aim to improve miss ratios, but there is no good sense among practitioners of how much further miss ratios can be improved. In other words, should the systems community continue working on this problem?

Currently, there is no principled answer to this question. In practice, object sizes often vary by several orders of magnitude, where computing the optimal miss ratio (OPT) is known to be NP-hard. The few known results on caching with variable object sizes provide very weak bounds and are impractical to compute on traces of realistic length.

We propose a new method to compute upper and lower bounds on OPT. Our key insight is to represent caching as a min-cost flow problem, hence we call our method the *flow-based offline optimal* (FOO). We prove that, under simple independence assumptions, FOO's bounds become tight as the number of objects goes to infinity. Indeed, FOO's error over 10M requests of production CDN and storage traces is negligible: at most 0.3%. FOO thus reveals, for the first time, the limits of caching with variable object sizes.

While FOO is very accurate, it is computationally impractical on traces with hundreds of millions of requests. We therefore extend FOO to obtain more efficient bounds on OPT, which we call *practical flow-based offline optimal* (PFOO). We evaluate PFOO on several full production traces and use it to compare OPT to prior online policies. This analysis shows that current caching systems are in fact still far from optimal, suffering 11–43% more cache misses than OPT, whereas the best prior offline bounds suggest that there is essentially no room for improvement.

CCS Concepts: • **Theory of computation → Caching and paging algorithms**; *Approximation algorithms analysis*; • **General and reference** → *Metrics*; *Performance*;

Additional Key Words and Phrases: Caching, optimal caching, offline optimum, limits of caching

ACM Reference Format:
Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical Bounds on Optimal Caching with Variable Object Sizes. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 32 (June 2018), 38 pages. https://doi.org/10.1145/3224427

1 INTRODUCTION

Caches are pervasive in computer systems, and their miss ratio often determines end-to-end application performance. For example, content distribution networks (CDNs) depend on large, geo-distributed caching networks to serve user requests from nearby datacenters, and their response time degrades dramatically when the requested content is not cached nearby. Consequently, there has been a renewed focus on improving cache miss ratios, particularly for web services and content

Authors' addresses: Daniel S. Berger, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213, dsberger@cs.cmu.edu; Nathan Beckmann, Carnegie Mellon University, beckmann@cs.cmu.edu; Mor Harchol-Balter, Carnegie Mellon University, harchol@cs.cmu.edu.

2476-1249/2018/6-ART32 $15.00
https://doi.org/10.1145/3224427

delivery [16, 17, 24, 34, 42, 48, 60, 62, 68]. These systems have demonstrated significant miss ratio improvements over least-recently used (LRU) caching, the de facto policy in most production systems. *But how much further can miss ratios improve? Should the systems community continue working on this problem, or have all achievable gains been exhausted?*

1.1 The problem: Finding the offline optimal

To answer these questions, one would like to know the best achievable miss ratio, free of constraints—i.e., the offline optimal (OPT). Unfortunately, very little is known about OPT with variable object sizes. For objects with equal sizes, computing OPT is simple (i.e., Belady [13, 64]), and it is widely used in the systems community to bound miss ratios. But object sizes often vary widely in practice, from a few bytes (e.g., metadata [69]) to several gigabytes (e.g., videos [16, 48]). We need a way to compute OPT for variable object sizes, but unfortunately this is known to be NP-hard [23].

There has been little work on bounding OPT with variable object sizes, and all of this work gives very weak bounds. On the theory side, prior work gives only three approximation algorithms [4, 7, 50], and the best approximation is only provably within a factor of 4 of OPT. Hence, when this algorithm estimates a miss ratio of 0.4, OPT may lie anywhere between 0.1 and 0.4. This is a big range—in practice, a difference of 0.05 in miss ratio is significant—, so bounds from prior theory are of limited practical value. From a practical perspective, there is an even more serious problem with the theoretical bounds: they are simply too expensive to compute. The best approximation takes 24 hours to process 500 K requests and scales poorly (Section 2), while production traces typically contain hundreds of millions of requests.

Since the theoretical bounds are incomputable, practitioners have been forced to use conservative lower bounds or pessimistic upper bounds on OPT. The only prior lower bound is an infinitely large cache [1, 24, 48], which is very conservative and gives no sense of how OPT changes at different cache sizes. Belady variants (e.g., Belady-Size in Section 2.4) are widely used as an upper bound [46, 48, 61, 77], despite offering no guarantees of optimality. While these offline bounds are easy to compute, we will show that they are in fact far from OPT. They have thus given practitioners a false sense of complacency, since existing online algorithms often achieve similar miss ratios to these weak offline upper bounds.

1.2 Our approach: Flow-based offline optimal

We propose a new approach to compute bounds on OPT with variable object sizes, which we call the *flow-based offline optimal (FOO).* The key insight behind FOO is to represent caching as a min-cost flow problem. This formulation yields a lower bound on OPT by allowing non-integer decisions, i.e., letting the cache retain fractions of objects for a proportionally smaller reward. It also yields an upper bound on OPT by ignoring all non-integer decisions. Under simple independence assumptions, we prove that the non-integer decisions become negligible as the number of objects goes to infinity, and thus the bounds are asymptotically tight.

Our proof is based on the observation that an optimal policy will strictly prefer some requests over others, forcing integer decisions. We show such preferences apply to almost all requests by relating such preferences to the well-known coupon collector problem.

Indeed, FOO's error over 10M requests of five production CDN and web application traces is negligible: at most 0.15%. Even on storage traces, which have highly correlated requests that violate our proof's independence assumptions, FOO's error remains below 0.3%. FOO thus reveals, for the first time, the limits of caching with variable object sizes.

While FOO is very accurate, it is too computationally expensive to apply directly to production traces containing hundreds of millions of requests. To extend our analysis to such traces, we develop more efficient upper and lower bounds on OPT, which we call *practical flow-based offline optimal*

(PFOO). PFOO enables the first analysis of optimal caching on traces with hundreds of millions of requests, and reveals that there is still significant room to improve current caching systems.

1.3 Summary of results

PFOO yields nearly tight bounds, as shown in Figure 1. This figure plots the miss ratio obtained on a production CDN trace with over 400 million requests for several techniques: online algorithms (LRU, GDSF [22], and AdaptSize [16]), prior offline upper bounds (Belady and Belady-Size), the only prior offline lower bound (Infinite-Cap), and our new upper and lower bounds on OPT (PFOO). The theoretical bounds, including both prior work and FOO, are too slow to run on traces this long.

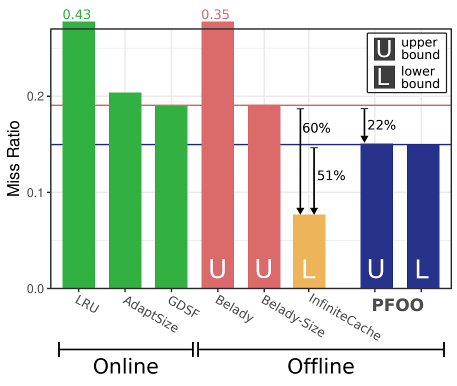


Fig. 1. Miss ratios on a production CDN trace for a 4 GB cache. Prior to our work, the best prior upper bound on OPT is within 1% of online algorithms on this trace, leading to the false impression that there is no room for improvement. The only prior lower bound on OPT (Infinite-Cap) is 60% lower than the best upper bound on OPT (Belady-Size). By contrast, PFOO provides nearly tight upper and lower bounds, which are 22% below the online algorithms. PFOO thus shows that there is actually significant room for improving current caching algorithms.

Figure 1 illustrates the two problems with prior practical bounds and how PFOO improves upon them. First, prior bounds on OPT are very weak. The miss ratio of Infinite-Cap is 60% lower than Belady-Size, whereas the gap between PFOO's upper and lower bounds is less than 1%. Second, prior bounds on OPT are misleading. Comparing GDSF and Belady-Size, which are also within 1%, one would conclude that online policies are nearly optimal. In contrast, PFOO gives a much better upper bound and shows that there is in fact a 22% gap between state-of-the-art online policies and OPT.

We evaluate FOO and PFOO extensively on eight production traces with hundreds of millions of requests across cache capacities from 16 MB to 64 GB. On CDN traces from two large Internet companies, PFOO bounds OPT within 1.9% relative error at most and 1.4% on average. On web application traces from two other large Internet companies, PFOO bounds OPT within 8.6% relative error at most and 6.1% on average. On storage traces from Microsoft [79], where our proof assumptions do not hold, PFOO still bounds OPT within 11% relative error at most and 5.7% on average.

PFOO thus gives nearly tight bounds on the offline optimal miss ratio for a wide range of real workloads. We find that PFOO achieves on average 19% lower miss ratios than Belady, Belady-Size, and other obvious offline upper bounds, demonstrating the value of our min-cost flow formulation.

1.4 Contributions

In summary, this paper contributes the following:

- We present *flow-based offline optimal (FOO)*, a novel formulation of offline caching as a min-cost flow problem (Sections 3 and 4). FOO exposes the underlying problem structure and enables our remaining contributions.
- We prove that FOO is asymptotically exact under simple independence assumptions, giving the first tight, polynomial-time bound on OPT with variable object sizes (Section 5).

- We present *practical flow-based offline optimal (PFOO)*, which relaxes FOO to give fast and nearly tight bounds on real traces with hundreds of millions of requests (Section 6). PFOO gives the first reasonably tight lower bound on OPT on long traces.
- We perform the first extensive evaluation of OPT with variable object sizes on eight production traces (Sections 7 and 8). In contrast to prior offline bounds, PFOO reveals that the gap between online policies and OPT is much larger than previously thought: 27% on average, and up to 43% on web application traces.

Our implementations of FOO, PFOO, and of the previously unimplemented approximation algorithms are publicly available[1].

2 BACKGROUND AND MOTIVATION

Little is known about how to efficiently compute OPT with variable object sizes. On the theory side, the best known approximation algorithms give very weak bounds and are too expensive to compute. On the practical side, system builders use offline heuristics that are much cheaper to compute, but give no guarantee that they are close to OPT. This section surveys known theoretical results on OPT and the offline heuristics used in practice.

2.1 Offline bounds are more robust than online bounds

Figure 1 showed a 22% gap between the best online policies and OPT. One might wonder whether this gap arises from comparing an *offline* policy with an *online* policy, rather than from any weakness in the online policies. In other words, offline policies have an advantage because they know the future. Perhaps this advantage explains the gap.

This raises the question of whether we could instead compute the *online optimal* miss ratio. Such a bound would be practically useful, since all systems necessarily use online policies. Our answer is that we would like to, but unfortunately this is impossible. To bound the online optimal miss ratio, one must make assumptions about what information an online policy can use to make caching decisions. For example, traditional policies such as LRU and LFU make decisions using the history of past requests. These policies are conceptually simple enough that prior work has proven online bounds on their performance in a variety of settings [2, 6, 36, 49, 72]. However, real systems have quirks and odd behaviors that are hard to capture in tractable assumptions. With enough effort, system designers can exploit these quirks to build online policies that *outperform the online bounds* and approach OPT's miss ratio for specific workloads. For example, Hawkeye [51] recently demonstrated that a seemingly irrelevant request feature in computer architecture (the program counter) is in fact tightly correlated with OPT's decisions, allowing Hawkeye to mimic OPT on many programs. Online bounds are thus inherently fragile because real systems behave in strange ways, and it is impossible to bound the ingenuity of system designers to discover and exploit these behaviors.

We instead focus on the offline optimal to get robust bounds on cache performance. Offline bounds are widely used by practitioners, especially when objects have the same size and OPT is simple to compute [13, 64]. However, computing OPT with variable object sizes is strongly NP-complete [23], meaning that no fully polynomial-time approximation scheme (FPTAS) can exist.[2] We are therefore unlikely to find tight bounds on OPT for arbitrary traces. Instead, our approach is to develop a technique that can estimate OPT on *any* trace, which we call FOO, and then show that FOO yields tight bounds on traces *seen in practice*. To do this, our approach is two-fold. First, we prove that FOO gives tight bounds on traces that obey the independent reference

[1]https://github.com/dasebe/optimalwebcaching
[2]That no FPTAS can exist follows from Corollary 8.6 in [83], as OPT meets the assumptions of Theorem 8.5.

model (IRM), the most common assumptions used in prior analysis of online policies [2, 14, 15, 20, 25, 26, 29, 31, 37, 38, 40–43, 45, 52–54, 58, 63, 65, 72, 73, 75, 76, 80, 82, 86]. The IRM assumption is only needed for the proof; FOO is designed to work on any trace and does not itself rely on any properties of the IRM. Second, we show empirically that FOO's error is less than 0.3% on real-world traces, including several that grossly violate the IRM. Together, we believe these results demonstrate that FOO is both theoretically well-founded and practically useful.

2.2 OPT with variable object sizes is hard

Since OPT is simple to compute for equal object sizes [13, 64], it is surprising that OPT becomes NP-hard when object sizes vary. Caching may seem similar to Bin-Packing or Knapsack, which can be approximated well when there are only a few object sizes and costs. But caching is quite different because the order of requests constrains OPT's choices in ways that are not captured by these problems or their variants. In fact, the NP-hardness proof in [23] is quite involved and reduces from Vertex Cover, not Knapsack. Furthermore, OPT remains strongly NP-complete even with just three object sizes [39], and heuristics that work well on Knapsack perform badly in caching (see "Freq/Size" in Section 2.4 and Section 8).

2.3 Prior bounds on OPT with variable object sizes are impractical

Prior work gives only three polynomial time bounds on OPT [4, 7, 50], which vary in time complexity and approximation guarantee. Table 1 summarizes these bounds by comparing their asymptotic runtime, how many requests can be calculated in practice (e.g., within 24 hrs), and their approximation guarantee.

Technique	**Time**	**Requests / 24hrs**	**Approximation**
OPT	NP-hard [23]	<1 K	1
LP rounding [4]	$\Omega(N^{5.6})$	50 K	$O\left(\log \frac{\max_i \{s_i\}}{\min_i \{s_i\}}\right)$
LocalRatio [7]	$O(N^3)$	500 K	4
OFMA [50]	$O(N^2)$	28 M	$O(\log C)$
FOO[a]	$O(N^{3/2})$	28 M	1
PFOO[b]	$O(N \log N)$	250 M	≈1.06

Notation: N is the trace length, C is the cache capacity, and s_i is the size of object i.

[a]FOO's approximation guarantee holds under independence assumptions.

[b]PFOO does not have an approximation guarantee but its upper and lower bounds are within 6% on average on production traces.

Table 1. Comparison of FOO and PFOO to prior bounds on OPT with variable object sizes. Computing OPT is NP-hard. Prior bounds [4, 7, 50] provide only weak approximation guarantees, whereas FOO's bounds are tight. PFOO performs well empirically and can be calculated for hundreds of millions of requests.

Albers et al. [4] propose an LP relaxation of OPT and a rounding scheme. Unfortunately, the LP requires N^2 variables, which leads to a high $\Omega(N^{5.6})$-time complexity [59]. Not only is this running time high, but the approximation factor is logarithmic in the ratio of largest to smallest object (e.g., around 30 on production traces), making this approach impractical.

Bar et al. [7] propose a general approximation framework (which we call *LocalRatio*), which can be applied to the offline caching problem. This algorithm gives the best known approximation guarantee, a factor of 4. Unfortunately, this is still a weak guarantee, as we saw in Section 1. Additionally, LocalRatio is a purely theoretical algorithm, with a high running time of $O(N^3)$, and which we believe had not been implemented prior to our work. Our implementation of LocalRatio can calculate up to 500 K requests in 24 hrs, which is only a small fraction of the length of production traces.

Irani proposes the OFMA approximation algorithm [50], which has $O(N^2)$ running time. This running time is small enough for our implementation of OFMA to run on small traces (Section 8). Unfortunately, OFMA achieves a weak approximation guarantee, logarithmic in the cache capacity C, and in fact OFMA does badly on our traces, giving much weaker bounds than simple Belady-inspired heuristics.

Hence, prior work that considers adversarial assumptions yields only weak approximation guarantees. We therefore turn to stochastic assumptions to obtain tight bounds on the kinds of traces actually seen in practice. Under independence assumptions, FOO achieves a tight approximation guarantee on OPT, unlike prior approximation algorithms, and also has asymptotically better runtime, specifically $O\left(N^{3/2}\right)$.

We are not aware of any prior stochastic analysis of offline optimal caching. In the context of *online* caching policies, there is an extensive body of work using stochastic assumptions similar to ours [12, 14, 15, 20, 25, 26, 29, 31, 37, 38, 40–43, 45, 52–54, 58, 63, 65, 72, 73, 75, 76, 80, 82, 86], of which five prove optimality results [2, 6, 36, 49, 72]. Unfortunately, these results are only for objects with equal sizes, and these policies perform poorly in our experiments because they do not account for object size.

2.4 Heuristics used in practice to bound OPT give weak bounds

Since the running times of prior approximation algorithms are too high for production traces, practitioners have been forced to rely on heuristics that can be calculated more quickly. However, these heuristics only give upper bounds on OPT, and there is no guarantee on how close to OPT they are.

The simplest offline upper bound is Belady's algorithm, which evicts the object whose next use lies furthest in the future.[3] Belady is optimal in caching variants with equal object sizes [13, 44, 55, 64]. Even though it has no approximation guarantees for variable object sizes, it is still widely used in the systems community [46, 48, 61, 77]. However, as we saw in Figure 1, Belady performs very badly with variable object sizes and is easily outperformed by state-of-the-art online policies.

A straightforward size-aware extension of Belady is to evict the object with the highest cost = object size × next-use distance. We call this variant *Belady-Size*. Among practitioners, Belady-Size is widely believed to perform near-optimally, but it has no guarantees. It falls short on simple examples: e.g., imagine that A is 4 MB and is referenced 10 requests hence and never referenced again, and B is 5 MB and is referenced 9 and 12 requests hence. With 5 MB of cache space, the best choice between these objects is to keep B, getting two hits. But A has cost = $4 \times 10 = 40$, and B has cost = $5 \times 9 = 45$, so Belady-Size keeps A and gets only one hit. In practice, Belady-Size often leads to poor decisions when a large object is referenced twice in short succession: Belady-Size will evict many other useful objects to make space for it, sacrificing many future hits to gain just one.

Alternatively, one could use Knapsack heuristics as size-aware offline upper bounds, such as the density-ordered Knapsack heuristic, which is known to perform well on Knapsack in practice [33]. We call this heuristic *Freq/Size*, as Freq/Size evicts the object with the lowest utility = frequency / size, where frequency is the number of requests to the object. Unfortunately, Freq/Size also falls short on simple examples: e.g., imagine that A is 2 MB and is referenced 10 requests hence, and B is (as before) 5 MB and is referenced 9 and 12 requests hence. With 5 MB of cache space, the best

[3]Belady's MIN algorithm actually operates somewhat differently. The algorithm commonly called "Belady" was invented (and proved to be optimal) by Mattson [64]. For a longer discussion, see [66].

choice between these objects is to keep B, getting two hits. But A has utility = 1 ÷ 2 = 0.5, and B has utility = 2 ÷ 5 = 0.4, so Freq/Size keeps A and gets only one hit. In practice, Freq/Size often leads to poor decisions when an object is referenced in bursts: Freq/Size will retain such objects long after the burst has passed, wasting space that could have earned hits from other objects.

Though these heuristics are easy to compute and intuitive, they give no approximation guarantees. We will show that they are in fact far from OPT on real traces, and PFOO is a much better bound.

3 FLOW-BASED OFFLINE OPTIMAL

This section gives a conceptual roadmap for our proof of FOO's optimality, which we present formally in Sections 4 and 5. Throughout this section we use a small request trace shown in Figure 2 as a running example. This trace contains four objects, **a**, **b**, **c**, and **d**, with sizes 3, 1, 1, and 2, respectively.

Object	**a**	**b**	**c**	**b**	**d**	**a**	**c**	**d**	**a**	**b**	**b**	**a**
Size	3	1	1	1	2	3	1	2	3	1	1	3

Fig. 2. Example trace of requests to objects a, b, c, and d, of sizes 3, 1, 1, and 2, respectively.

First, we introduce a new integer linear program to represent OPT (Section 3.1). After relaxing integrality constraints, we derive FOO's min-cost flow representation, which can be solved efficiently (Section 3.2). We then observe how FOO yields tight upper and lower bounds on OPT (Section 3.3). To prove that FOO's bounds are tight on real-world traces, we relate the gap between FOO's upper and lower bounds to the occurrence of a partial order on intervals, and then reduce the partial order's occurrence to an instance of the generalized coupon collector problem (Section 3.4).

3.1 Our new interval representation of OPT

We start by introducing a novel representation of OPT. Our integer linear program (ILP) minimizes the number of cache misses, while having full knowledge of the request trace.

We exploit a unique property of offline optimal caching: OPT never changes its decision to cache object k in between two requests to k (see Section 4). This naturally leads to an interval representation of OPT as shown in Figure 3. While the classical representation of OPT uses decision variables to track the state of every object at every time step [4], our ILP only keeps track of interval-level decisions. Specifically, we use decision variables x_i to indicate whether OPT caches the object requested at time i, or not.

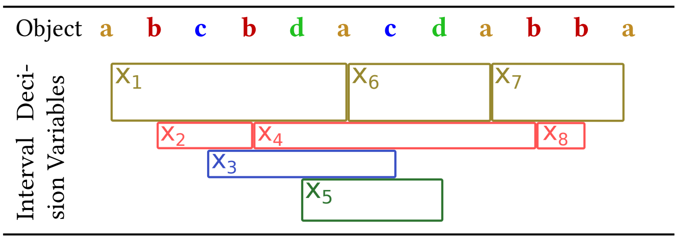
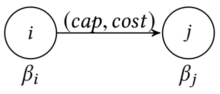


Fig. 3. Interval ILP representation of OPT.

3.2 FOO's min-cost flow representation

This interval representation leads naturally to FOO's flow-based representation, shown in Figure 4. We use the min cost flow notation shown in Table 2. The key idea is to use flow to represent the interval decision variables. Each request is represented by a node. Each object's first request is a source of flow equal to the object's size, and its last request is a sink of flow in the same amount. This flow must be routed along intervening edges, and hence min-cost flow must decide whether to cache the object throughout the trace.



cap	Capacity of edge (i, j), i.e., maximum flow along this edge.
cost	Cost per unit flow on edge (i, j).
β_i	Flow surplus at node i, if $\beta_i > 0$, flow demand if $\beta_i < 0$.

Table 2. Notation for FOO's min-cost flow. Edges are labeled with their (*capacity*, *cost*), and nodes are labeled with flow −*demand* or +*surplus*.

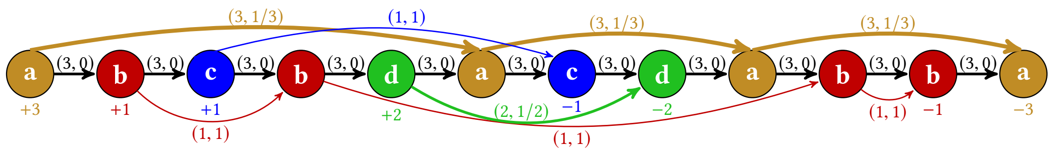


Fig. 4. FOO's min-cost flow problem for the short trace in Figure 2, using the notation from Table 2. Nodes represent requests, and cost measures cache misses. Requests are connected by central edges with capacity equal to the cache capacity and cost zero—flow routed along this path represents cached objects (hits). Outer edges connect requests to the same object, with capacity equal to the object's size—flow routed along this path represents misses. The first request for each object is a source of flow equal to the object's size, and the last request is a sink of flow of the same amount. Outer edges' costs are inversely proportional to object size so that they cost 1 miss when an entire object is not cached. The min-cost flow achieves the fewest misses.

For cached objects, there is a central path of black edges connecting all requests. These edges have capacity equal to the cache capacity and cost zero (since cached objects lead to zero misses). Min-cost flow will thus route as much flow as possible through this central path to avoid costly misses elsewhere [3].

To represent cache misses, FOO adds outer edges between subsequent requests to the same object. For example, there are three edges along the top of Figure 4 connecting the requests to **a**. These edges have capacity equal to the object's size s and cost inversely proportional to the object's size $1/s$. Hence, if an object of size s is not cached (i.e., its flow s is routed along this outer edge), it will incur a cost of $s \times (1/s) = 1$ miss.

The routing of flow through this graph implies which objects are cached and when. When no flow is routed along an outer edge, this implies that the object is cached and the subsequent request is a hit. All other requests, i.e., those with any flow routed along an outer edge, are misses. The min-cost flow gives the decisions that minimize total misses.

3.3 FOO yields upper and lower bounds on OPT

FOO can deviate from OPT as there is no guarantee that an object's flow will be entirely routed along its outer edge. Thus, FOO allows the cache to keep fractions of objects, accounting for only a fractional miss on the next request to that object. In a real system, each fractional miss would be a full miss. This error is the price FOO pays for making the offline optimal computable.

To deal with fractional (non-integer) solutions, we consider two variants of FOO. FOO-L keeps all non-integer solutions and is therefore a lower bound on OPT. FOO-U considers all non-integer decisions as uncached, “rounding up” flow along outer edges, and is therefore an upper bound on OPT. We will prove this in Section 4.

Object	**a**	**b**	**c**	**b**	**d**	**a**	**c**	**d**	**a**	**b**	**b**	**a**
OPT decision	✗	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗
FOO-L decision	0	1	1	1	$\frac{1}{2}$	0	1	$\frac{1}{2}$	0	1	0	0
FOO-U decision	0	1	1	1	0	0	1	0	0	1	0	0

Fig. 5. Caching decisions made by OPT, FOO-L, and FOO-U with a cache capacity of $C = 3$.

Figure 5 shows the caching decisions made by OPT, FOO-L, and FOO-U assuming a cache of size 3. A “✓” indicates that OPT caches the object until its next request, and a “✗” indicates it is not cached. OPT suffers five misses on this trace by caching object **b** and either **c** or **d**. OPT caches **b** because it is referenced thrice and is small. This leaves space to cache the two references to either **c** or **d**, but not both. (OPT in Figure 5 chooses to cache **c** since it requires less space.) OPT does not cache **a** because it takes the full cache, forcing misses on all other requests.

The solutions found by FOO-L are very similar to OPT. FOO-L decides to cache objects **b** and **c**, matching OPT, and also caches *half* of **d**. FOO-L thus underestimates the misses by one, counting **d**’s misses fractionally. FOO-U gives an upper bound for OPT by counting **d**’s misses fully. In this example, FOO-U matches OPT exactly.

3.4 Overview of our proof of FOO’s optimality

We show both theoretically and empirically that FOO-U and FOO-L yield tight bounds. Specifically, we prove that FOO-L’s solutions are almost always integer when there are many objects (as in production traces). Thus, FOO-U and FOO-L coincide with OPT.

Our proof is based on a natural precedence relation between intervals such that an optimal policy strictly prefers some intervals over others. For example, in Figure 2, FOO will always prefer x_2 over x_1 and x_8 over x_7. This can be seen in the figure, as interval x_2 fits entirely within x_1, and likewise x_8 fits within x_7. In contrast, no such precedence relation exists between x_6 and x_4 because **a** is larger than **b**, and so x_6 does not fit within x_4. Similarly, no precedence relation exists between x_2 and x_5 because, although x_5 is longer and larger, their intervals do not overlap, and so x_2 does not fit within x_5.

This precedence relation means that if FOO caches any part of x_1, then it must have cached all of x_2. Likewise, if FOO caches any part of x_7, then it must have cached all of x_8. The precedence relation thus forces integer solutions in FOO. Although this relation is sparse in the small trace from Figure 2, as one scales up the number of objects the precedence relation becomes dense. Our challenge is to prove that this holds on traces seen in practice.

At the highest level, our proof distinguishes between “typical” and “atypical” objects. Atypical objects are those that are exceptionally unpopular or exceptionally large; typical objects are everything else. While the precedence relation may not hold for atypical objects, intervals from atypical objects are rare enough that they can be safely ignored. We then show that for all the typical objects, the precedence relation is dense. In fact, one only needs to consider precedence relations among *cached* objects, as all other interval have zero decision variables. The basic intuition behind our proof is that a popular cached object almost always takes precedence over another object. Specifically, it will take precedence over one of the exceptionally large objects, since the only way it could not is if *all* of the exceptionally large objects were requested before it was requested again. There are enough large objects to make this vanishingly unlikely.

This is an instance of the generalized *coupon collector problem* (CCP). In the CCP, one collects coupons (with replacement) from an urn with k distinct types of coupons, stopping once all k types have been collected. The classical CCP (where coupons are equally likely) is a well-studied problem [35]. The generalized CCP, where coupons have non-uniform probabilities, is very challenging and the focus of recent work in probability theory [5, 32, 67, 85].

Applying these recent results, we show that it is extremely unlikely that a popular object does not take precedence over any others. Therefore, there are very few non-integer solutions among popular objects, which make up nearly all hits, and the gap between FOO-U and FOO-L vanishes as the number of objects grows large.

4 FORMAL DEFINITION OF FOO

This section shows how to construct FOO and that FOO yields upper and lower bounds on OPT. Section 4.1 introduces our notation. Section 4.2 defines our new interval representation of OPT. Section 4.3 relaxes the integer constraints and proves that our min-cost flow representation yields upper and lower bounds on OPT.

4.1 Notation and definitions

The trace σ consists of N requests to M distinct objects. The i-th request σ_i contains the corresponding object id, for all $i \in \{1 \dots N\}$. We use s_i to reference the size of the object σ_i referenced in the i-th request. We denote the i-th interval (e.g., in Figure 3) by $[i, \ell_i)$, where ℓ_i is the time of the next request to object σ_i after time i, or ∞ if σ_i is not requested again.

OPT minimizes the number of cache misses, while having full knowledge of the request trace. OPT is constrained to only use cache capacity C (bytes), and is *not allowed to prefetch* objects as this would lead to trivial solutions (no misses) [4]. Formally,

Assumption 1. *An object $k \in \{1 \dots M\}$ can only enter the cache at times $i \in \{1 \dots N\}$ with $\sigma_i = k$.*

4.2 New ILP representation of OPT

We start by formally stating our ILP formulation of OPT, based on intervals as illustrated in Figure 3. First, we define the set I of all requests i where σ_i is requested again, i.e., $I = \{i : \ell_i < \infty\}$. I is the times when OPT must decide whether to cache an object. For all $i \in I$, we associate a decision variable x_i. This decision variable denotes whether object σ_i is cached during the interval $[i, \ell_i)$. Our ILP formulation needs only $N - M$ variables, vs. $N \times M$ for prior approaches [4], and leads directly to our flow-based approximation.

Definition 1 (**Definition of OPT**). *The interval representation of OPT for a trace of length N with M objects is as follows.*

$$OPT = \min \sum_{i \in I} (1 - x_i) \tag{1}$$

subject to:

$$\sum_{j : j < i < \ell_j} s_j x_j \le C \qquad \forall i \in I \tag{2}$$

$$x_i \in \{0, 1\} \qquad \forall i \in I \tag{3}$$

To represent the capacity constraint at every time step i, our representation needs to find all intervals $[j, \ell_j)$ that intersect with i, i.e., where $j < i < \ell_j$. Eq. (2) enforces the capacity constraint by bounding the size of cached intervals to be less than the cache size C. Eq. (3) ensures that decisions are integral, i.e., that each interval is cached either fully or not at all. Section A.1 proves that our interval ILP is equivalent to classic ILP formulations of OPT from prior work [4].

Having formulated OPT with fewer decision variables, we could try to solve the LP relaxation of this specific ILP. However, the capacity constraint, Eq. (2), still poses a practical problem since finding the intersecting intervals is computationally expensive. Additionally, the LP formulation does not exploit the underlying problem structure, which we need to bound the number of integer solutions. We instead reformulate the problem as min-cost flow.

4.3 FOO's min-cost flow representation of OPT

This section presents the relaxed version of OPT as an instance of min-cost flow (MCF) in a graph G. We denote a surplus of flow at a node i with $\beta_i > 0$, and a demand for flow with $\beta_i < 0$. Each edge (i, j) in G has a cost per unit flow $\gamma_{(i,j)}$ and a capacity for flow $\mu_{(i,j)}$ (see right-hand side of Figure 4).

As discussed in Section 3, the key idea in our construction of an MCF instance is that each interval introduces an amount of flow equal to the object's size. The graph G is constructed such that this flow competes for a single sequence of edges (the "inner edges") with zero cost. These "inner edges" represent the cache's capacity: if an object is stored in the cache, we incur zero cost (no misses). As not all objects will fit into the cache, we introduce "outer edges", which allow MCF to satisfy the flow constraints. However, these outer edges come at a cost: when the full flow of an object uses an outer edge we incur cost 1 (i.e., a miss). Non-integer decision variables arise if part of an object is in the cache (flow along inner edges) and part is out of the cache (flow along outer edges).

Formally, we construct our MCF instance of OPT as follows:

Definition 2 (**FOO's representation of OPT**). *Given a trace with N requests and M objects, the MCF graph G consists of N nodes. For each request $i \in \{1 \dots N\}$ there is a node with supply/demand*

$$\beta_i = \begin{cases} s_i & \text{if } i \text{ is the first request to } \sigma_i \\ -s_i & \text{if } i \text{ is the last request to } \sigma_i \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

*An **inner edge** connects nodes i and $i+1$. Inner edges have capacity $\mu_{(i,i+1)} = C$ and cost $\gamma_{(i,i+1)} = 0$, for $i \in \{1 \dots N-1\}$.*

*For all $i \in I$, an **outer edge** connects nodes i and ℓ_i. Outer edges have capacity $\mu_{(i,\ell_i)} = s_i$ and cost $\gamma_{(i,\ell_i)} = 1/s_i$. We denote the flow through outer edge (i, ℓ_i) as f_i.*

***FOO-L** denotes the cost of an optimal feasible solution to the MCF graph G. **FOO-U** denotes the cost if all non-zero flows through outer edges f_i are rounded up to the edge's capacity s_i.*

This representation yields a min-cost flow instance with $2N - M - 1$ edges, which is solvable in $O(N^{3/2})$ [8, 9, 28]. Note that while this paper focuses on optimizing miss ratio (i.e., the fault model [4], where all misses have the same cost), Definition 2 easily supports non-uniform miss costs by setting outer edge costs to $\gamma_{(i,\ell_i)} = \text{cost}_i / s_i$. We next show how to derive upper and lower bounds from this min-cost flow representation.

Theorem 1 (**FOO bounds OPT**). *For FOO-L and FOO-U from Definition 2,*

$$\boxed{\text{FOO-L} \le \text{OPT} \le \text{FOO-U}} \tag{5}$$

Proof. We observe that f_i as defined in Definition 2, defines the number of bytes "not stored" in the cache. f_i corresponds to the i-th decision variable x_i from Definition 1 as $x_i = (1 - f_i/s_i)$.

(FOO-L $\leq$ OPT): FOO-L is a feasible solution for the LP relaxation of Definition 1, because a total amount of flow s_i needs to flow from node i to node ℓ_i (by definition of β_i). At most $\mu_{(i,i+1)} = C$ flows uses an inner edge which enforces constraint Eq. (2). FOO-L is an optimal solution because it minimizes the total cost of flow along outer edges. Each outer edge's cost is $\gamma_{(i,\ell_i)} = 1/s_i$, so $\gamma_{(i,\ell_i)} f_i = (1 - x_i)$, and thus

$$\text{FOO-L} = \min\Big\{\sum_{i\in I} \gamma_{(i,\ell_i)} f_i\Big\} = \min\Big\{\sum_{i\in I}(1 - x_i)\Big\} \leq \text{OPT} \tag{6}$$

(OPT $\leq$ FOO-U): After rounding, each outer edge (i, ℓ_i) has flow $f_i \in \{0, s_i\}$, so the corresponding decision variable $x_i \in \{0, 1\}$. FOO-U thus yields a feasible integer solution, and OPT yields no more misses than any feasible solution. □

5 FOO IS ASYMPTOTICALLY OPTIMAL

This section proves that FOO is asymptotically optimal, namely that the gap between FOO-U and FOO-L vanishes as the number of objects grows large. Section 5.1 formally states this result and our assumptions, and Sections 5.2–5.5 present the proof.

5.1 Main result and assumptions

Our proof of FOO's optimality relies on two assumptions: *(i)* that the trace is created by stochastically independent request processes and *(ii)* that the popularity distribution is not concentrated on a finite set of objects as the number of objects grows.

Assumption 2 (**Independence**). *The request sequence is generated by independently sampling from a popularity distribution $\mathcal{P}^M$. Object sizes are sampled from an arbitrary size distribution $\mathcal{S}$, which is independent of M and has a finite* $\max_i s_i$.

The assumptions on the object sizes can be relaxed further as we only require the existence of a scaling regime, i.e., that the number of cached objects grows large. For the same reason, we exclude trivial cases where a finite set of objects dominates the request sequence even as the total universe of objects grows large:

Assumption 3 (**Diverging popularity distribution**). *For any number $M > 0$ of objects, the popularity distribution $\mathcal{P}^M$ is defined via an infinite sequence ψ_k. At any time $1 \leq i \leq N$,*

$$\mathbb{P}\left[\textit{object } k \textit{ is requested} \mid M \textit{ objects overall}\right] = \frac{\psi_k}{\sum_{k=1}^{M} \psi_k} \tag{7}$$

The sequence ψ_k must be positive and diverging such that cache size $C \longrightarrow \infty$ is required to achieve a constant miss ratio as $M \longrightarrow \infty$.

Our assumptions on $\mathcal{P}^M$ allow for many common distributions, such as uniform popularities ($\psi_k = 1$) or heavy-tailed Zipfian probabilities ($\psi_k = 1/k^\alpha$ for $\alpha \leq 1$, as is common in practice [16, 19, 48, 62, 78]). Moreover, with some change to notation, our proofs can be extended to require only that ψ_k remains constant over short timeframes. With these assumptions in place, we are now ready to state our main result on FOO's asymptotic optimality.

Theorem 2 (**FOO is Asymptotically Optimal**). *Under Assumptions 2 and 3, for any error ε and violation probability κ, there exists an M^* such that for any trace with $M > M^*$ objects*

$$\mathbb{P}\left[\textit{FOO-U} - \textit{FOO-L} \geq \varepsilon\, N\right] \leq \kappa \tag{8}$$

where the trace length $N \geq M \log^2 M$ and the cache capacity C is scaled with M such that FOO-L's miss ratio remains constant.

Theorem 2 states that, as $M \to \infty$, FOO's *miss ratio error* is almost surely less than ε for any $\varepsilon > 0$. Since FOO-L and FOO-U bound OPT (Theorem 1), FOO-L = OPT = FOO-U.

The rest of this section is dedicated to the proof Theorem 2. The key idea in our proof is to bound the number of non-integer solutions in FOO-L via a precedence relation that forces FOO-L to strictly prefer some decision variables over others, which forces them to be integer. Section 5.2 introduces this precedence relation. Section 5.3 maps this relation to a representation that can be stochastically analyzed (as a variant of the coupon collector problem). Section 5.4 then shows that almost all decision variables are part of a precedence relation and thus integer, and Section 5.5 brings all these parts together in the proof of Theorem 2.

5.2 Bounding the number of non-integer solutions using a precedence graph

This section introduces the precedence relation $\prec$ between caching intervals. The intuition behind $\prec$ is that if an interval i is nested entirely within interval j, then min-cost flow must prefer i over j. We first formally define $\prec$, and then state the property about optimal policies in Theorem 3.

Definition 3 (**Precedence relation**). *For two caching intervals $[i, \ell_i)$ and $[j, \ell_j)$, let the relation $\prec$ be such that $i \prec j$ ("i takes precedence over j") if*

(1) $j < i$,
(2) $\ell_j > \ell_i$, and
(3) $s_i \le s_j$.

The key property of $\prec$ is that it forces integer decision variables.

Theorem 3 (**Precedence forces integer decisions**). *If $i \prec j$, then there exists a min-cost flow solution in which $x_j > 0$ implies $x_i = 1$.*

In other words, if interval i is strictly preferable to interval j, then FOO-L will take all of i before taking any of j. The proof of this result relies on the notion of a residual MCF graph [3, p.304 ff], where for any edge $(i, j) \in G$ with positive flow, we add a backwards edge (j, i) with cost $\gamma_{j,i} = -\gamma_{i,j}$.

Proof. Let G' be the residual MCF graph induced by a given MCF solution. Figure 6 sketches the MCF graph in the neighborhood of $j, \ldots, i, \ldots, \ell_i, \ldots, \ell_j$.

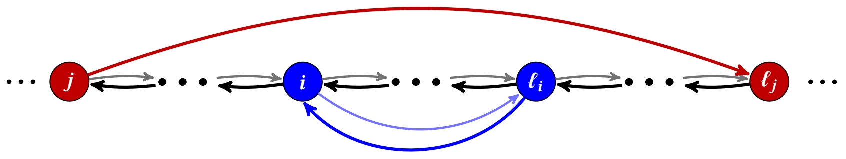


Fig. 6. The precedence relation $i \prec j$ from Definition 3 forces integer decisions on interval i. In any min-cost flow solution, we can reroute flow such that if $x_j > 0$ then $x_i = 1$.

Assume that $x_j > 0$ and that $x_i < 1$, as otherwise the statement is trivially true. Because $x_j > 0$ there exist backwards inner edges all the way between ℓ_j and j. Because $x_i < 1$, the MCF solution must include some flow on the outer edge (i, ℓ_i), and there exists a backwards outer edge $(\ell_i, i) \in G'$ with cost $\gamma_{\ell_i,i} = -1/s_i$.

We can use the backwards edges to create a cycle in G', starting at j, then following edge (j, ℓ_j), backwards inner edges to ℓ_i, the backwards outer edge (ℓ_i, i), and finally backwards inner edges to return to j. Figure 6 highlights this clockwise path in darker colored edges.

This cycle has cost $= 1/s_j - 1/s_i$, which is zero or negative because $s_i \leq s_j$ (by definition of $\prec$ and since $i \prec j$). As negative-cost cycles cannot exist in a MCF solution [3, Theorem 9.1, p.307], the cost of this cycle must be zero. Hence, routing flow along this cycle does not change the solution's cost and maintains all capacity constraints.

Now, we reroute flow along the cycle, "draining" x_j to "fill" x_i. We must consider how much we can drain x_j, i.e., the remaining capacity along j's outer edge, $s_j - f_j = s_j x_j$; and how much we can fill in x_i, i.e., the flow routed along i's outer edge, $f_i = s_i(1 - x_i)$. After routing maximal flow along this cycle, two cases can occur:

- $(s_j - f_j \leq f_i)$: In this case, there is less capacity along j's outer edge than flow along i's outer edge, so we are going to fully drain x_j before filling x_i. Thus, $x_j = 0$.
- $(s_j - f_j > f_i)$: In this case, there is less flow along i's outer edge than capacity along j's outer edge, so we are going to fully fill x_i before draining x_j. Thus, $x_i = 1$.

Hence, either $x_j = 0$ or $x_i = 1$, and the Theorem's implication that $x_j > 0 \Rightarrow x_i = 1$ holds. □

Throughout the remainder of the paper, we consider an MCF solution where the implication in Theorem 3 holds for all $i \prec j$. One could always construct this solution by repeatedly applying the procedure in the above proof. Hence, while FOO's MCF solution may not obey this property, it always yields the same cost as one that does.

To quantify how many integer solutions there are, we need to know the general structure of the precedence graph. Intuitively, for large production traces with many overlapping intervals, the graph will be very dense. We have empirically verified this for our production traces.

Unfortunately, the combinatorial nature of caching traces made it difficult for us to characterize the general structure of the precedence graph under stochastic assumptions. For example, we considered classical results on random graphs [56] and the concentration of measure in random partial orders [18]. None of these paths yielded sufficiently tight bounds on FOO. Instead, we bound the number of non-integer solutions via the generalized coupon collector problem.

5.3 Relating the precedence graph to the coupon collector problem

We now translate the problem of intervals without child in the precedence graph (i.e., intervals i for which there exists no $i \prec j$) into a tractable stochastic representation.

We first describe the intuition for equal object sizes, and then consider variable object sizes.

Definition 4 (**Cached objects**). *Let H_i denote the set of cached intervals that overlap time i, excluding i.*

$$H_i = \left\{ j \neq i : \quad x_j > 0 \text{ and } i \in [j, \ell_j) \right\} \qquad \text{and} \qquad h_i = |H_i| \tag{9}$$

We observe that interval $[i, \ell_i)$ is without child if and only if *all other objects* $x_j \in H_i$ are *requested at least once* in $[i, \ell_i)$. Figure 7 shows an example where H_i consists of five objects (intervals $x_a, \ldots, x_e$). As all five objects are requested before ℓ_i, all five intervals end before x_i ends, and so $[i, \ell_i)$ cannot fit in any of them. To formalize this observation, we introduce the following

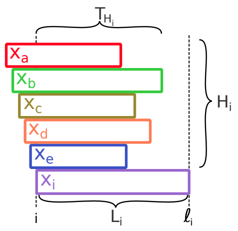


Fig. 7. Simplified notation for the coupon collector representation of offline caching with equal object sizes. We translate the precedence relation from Theorem 3 into the relation between two random variables. L_i denotes the length of interval i. T_{H_i} is the coupon collector time, where we wait until all objects that are cached at the beginning of L_i (H_i denotes these objects) are requested at least once.

random variables, also illustrated in Figure 7. L_i is the length of the i-th interval, i.e., $L_i = \ell_i - i$. T_{H_i} is the time after i when all intervals in H_i end. We observe that T_{H_i} is the stopping time in a *coupon-collector problem* (CCP) where we associate a coupon type with every object in H_i. With equal object sizes, the event $\{i \text{ has a child}\}$ is equivalent to the event $\{T_{H_i} > L_i\}$.

We now extend our intuition to the case of variable object sizes. We now need to consider that objects in H_i can be smaller than s_i and thus may not be i's children for a new reason: the precedence relation (Definition 3) requires i's children to have size larger than or equal to s_i. Figure 8 shows an example where L_i is without child because *(i)* x_b, which ends after ℓ_i, is smaller than s_i, and *(ii)* all larger objects (x_a, x_c, x_d) are requested before ℓ_i. The important conclusion is that, by ignoring the smaller objects, we can reduce the problem back to the CCP.

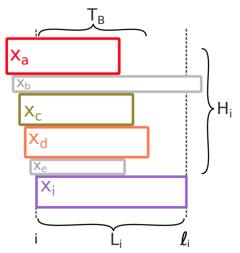


Fig. 8. Full notation for the coupon collector representation of offline caching. With variable object sizes, we need to ignore all objects with a smaller size than s_i (greyed out intervals x_b and x_e). We then define the coupon collector time T_B among a subset $B \subset H_i$ of cached objects with a size larger than or equal to s_i. Using this notation, the event $T_B > L_i$ implies that x_i has a child, which forces x_i to be integer by Theorem 3.

To formalize our observation about the relation to the CCP, we introduce the following random variables, also illustrated in Figure 8. We define B, which is a subset of the cached objects H_i with a size equal to or larger than s_i, and the coupon collector time T_B for B-objects. These definitions are useful as the event $\{T_B > L_i\}$ implies that i has a child and thus x_i is integer, as we now show.

Theorem 4 (**Stochastic bound on non-integer variables**). *For decision variable x_i, $i \in \{1 \dots N\}$, assume that $B \subseteq H_i$ is a subset of cached objects where $s_j \geq s_i$ for all $j \in B$. Further, let the random variable T_B denote the time until all intervals in B end, i.e., $T_B = \max_{j \in B} \ell_j - i$.*

If B is non-empty, then the probability that x_i is non-integer is upper bounded by the probability interval i ends after all intervals in B, i.e.,

$$\mathbb{P}[0 < x_i < 1] \leq \mathbb{P}[L_i > T_B] \quad . \tag{10}$$

The proof works backwards by assuming that $T_B > L_i$. We then show that this implies that there exists an interval j with $i \prec j$ and then apply Theorem 3 to conclude that x_i is integer. Finally, we use this implication to bound the probability.

Proof. Consider an arbitrary $i \in \{1 \dots N\}$ with $T_B > L_i$. Let $[j, \ell_j)$ denote the interval in B that is last requested, i.e., $\ell_j = \max_{k \in B} \ell_k$ (j exists because B is non-empty). To show that $i \prec j$, we check the three conditions of Definition 3.

(1) $j < i$, because $j \in H_i$ (i.e., j is cached at time i);
(2) $\ell_j > \ell_i$, because $\ell_j = \max_{k \in B} \ell_k = i + T_B > i + L_i = \ell_i$; and
(3) $s_i \leq s_j$, because $j \in B$ (i.e., s_j is bigger than s_i by assumption).

Having shown that $i \prec j$, we can apply Theorem 3, so that $x_j > 0$ implies $x_i = 1$. Because $j \in H_i$, j is cached $x_j > 0$ and thus $x_i = 1$. Finally, we observe that $L_i \neq T_B$ and conclude the theorem's statement by translating the above implications, $x_i = 1 \Leftarrow i \prec j \Leftarrow T_B > L_i$, into probability.

$$\mathbb{P}[0 < x_i < 1] = 1 - \mathbb{P}[x_i \in \{0,1\}] \leq 1 - \mathbb{P}[i \prec j] \leq 1 - \mathbb{P}[T_B > L_i] = \mathbb{P}[L_i > T_B] \quad . \tag{11}$$

□

Theorem 4 simplifies the analysis of non-integer x_i to the relation of two random variables, L_i and T_B. While L_i is geometrically distributed, T_B's distribution is more involved.

We map T_B to the stopping time $T_{b,p}$ of a generalized CCP with $b = |B|$ different coupon types. The coupon probabilities p follow from the object popularity distribution $\mathcal{P}^M$ by conditioning on objects in B. As the object popularities p are not equal in general, characterizing the stopping time $T_{b,p}$ is much more challenging than in the classical CCP, where the coupon probabilities are assumed to be equal. We solve this problem by observing that collecting b coupons under equal probabilities stops faster than under p. This fact may appear obvious, but it was only recently shown by Anceaume et al. [5, Theorem 4, p. 415] (the proof is non-trivial). Thus, we can use a classical CCP to bound the generalized CCP's stopping time and T_B.

Lemma 1 (**Connection to classical coupon connector problem**). *For any object popularity distribution $\mathcal{P}^M$, and for $q = (1/b, \ldots, 1/b)$, using the notation from Theorem 4*

$$\mathbb{P}\left[T_B < l\right] \le \mathbb{P}\left[T_{b,q} < l\right] \qquad \text{for any} \qquad l \ge 0 \quad . \tag{12}$$

The proof of this Lemma simply extends the result by Anceaume et al. to account for requests to objects not in B. We state the full proof in Section A.2.

5.4 Typical objects almost always lead to integer decision variables

We now use the connection to the coupon collector problem to show that almost all of FOO's decision variables are integer. Specifically, we exclude a small number of very large and unpopular objects, and show that the remaining objects are almost always part of a precedence relation, which forces the corresponding decision variables to become integer. In Section 5.5, we show that the excluded fraction is diminishingly small.

We start with a definition of the sets of large objects and the set of popular objects.

Definition 5 (**Large objects and popular objects**). *Let N^* be the time after which the cache needs to evict at least one object. For time $i \in \{N^* \ldots N\}$, we define the sets of large objects, B_i, and the set of popular objects, F_i.*

The set $B_i \subseteq H_i$ consists of the requests to the fraction δ largest objects of H_i ($0 < \delta < 1$). We also define $b_i = |B_i|$, and we write "$s_i \le B_i$" if $s_i \le s_j$ for all $j \in B_i$ and "$s_i \not\le B_i$" otherwise.

The set F_i consists of those objects k with a request probability ρ_k which lies above the following threshold.

$$F_i = \left\{ k : \rho_k \ge \frac{1}{b_i \log\log b_i} \right\} \tag{13}$$

Using the above definitions, we prove that "typical" objects (i.e., popular objects that are not too large) rarely lead to non-integer decision variables as the number of objects M grows large.

Theorem 5 (**Typical objects are rarely non-integer**). *For $i \in \{N^* \ldots N\}$, B_i, and F_i from Definition 5,*

$$\mathbb{P}\left[0 < x_i < 1 \,\middle|\, s_i \le B_i, \sigma_i \in F_i\right] \to 0 \text{ as } M \to \infty \quad . \tag{14}$$

The intuition is that, as the number of cached objects grows large, it is vanishingly unlikely that *all* objects in B_i will be requested before a *single* object is requested again. That is, though there are not many large objects in B_i, there are enough that, following Theorem 4, x_i is vanishingly unlikely to be non-integer. The proof of Theorem 5 uses elementary probability but relies on several very technical proofs. We sketch the proof here, and state the formal proofs in the Appendices A.3 to A.5.

Proof sketch. Following Theorem 4, it suffices to consider the event $\{L_i > T_{B_i}\}$.

- ($\mathbb{P}\left[L_i > T_{B_i}\right] \longrightarrow 0$ as $h_i \longrightarrow \infty$): We first condition on $L_i = l$, so that L_i and T_{B_i} become stochastically independent and we can bound $\mathbb{P}\left[L_i > T_{B_i}\right]$ by bounding either $\mathbb{P}\left[L_i > l\right]$ or $\mathbb{P}\left[l > T_{B_i}\right]$. Specifically, we split l carefully into "small l" and "large l", and then show that L_i is concentrated at small l, and T_{B_i} is concentrated at large l. Hence, $\mathbb{P}\left[L_i > T_{B_i}\right]$ is negligible.
  - (Small l:) For small l, we show that it is unlikely for all objects in B_i to have been requested after l requests. We upper bound the distribution of T_{B_i} with $T_{b_i,u}$ (Lemma 1). We then show that the distribution of $T_{b_i,u}$ decays exponentially at values below its expectation (Lemma 4). Hence, for l far below the expectation of $T_{b_i,u}$, the probability vanishes $\mathbb{P}\left[T_{b_i,u} < l\right] \longrightarrow 0$, so long as $b_i = \delta h_i$ grows large, which it does because h_i grows large.
  - (Large l:) For large l, $\mathbb{P}\left[L_i > l\right] \longrightarrow 0$ because we only consider popular objects $\sigma_i \in F_i$ by assumption, and it is highly unlikely that a popular object is not requested after many requests.
- ($h_i \longrightarrow \infty$): What remains to be shown is that the number of cached objects h_i actually grows large. Since the cache size $C \longrightarrow \infty$ as $M \longrightarrow \infty$ by Assumption 3, this may seem obvious. Nevertheless, it must be demonstrated (Lemma 3). The basic intuition is that h_i is almost never much less than $h^* = C/\max_k s_k$, the fewest number of objects that could fit in the cache, and $h^* \longrightarrow \infty$ as $C \longrightarrow \infty$.

  To see why h_i is almost never much less than h^*, consider the probability that $h_i < x$, where x is constant with respect to M. For any x, take large enough M such that $x < h^*$.

  Now, in order for $h_i < x$, almost all requests must go to distinct objects. Any object that is requested twice between u (the last time where $h_u \geq h^*$) and v (the next time where $h_v \geq h^*$) produces an interval (see Figure 9). This interval is guaranteed to fit in the cache, since $h_i < x < h^*$ means there is space for an object of any size. As h^* and M grow further, the amount of cache resources that must lay unused for $h_i < x$ grows further and further, and the probability that no interval fits within these resources becomes negligible.

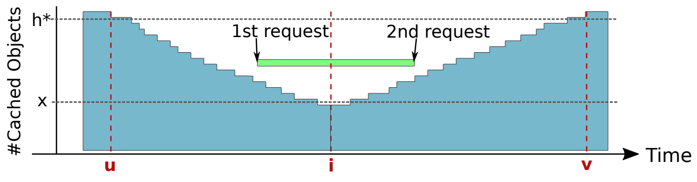


Fig. 9. The number of cached objects at time i, h_i, is unlikely to be far below $h^* = C/\max_k s_k$, the fewest number of objects that fit in the cache. In order for $h_i < h^*$ to happen, no other interval must fit in the white triangular space centered at i (otherwise FOO-L would cache the interval).

□

5.5 Bringing it all together: Proof of Theorem 2

This section combines our results so far and shows how to obtain Theorem 2: *There exists M^* such that for any $M > M^*$ and for any error ε and violation probability κ,*

$$\mathbb{P}\left[\text{FOO-U} - \text{FOO-L} \geq \varepsilon\, N\right] \leq \kappa \tag{15}$$

Proof of Theorem 2. We start by bounding the cost of non-integer solutions by the number of non-integer solutions, Ω.

$$\sum_{\{i:\, 0<x_i<1\}} x_i \leq \sum_{i \in I} \mathbb{1}_{\{i:\, 0<x_i<1\}} = \Omega \tag{16}$$

It follows that (FOO-U – FOO-L) $\leq \Omega$.

$$\mathbb{P}\left[\text{FOO-U} - \text{FOO-L} \geq \varepsilon\, N\right] \leq \mathbb{P}\left[\Omega \geq \varepsilon\, N\right] \tag{17}$$

We apply the Markov inequality.

$$\leq \frac{\mathbb{E}\left[\Omega\right]}{N\,\varepsilon} = \frac{1}{N\,\varepsilon}\sum_{i \in I} \mathbb{P}\left[0 < x_i < 1\right] \tag{18}$$

There are at most N terms in the sum.

$$\leq \frac{\mathbb{P}\left[0 < x_i < 1\right]}{\varepsilon} \tag{19}$$

To complete Eq. (15), we upper bound $\mathbb{P}\left[0 < x_i < 1\right]$ to be less than $\varepsilon\,\kappa$. We first condition on $s_i \leq B_i$ and $\sigma_i \in F_i$, double counting those i where $s_i \not\leq B_i$ and $\sigma_i \notin F_i$.

$$\mathbb{P}\left[0 < x_i < 1\right] \leq \mathbb{P}\left[0 < x_i < 1 \,\middle|\, s_i \leq B_i, \sigma_i \in F_i\right] \mathbb{P}\left[s_i \leq B_i, \sigma_i \in F_i\right] \tag{20}$$

$$+ \mathbb{P}\left[0 < x_i < 1 \,\middle|\, s_i \not\leq B_i\right] \mathbb{P}\left[s_i \not\leq B_i\right] \tag{21}$$

$$+ \mathbb{P}\left[0 < x_i < 1 \,\middle|\, \sigma_i \notin F_i\right] \mathbb{P}\left[\sigma_i \notin F_i\right] \tag{22}$$

Drop ≤ 1 terms.

$$\leq \mathbb{P}\left[0 < x_i < 1 \,\middle|\, s_i \leq B_i, \sigma_i \in F_i\right] + \mathbb{P}\left[s_i \not\leq B_i\right] + \mathbb{P}\left[i \notin F_i\right] \tag{23}$$

To bound $\mathbb{P}\left[0 < x_i < 1\right] \leq \varepsilon\,\kappa$, we choose parameters such that each term in Eq. (23) is less than $\varepsilon\,\kappa/3$. The first term vanishes by Theorem 5. The second term is satisfied by choosing $\delta = \varepsilon\,\kappa/3$ (Definition 5). For the third term, the probability that any cached object is unpopular vanishes as h_i grows large.

$$\mathbb{P}\left[i \notin F_i\right] \leq \frac{h_i}{b_i \log\log b_i} = \frac{3}{\varepsilon\,\kappa \log\log \varepsilon\,\kappa\, h_i/3} \longrightarrow 0 \text{ as } h_i \longrightarrow \infty \tag{24}$$

Finally, we choose M^* large enough that the first and third terms in Eq. (23) are each less than $\varepsilon\,\kappa/3$. □

This concludes our theoretical proof of FOO's optimality.

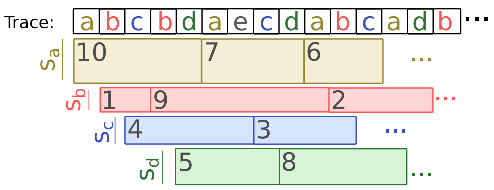
6 PRACTICAL FLOW-BASED OFFLINE OPTIMAL FOR REAL-WORLD TRACES

While FOO is asymptotically optimal and very accurate in practice, as well as faster than prior approximation algorithms, it is still not fast enough to process production traces with hundreds of millions of requests in a reasonable timeframe. We now use the insights gained from FOO's graph-theoretic formulation to design new upper and lower bounds on OPT, which we call *practical flow-based offline optimal (PFOO)*. We provide the first practically useful lower bound, PFOO-L, and an upper bound that is much tighter than prior practical offline upper bounds, PFOO-U:

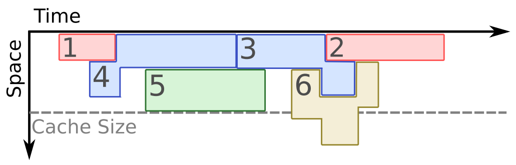
$$\boxed{\text{PFOO-L} \leq \text{FOO-L} \leq \text{OPT} \leq \text{FOO-U} \leq \text{PFOO-U}} \tag{25}$$

6.1 Practical lower bound: PFOO-L

PFOO-L considers the *total resources* consumed by OPT. As Figure 10a illustrates, cache resources are limited in both space and time [10]: measured in resources, the cost to cache an object is the product of *(i)* its size and *(ii)* its reuse distance (i.e., the number of accesses until it is next requested). On a trace of length N, a cache of size C has total resources $N \times C$. The objects cached by OPT, or any other policy, cannot cost more total resources than this.



(a) Intervals sorted by resource cost = reuse distance × object size.



(b) PFOO-L greedily claims the smallest intervals while not exceeding the average cache size.

Fig. 10. PFOO's lower bound, PFOO-L, constrains the total resources used over the full trace (i.e., size × time). PFOO-L claims the hits that require fewest resources, allowing cached objects to temporarily exceed the cache capacity.

Definition of PFOO-L. PFOO-L sorts all intervals by their resource cost, and caches the smallest-cost intervals up a total resource usage of $N \times C$. Figure 10b shows PFOO-L on a short request trace. By considering only the total resource usage, PFOO-L ignores other constraints that are faced by caching policies. In particular, PFOO-L does not guarantee that cached intervals take less than C space at all times, as shown by interval 6 for object **a** in Figure 10b, which exceeds the cache capacity during part of its interval.

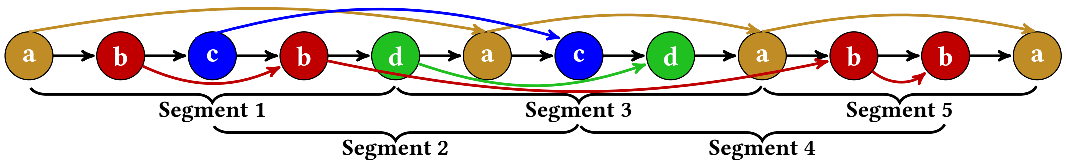
Why PFOO-L works. PFOO-L is a lower bound because no policy, including OPT or FOO-L, can get fewer misses using $N \times C$ total resources. It gives a reasonably tight bound because, on large caches with many objects, the distribution of interval costs is similar throughout the request trace. Hence, for a given cache capacity, the "marginal interval" (i.e., the one barely does not fit in the cache under OPT) is also of similar cost throughout the trace. Informally, PFOO-L caches intervals up to this marginal cost, and so rarely exceeds cache capacity by very much. The intuition behind PFOO-L is thus similar to the widely used Che approximation [21], which states that LRU caches keep objects up until some "characteristic age". But unlike the Che approximation, which has unbounded error, PFOO-L uses this intuition to produce a robust lower bound. This intuition holds particularly when requests are largely independent, as in our proof assumptions and in traces from CDNs or other Internet services. However, as we will see, PFOO-L introduces modest error even on other workloads where these assumptions do not hold.

PFOO-L uses a similar notion of "cost" as Belady-Size, but provides two key advantages. First, PFOO-L is closer to OPT. Relaxing the capacity constraint lets PFOO-L avoid the pathologies discussed in Section 2.4, since PFOO-L can temporarily exceed the cache capacity to retain valuable objects that Belady-Size is forced to evict. Second, relaxing the capacity constraint makes PFOO-L a *lower* bound, giving the first reasonably tight lower bound on long traces.

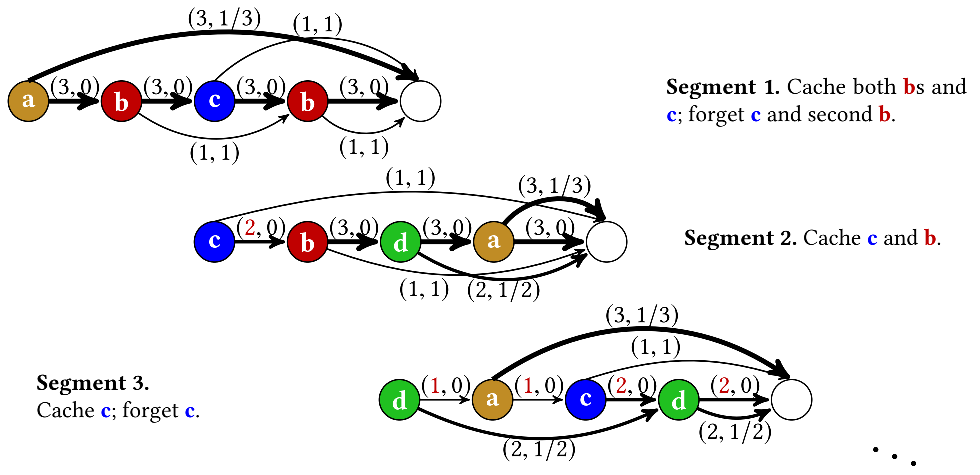
6.2 Practical upper bound: PFOO-U

Definition of PFOO-U. PFOO-U breaks FOO's min-cost flow graph into smaller segments of constant size, and then solves each segment using min-cost flow incrementally. By keeping track of the resource usage of already solved segments, PFOO-U yields a globally-feasible solution, which is an upper bound on OPT or FOO-U.

Example of PFOO-U. Figure 11 shows our approach on the trace from Figure 2 for a cache capacity of 3. At the top is FOO's full min-cost flow problem; for large traces, this MCF problem is too expensive to solve directly. Instead, PFOO-U breaks the trace into segments and constructs a min-cost flow problem for each segment.



(a) PFOO-U breaks the trace into small, overlapping segments . . .


(3, 1/3)
(1, 1)
(3, 0)
(1, 1)
Segment 1. Cache both bs and c; forget c and second b.
(2, 0)
(3, 1/3)
(2, 1/2)
Segment 2. Cache c and b.
Segment 3. Cache c; forget c.
(1, 0)
(2, 0)
(2, 1/2)

(b) . . . and solves min-cost flow for each segment.

Fig. 11. Starting from FOO's full formulation, PFOO-U breaks the min-cost flow problem into overlapping segments. Going left-to-right through the trace, PFOO-U optimally solves MCF on each segment, and updates link capacities in subsequent segments to maintain feasibility for all cached objects. The segments overlap to capture interactions across segment boundaries.

PFOO-U begins by solving the min-cost flow for the first segment. In this case, the solution is to cache both **b**s, **c**, and one-third of **a**, since these decisions incur the minimum cost of two-thirds, i.e., less than one cache miss. As in FOO-U, PFOO-U rounds down the non-integer decision for **a** and all following non-integer decisions. Furthermore, PFOO-U only fixes decisions for objects in the first half of this segment. This is done to capture interactions between intervals that cross segment boundaries. Hence, PFOO-U "forgets" the decision to cache **c** and the second **b**, and its final decision for this segment is only to cache the first **b** interval.

PFOO-U then updates the second segment to account for its previous decisions. That is, since **b** is cached until the second request to **b**, capacity must be removed from the min-cost flow to reflect this allocation. Hence, the capacity along the inner edge **c** $\rightarrow$ **b** is reduced from 3 to 2 in the second segment (**b** is size 1). Solving the second segment, PFOO-U decides to cache **c** and **b** (as well as half of **d**, which is ignored). Since these are in the first half of the segment, we fix both decisions, and move onto the third segment, updating the capacity of edges to reflect these decisions as before.

PFOO-U continues to solve the following segments in this manner until the full trace is processed. On the trace from Section 3, it decides to cache all requests to **b** and **c**, yielding 5 misses on the requests to a and d. These are the same decisions as taken by FOO-U and OPT. We generally find that PFOO-U yields nearly identical miss ratios as FOO-U, as we next demonstrate on real traces.

6.3 Summary

Putting it all together, PFOO provides efficient lower and upper bounds on OPT with variable object sizes. PFOO-L runs in $O(N \log N)$ time, as required to sort the intervals; and PFOO-U runs in $O(N)$ because it divides the min-cost flow into segments of constant size. In practice, PFOO-L is faster than PFOO-U at realistic trace lengths, despite its worse asymptotic runtime, due to the large constant factor in solving each segment in PFOO-U.

PFOO-L gives a lower bound (PFOO-L $\leq$ FOO-L, Eq. (25)) because PFOO-L caches all of the smallest intervals, so no policy can get more hits (i.e., cache more intervals) without exceeding $N \times C$ overall resources. Resources are a hard constraint obeyed by all policies, including FOO-L, whose resources are constrained by the capacity C of all $N - 1$ inner edges.

PFOO-U gives an upper bound (FOO-U $\leq$ PFOO-U, Eq. (25)) because FOO-U and PFOO-U both satisfy the capacity constraint and the integrality constraint of the ILP formulation of OPT (Definition 1). By sequentially optimizing the MCF segment after segment, we induce additional constraints on PFOO-U, which can lead to suboptimal solutions.

7 EXPERIMENTAL METHODOLOGY

We evaluate FOO and PFOO against prior offline bounds and online caching policies on eight different production traces.

7.1 Trace characterization

We use production traces from three global content-distribution networks (CDNs), two web-applications from different anonymous large Internet companies, and storage workloads from Microsoft [79]. We summarize the trace characteristics in Table 3. The table shows that our traces typically span several tens to hundreds of million of requests and tens of millions of objects. Figure 12 shows four key distributions of these workloads.

Trace	Year	# Requests	# Objects	Object sizes
CDN 1	2016	500 M	18 M	10 B – 616 MB
CDN 2	2015	440 M	19 M	1 B – 1.5 GB
CDN 3	2015	420 M	43 M	1 B – 2.3 GB
WebApp 1	2017	104 M	10 M	3 B – 1.9 MB
WebApp 2	2016	100 M	14 M	5 B – 977 KB
Storage 1	2008	29 M	16 M	501 B – 780 KB
Storage 2	2008	37 M	6 M	501 B – 78 KB
Storage 3	2008	45 M	14 M	501 B – 489 KB

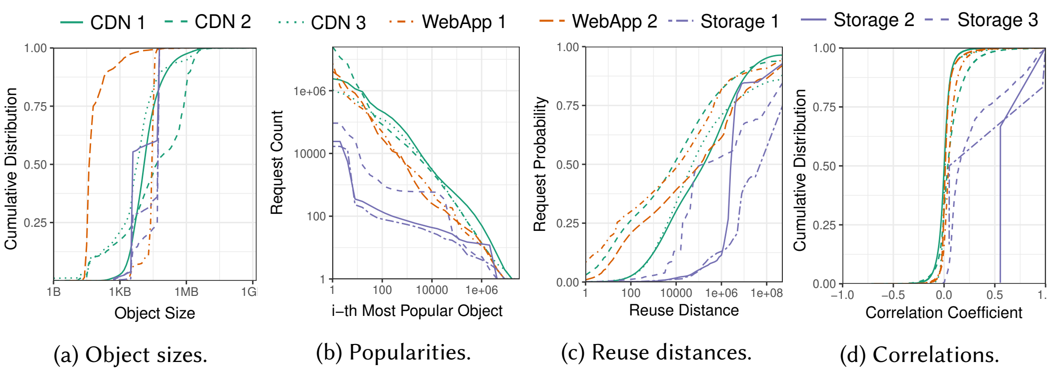
Table 3. Overview of key properties of the production traces used in our evaluation.

The object size distribution (Figure 12a) shows that object sizes are variable in all traces. However, while they span almost ten orders of magnitude in CDNs, object sizes vary only by six orders of magnitude in web applications, and only by three orders of magnitude in storage systems. WebApp 1 also has noticeably smaller object sizes throughout, as is representative for application-cache workloads.

The popularity distribution (Figure 12b) shows that CDN workloads and WebApp workloads all follow approximately a Zipf distribution with α between 0.85 and 1. In contrast, the popularity distribution of storage traces is much more irregular with a set of disproportionally popular objects, an approximately log-linear middle part, and an exponential cutoff for the least popular objects.

The reuse distance distribution (Figure 12c) — i.e., the distribution of the number of requests between requests to the same object — further distinguishes CDN and WebApp traces from storage workloads. CDNs and WebApps serve millions of different customers and so exhibit largely independent requests with smoothly diminishing object popularities, which matches our proof assumptions. Thus, the CDN and WebApp traces lead to a smooth reuse distance, as shown in the figure. In contrast, storage workloads serve requests from one or a few applications, and so often exhibit highly correlated requests (producing spikes in the reuse distance distribution). For example, scans are common in storage (e.g., traces like: ABCD ABCD ...), but never seen in CDNs. This is evident from the figure, where the storage traces exhibit several steps in their cumulative request probability, as correlated objects (e.g., due to scans) have the same reuse distance.

Finally, we measure the correlation across different objects (Figure 12d). Ideally, we could directly test our independence assumption (Assumption 2). Unfortunately, quantifying independence on real traces is challenging. For example, classical methods such as Hoeffding's independence test [47] only apply to continuous distributions, whereas we consider cache requests in discrete time.



(a) Object sizes. (b) Popularities. (c) Reuse distances. (d) Correlations.

Fig. 12. The production traces used in our evaluation comes from three different domains (CDNs, WebApps, and storage) and thus exhibit starkly different request patterns. (a) The object size distribution in CDN traces space more than nine orders of mangitude, where object sizes in WebApp and Storage traces are much smaller. (b) Object popularities in CDNs and WebApps follow approximately Zipf popularities, whereas the popularity distribution for storage traces is more complex. (c) The reuse distance distribution of CDNs and WebApps is smooth, whereas in the Storage there are jumps – indicating correlated request sequences such as scans. (d) The distribution of Pearson correlation coffecients for the top 10k-most popular objects shows that CDN and WebApp traces do not exhibit linear correlations, whereas the three storage traces show distince patterns with significant positive correlation. One of the storage traces (Storage 2) in fact shows a correlation coefficient greater than 0.5 for all 10k-most popular objects.

We therefore turn to correlation coefficients. Specifically, we use the Pearson correlation coefficient as it is computable in linear time (as opposed to Spearman's rank and Kendall's tau [30]). We define the coefficient based on the number of requests each object receives in a time bucket that spans 4000 requests (we verified that the results do not change significantly for time bucket sizes in the range 400 - 40k requests). In order to capture pair-wise correlations, we chose the top 10k objects in each trace, calculated the request counts for all time buckets, and then calculated the Pearson coefficient for all possible combinations of object pairs.

Figure 12d shows the distribution of coefficients for all object pair combinations. We find that both CDN and WebApps do not show a significant correlation; over 95% of the object pairs have a coefficient coefficient between -0.25 and 0.25. In contrast, we find strong positive correlations in the storage traces. For the first storage trace, we measure a correlation coefficient greater than 0.5 for all 10k-most popular objects. For the second storage trace, we measure a correlation coefficient greater than 0.5 for more than 20% of the object pairs. And, for the third storage trace, we measure a correlation coefficient greater than 0.5 for more than 14% of the object pairs. We conclude that the simple Pearson correlation coefficient is sufficiently powerful to quantify the correlation inherent to storage traces (such as loops and scans).

7.2 Caching policies

We evaluate three classes of policies: theoretical bounds on OPT, practical offline heuristics, and online caching policies. Besides FOO, there exist three other theoretical bounds on OPT with approximation guarantees: OFMA, LocalRatio, and LP (Section 2.3). Besides PFOO-U, we consider three other practical upper bounds: Belady, Belady-Size, and Freq/Size (Section 2.4). Besides PFOO-L, there is only one other practical lower bound: a cache with infinite capacity (Infinite-Cap). Finally, for online policies, we evaluated GDSF [22], GD-Wheel [60], AdaptSize [16], Hyperbolic [17], and several other older policies which perform much worse on our traces (including LRU-K [71], TLFU [34], SLRU [48], and LRU).

Our implementations are in C++ and use the COIN-OR::LEMON library [70], GNU parallel [81], OpenMP [27], and CPLEX 12.6.1.0. OFMA runs in $O(N^2)$, LocalRatio runs in $O(N^3)$, Belady in $O(N \log C)$. We rely on sampling [74] to run Belady-Size on large traces, which gives us an $O(N)$ implementation. We will publicly release our policy implementations and evaluation infrastructure upon publication of this work. To the best of our knowledge, these include the first implementations of the prior theoretical offline bounds.

7.3 Evaluation metrics

We compare each policy's *miss ratio*, which ranges from 0 to 1. We present absolute error as unitless scalars and relative error as percentages. For example, if FOO-U's miss ratio is 0.20 and FOO-L's is 0.15, then absolute error is 0.05 and relative error is 33%.

We present a *miss ratio curve* for each trace, which shows the miss ratio achieved by different policies at different cache capacities. We present these curves in log-linear scale to study a wide range of cache capacities. Miss ratio curves allow us to compare miss ratios achieved by different policies at fixed cache capacities, as well as compute cache capacities that achieve equivalent miss ratios.

8 EVALUATION

We evaluate FOO and PFOO to demonstrate the following: *(i)* PFOO is fast enough to process real traces, whereas FOO and prior theoretical bounds are not; *(ii)* FOO yields nearly tight bounds on OPT, even when our proof assumptions do not hold; *(iii)* PFOO is highly accurate on full production traces; and *(iv)* PFOO reveals that there is significantly more room for improving current caching systems than implied by prior offline bounds.

8.1 PFOO is necessary to process real traces

Figure 13 shows the execution time of FOO, PFOO, and prior theoretical offline bounds at different trace lengths. Specifically, we run each policy on the first N requests of the CDN 1 trace, and vary

N from a few thousand to over 30 million. Each policy ran alone on a 2016 SuperMicro server with 44 Intel Xeon E5-2699 cores and 500 GB of memory.

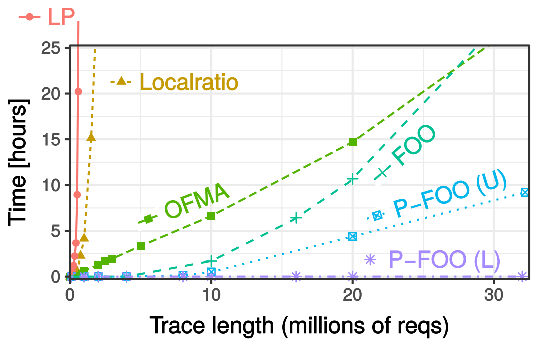


Fig. 13. Execution time of FOO, PFOO, and prior theoretical offline bounds at different trace lengths. Most prior bounds are unusable above 500 K requests. Only PFOO can process real traces with many millions of requests.

These results show that LP and LocalRatio are unusable: they can process only a few hundred thousand requests in a 24-hour period, and their execution time increases rapidly as traces lengthen. While FOO and OFMA are faster, they both take more than 24 hours to process more than 30 million requests, and their execution times increase super-linearly.

Finally, PFOO is much faster and scales well, allowing us to process traces with hundreds of millions of requests. PFOO's lower bound completes in a few minutes, and while PFOO's upper bound is slower, it scales linearly with trace length. *PFOO is thus the only bound that completes in reasonable time on real traces.*

8.2 FOO is nearly exact on short traces

We compare FOO, PFOO, and prior theoretical upper bounds on the first 10 million requests of each trace. Of the prior theoretical upper bounds, only OFMA runs in a reasonable time at this trace length,[4] so we compare FOO, PFOO, OFMA, the Belady variants, Freq/Size, and Infinite-Cap.

Our first finding is that *FOO-U and FOO-L are nearly identical*, as predicted by our analysis. The largest difference between FOO-U's and FOO-L's miss ratio on CDN and WebApp traces is 0.0005—a relative error of 0.15%. Even on the storage traces, where requests are highly correlated and hence our proof assumptions do not hold, the largest difference is 0.0014—a relative error of 0.27%. Compared to the other offline bounds, FOO is at least an order of magnitude and often several orders of magnitude more accurate.

Given FOO's high accuracy, we use FOO to estimate the error of the other offline bounds. Specifically, we assume that OPT lies in the middle between FOO-U and FOO-L. Since the difference between FOO-U and FOO-L is so small, this adds negligible error (less than 0.14%) to all other results.

Figure 14 shows the maximum error from OPT across five cache sizes on our first CDN production trace. All upper bounds are shown with a bar extending above OPT, and all lower bounds are shown with a bar extending below OPT. Note that the practical offline upper bounds (e.g., Belady) do not

[4]While we have tried downsampling the traces to run LP and LocalRatio (as suggested in [11, 57, 84] for objects with equal sizes), we were unable to achieve meaningful results. Under variable object sizes, scaling down the system (including the cache capacity), makes large objects disproportionately disruptive.

have corresponding lower bounds. Likewise, there is no upper bound corresponding to Infinite-Cap. Also note that OFMA's bars are so large than they extend above and below the figure. We have therefore annotated each bar with its absolute error from OPT.

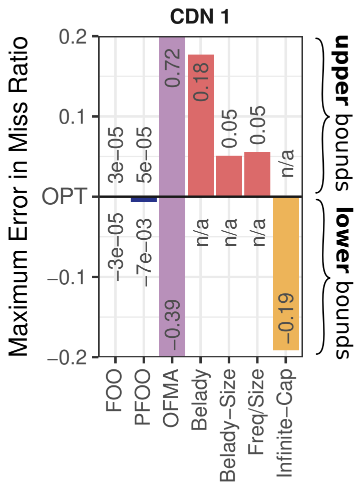


Fig. 14. Comparison of the maximum approximation error of FOO, PFOO, and prior offline upper and lower bounds across five cache sizes on a CDN production trace. FOO's upper and lower bounds are nearly identical, with a gap of less than 0.0005. We therefore assume that OPT is halfway between FOO-U and FOO-L, which introduces negligible error due to FOO's high accuracy. PFOO likewise introduces small error, with PFOO-U having a similar accuracy to FOO-U. PFOO-L leads to higher errors than FOO-L but is still highly accurate, within 0.007. In contrast, all prior policies lead to errors several orders-of-magnitude larger.

The figure shows that FOO-U and FOO-L nearly coincide, with error of 0.00003 (=3e−5) on this trace. PFOO-U is 0.00005 (=5e−5) above OPT, nearly matching FOO-U, and PFOO-L is 0.007 below OPT, which is very accurate though worse than FOO-L.

All prior techniques yield error several orders of magnitude larger. OFMA has very high error: its bounds are 0.72 above and 0.39 below OPT. The practical upper bounds are more accurate than OFMA: Belady is 0.18 above OPT, Belady-Size 0.05, and Freq/Size 0.05. Finally, Infinite-Cap is 0.19 below OPT. Prior to FOO and PFOO, the best bounds for OPT give a broad range of up to 0.24. *PFOO and FOO reduce error by 34× and 4000×, respectively.*

Figures 15 and 16 show the approximation error on all eight production traces. The prior upper bounds are much worse than PFOO-U, except on one trace (Storage 1), where Belady-Size and Freq/Size are somewhat accurate. Averaging across all traces, PFOO-U is 0.0014 above OPT. PFOO-U reduces mean error by 37× over Belady-Size, the best prior upper bound. Prior work gives even weaker lower bounds. PFOO-L is on average 0.004 below OPT on the CDN traces, 0.02 below OPT on the WebApp traces, and 0.04 below OPT on the storage traces. PFOO-L reduces mean error by 9.8× over Infinite-Cap and 27× over OFMA. Hence, across a wide range of workloads, PFOO is by far the best practical bound on OPT.

8.3 PFOO is accurate on real traces

Now that we have seen that FOO is accurate on short traces, we next show that PFOO is accurate on long traces. Figure 17 shows the miss ratio over the full traces achieved by PFOO, the best prior practical upper bounds (the best of Belady, Belady-Size, and Freq/Size), the Infinite-Cap lower bound, and the best online policy (see Section 7).

On average, PFOO-U and PFOO-L bound the optimal miss ratio within a narrow range of 4.2%. PFOO's bounds are tighter on the CDN and WebApp traces than the storage traces: PFOO gives an average bound of just 1.4% on CDN 1-3 and 1.3% on WebApp 1-2 but 5.7% on Storage 1-3. This is likely due to error in PFOO-L when requests are highly correlated, as they are in the storage traces.

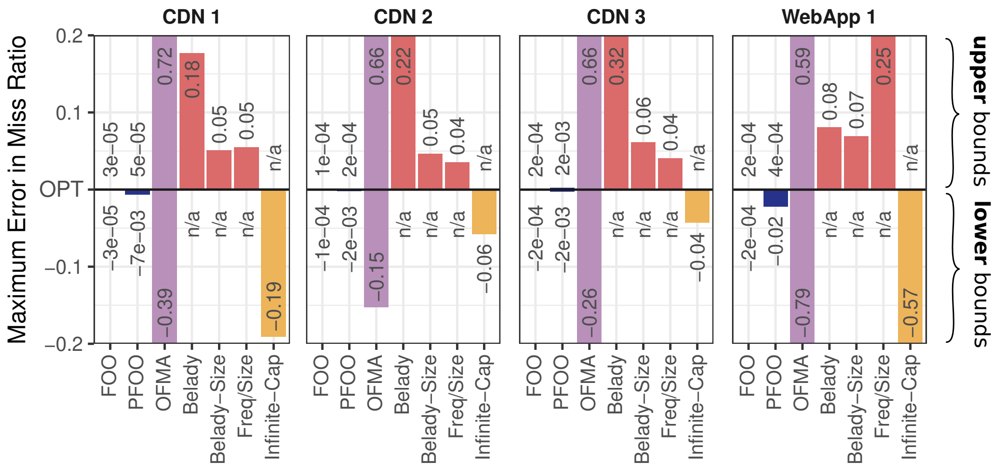


Fig. 15. Approximation error of FOO, PFOO, and several prior offline bounds on four of our eight production traces (Figure 16 shows the other four). FOO and PFOO's lower and upper bounds are orders of magnitude better than any other offline bound. (See Figure 14.)

Nevertheless, PFOO gives much tighter bounds than prior techniques on every trace. The prior offline upper bounds are noticeably higher than PFOO-U. On average, compared to PFOO-U, Belady-Size is 19% higher, Freq/Size is 22% higher, and Belady is fully 72% higher. These prior upper bounds are not, therefore, good proxies for the offline optimal. Moreover, the best upper bound varies across traces: Freq/Size is lower on CDN 1 and CDN 3, but Belady-Size is lower on the others. Unmodified Belady gives a very poor upper bound, showing that caching policies must account for object size. The only lower bound in prior work is an infinitely large cache, whose miss ratio is much lower than PFOO-L. *PFOO thus gives the first reasonably tight bounds on the offline miss ratio for real traces.*

8.4 PFOO shows that there is significant room for improvement in online policies

Finally, we compare with online caching policies. Figure 17 show the best online policy (by average miss ratio) and the offline bounds for each trace. We also show LRU for reference on all traces.

On all traces at most cache capacities, there is a large gap between the best online policy and PFOO-U, showing that there remains significant room for improvement in online caching policies. Moreover, *this gap is much larger than prior offline bounds would suggest.* On average, PFOO-U achieves 27% fewer misses than the best online policy, whereas the best prior offline policy achieves only 7.2% fewer misses; the miss ratio gap between online policies and offline optimal is thus 3.75× as large as implied by prior bounds. The storage traces are the only ones where PFOO does not consistently increase this gap vs. prior offline bounds, but even on these traces there is a large difference at some sizes (e.g., at 64 GB in Figure 17g). On CDN and WebApp traces, the gap is much larger.

For example, on CDN 2, GDSF (the best online policy) matches Belady-Size (the best prior offline upper bound) at most cache capacities. One would therefore conclude that existing online policies are nearly optimal, but PFOO-U reveals that there is in fact a large gap between GDSF and OPT on this trace, as it is 21% lower on average (refer back to Figure 1).

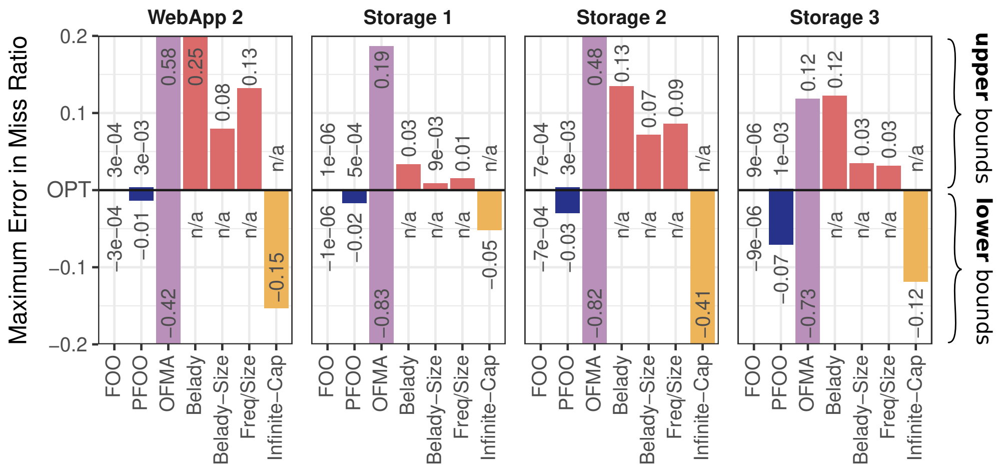


Fig. 16. Approximation error of FOO, PFOO, and several prior offline bounds on four of our eight production traces (Figure 15 shows the other four). FOO and PFOO's lower and upper bounds are orders of magnitude better than any other offline bound. (See Figure 14.)

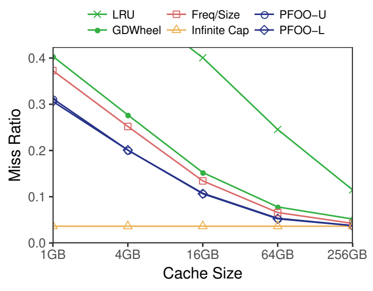
These miss ratio reductions make a large difference in real systems. For example, on CDN 2, CDN 3, and WebApp 1, OPT requires just 16 GB to match the miss ratio of the best prior offline bound at 64 GB (the x-axis in these figures is shown in log-scale). Prior bounds thus suggest that online policies require 4× as much cache space as is necessary.

9 CONCLUSION AND FUTURE WORK

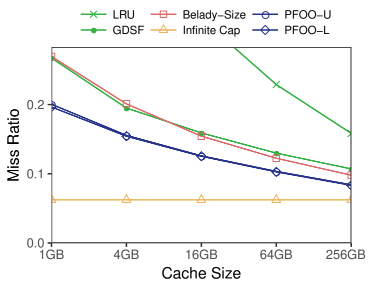
We began this paper by asking: *Should the systems community continue trying to improve miss ratios, or have all achievable gains been exhausted?* We have answered this question by developing new techniques, FOO and PFOO, to accurately and quickly estimate OPT with variable object sizes. Our techniques reveal that prior bounds for OPT lead to qualitatively wrong conclusions about the potential for improving current caching systems. Prior bounds indicate that current systems are nearly optimal, whereas PFOO reveals that misses can be reduced by up to 43%.

Moreover, since FOO and PFOO yield constructive solutions, they offer a new path to improve caching systems. While it is outside the scope of this work, we plan to investigate adaptive caching systems that tune their parameters online by learning from PFOO, e.g., by recording a short window of past requests and running PFOO to estimate OPT's behavior in real time. For example, AdaptSize, a caching system from 2017, assumes that admission probability should decay exponentially with object size. This is unjustified, and we can learn an optimized admission function from PFOO's decisions.

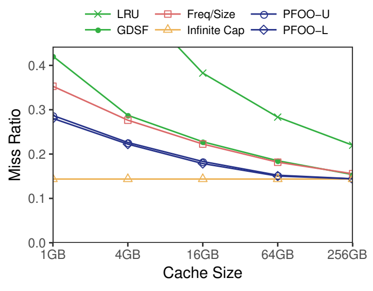
This paper gives the first principled way to evaluate caching policies with variable object sizes: FOO gives the first asymptotically exact, polynomial-time bounds on OPT, and PFOO gives the first practical and accurate bounds for long traces. Furthermore, our results are verified on eight production traces from several large Internet companies, including CDN, web application, and storage workloads, where FOO reduces approximation error by 4000×. We anticipate that FOO and PFOO will prove important tools in the design of future caching systems.



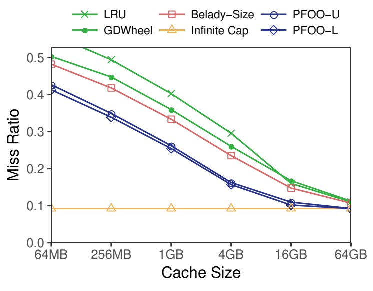
(a) CDN 1



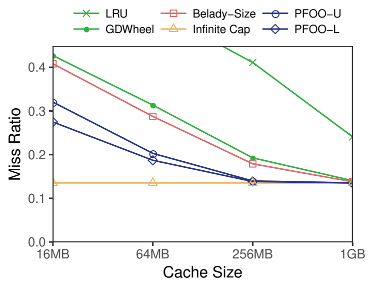
(b) CDN 2



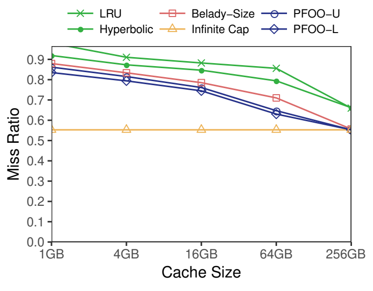
(c) CDN 3



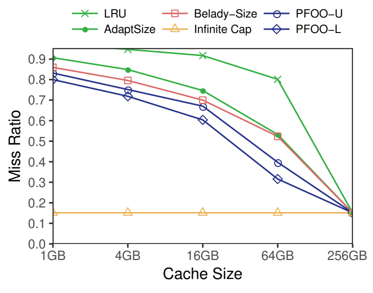
(d) WebApp 1



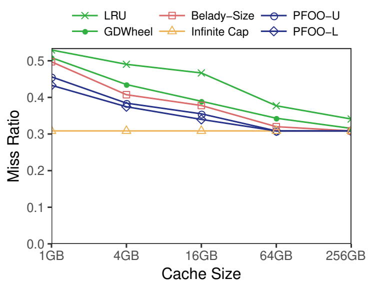
(e) WebApp 2



(f) Storage 1



(g) Storage 2



(h) Storage 3

Fig. 17. Miss ratio curves for PFOO vs. LRU, Infinite-Cap, the best prior offline upper bound, and the best online policy for our eight production traces.

10 ACKNOWLEDGMENTS

We thank Jens Schmitt, our shepherd Niklas Carlsson, and our anonymous reviewers for their valuable feedback.

Daniel S. Berger was supported by the Mark Stehlik Postdoctoral Teaching Fellowship of the School of Computer Science at Carnegie Mellon University. Nathan Beckmann was supported by a Google Faculty Research Award. Mor Harchol-Balter was supported by NSF-CMMI-1538204 and NSF-XPS-1629444.

REFERENCES

[1] Marc Abrams, C. R. Standridge, Ghaleb Abdulla, S. Williams, and Edward A. Fox. 1995. *Caching Proxies: Limitations and Potentials.* Technical Report. Virginia Polytechnic Institute & State University Blacksburgh, VA.

[2] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. 1971. Principles of optimal page replacement. *J. ACM* 18, 1 (1971), 80–93.

[3] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. 1993. *Network flows: theory, algorithms, and applications.* Prentice hall.

[4] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. 1999. Page replacement for general caching problems. In *SODA*. 31–40.

[5] Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. New results on a generalized coupon collector problem using Markov chains. *Journal of Applied Probability* 52, 2 (2015), 405–418.

[6] Omri Bahat and Armand M Makowski. 2003. Optimal replacement policies for nonuniform cache objects with optional eviction. In *IEEE INFOCOM.* 427–437.

[7] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. 2001. A unified approach to approximating resource allocation and scheduling. *J. ACM* 48, 5 (2001), 1069–1090.

[8] Ruben Becker, Maximilian Fickert, and Andreas Karrenbauer. [n. d.]. *A Novel Dual Ascent Algorithm for Solving the Min-Cost Flow Problem.* Chapter 12, 151–159.

[9] Ruben Becker and Andreas Karrenbauer. 2013. A Combinatorial O(m 3/2)-time Algorithm for the Min-Cost Flow Problem. *arXiv preprint arXiv:1312.3905* (2013).

[10] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Hit Rate by Maximizing Hit Density. In *USENIX NSDI.* 1–14.

[11] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *IEEE HPCA*. 64–75.

[12] Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache behavior beyond LRU. In *IEEE HPCA*. 225–236.

[13] L. A. Belady. 1996. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5 (1996), 78–101.

[14] Daniel S. Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. 2014. Exact analysis of TTL cache networks. *Perform. Eval.* 79 (2014), 2 – 23. Special Issue: Performance 2014.

[15] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. 2015. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review* 43, 2 (2015), 57–59.

[16] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a CDN. In *USENIX NSDI.* 483–498.

[17] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*. 499–511.

[18] Béla Bollobás and Graham Brightwell. 1992. The height of a random partial order: concentration of measure. *The Annals of Applied Probability* (1992), 1009–1018.

[19] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM.* 126–134.

[20] PJ Burville and JFC Kingman. 1973. On a model for storage and search. *Journal of Applied Probability* (1973), 697–701.

[21] Hao Che, Ye Tung, and Zhijun Wang. 2002. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE JSAC* 20 (2002), 1305–1314.

[22] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Technical Report. Hewlett-Packard Laboratories.

[23] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. 2012. Caching is hard—even in the fault model. *Algorithmica* 63 (2012), 781–794.

[24] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *USENIX NSDI.* 379–392.

[25] Edward Grady Coffman and Peter J Denning. 1973. *Operating systems theory.* Prentice-Hall.
[26] Edward Grady Coffman and Predrag Jelenković. 1999. Performance of the move-to-front algorithm with Markov-modulated request sequences. *Operations Research Letters* 25 (1999), 109–118.
[27] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.
[28] Samuel I Daitch and Daniel A Spielman. 2008. Faster approximate lossy generalized flow via interior point algorithms. In *ACM STOC*. 451–460.
[29] Asit Dan and Don Towsley. 1990. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *ACM SIGMETRICS*. 143–152.
[30] Wayne W Daniel et al. 1978. *Applied nonparametric statistics.* Houghton Mifflin.
[31] Robert P Dobrow and James Allen Fill. 1995. The move-to-front rule for self-organizing lists with Markov dependent requests. In *Discrete Probability and Algorithms*. Springer, 57–80.
[32] Aristides V Doumas and Vassilis G Papanicolaou. 2012. The coupon collector's problem revisited: asymptotics of the variance. *Advances in Applied Probability* 44, 1 (2012), 166–195.
[33] Ding-Zhu Du and Panos M Pardalos. 2013. *Handbook of combinatorial optimization* (2 ed.). Springer.
[34] Gil Einziger and Roy Friedman. 2014. Tinylfu: A highly efficient cache admission policy. In *IEEE Euromicro PDP*. 146–153.
[35] William Feller. 2008. *An introduction to probability theory and its applications*. Vol. 2. John Wiley & Sons.
[36] Andrés Ferragut, Ismael Rodríguez, and Fernando Paganini. 2016. Optimizing TTL caches under heavy-tailed demands. In *ACM SIGMETRICS*. 101–112.
[37] James Allen Fill and Lars Holst. 1996. On the distribution of search cost for the move-to-front rule. *Random Structures & Algorithms* 8 (1996), 179–186.
[38] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. 1992. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics* 39 (1992), 207–229.
[39] Lukáš Folwarczný and Jiří Sgall. 2017. General caching is hard: Even with small pages. *Algorithmica* 79, 2 (2017), 319–339.
[40] Christine Fricker, Philippe Robert, and James Roberts. 2012. A versatile and accurate approximation for LRU cache performance. In *ITC*. 8–16.
[41] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. 2012. Performance evaluation of the random replacement policy for networks of caches. In *ACM SIGMETRICS/ PERFORMANCE*. 395–396.
[42] Nicolas Gast and Benny Van Houdt. 2015. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS*. 123–136.
[43] Erol Gelenbe. 1973. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Trans. Comput.* 100 (1973), 611–618.
[44] Binny S Gill. 2008. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *USENIX FAST*. 4–18.
[45] WJ Hendricks. 1972. The stationary distribution of an interesting Markov chain. *Journal of Applied Probability* (1972), 231–233.
[46] Peter Hillmann, Tobias Uhlig, Gabi Dreo Rodosek, and Oliver Rose. 2016. Simulation and optimization of content delivery networks considering user profiles and preferences of internet service providers. In *IEEE Winter Simulation Conference*. 3143–3154.
[47] Wassily Hoeffding. 1948. A non-parametric test of independence. *The annals of mathematical statistics* (1948), 546–557.
[48] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *ACM SOSP*. 167–181.
[49] Stratis Ioannidis and Edmund Yeh. 2016. Adaptive caching networks with optimality guarantees. In *ACM SIGMETRICS*. 113–124.
[50] Sandy Irani. 1997. Page replacement with multi-size pages and applications to web caching. In *ACM STOC*. 701–710.
[51] Akanksha Jain and Calvin Lin. 2016. Back to the future: leveraging Belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*. 78–89.
[52] Predrag R Jelenković. 1999. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability* 9 (1999), 430–464.
[53] Predrag R Jelenković and Ana Radovanović. 2004. Least-recently-used caching with dependent requests. *Theoretical computer science* 326 (2004), 293–327.
[54] Predrag R Jelenkovic and Ana Radovanovic. 2004. Optimizing LRU caching for variable document sizes. *Combinatorics, Probability and Computing* 13 (2004), 627–643.
[55] Mahesh Kallahalla and Peter J Varman. 2002. PC-OPT: optimal offline prefetching and caching for parallel I/O systems. *IEEE Trans. Comput.* 51, 11 (2002), 1333–1344.

[56] Brian Karrer and Mark EJ Newman. 2009. Random graph models for directed acyclic networks. *Physical Review E* 80, 4 (2009), 046110.

[57] Richard E. Kessler, Mark D Hill, and David A Wood. 1994. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.* 43, 6 (1994), 664–675.

[58] W. Frank King. 1971. Analysis of Demand Paging Algorithms. In *IFIP Congress (1).* 485–490.

[59] Christos Koufogiannakis and Neal E Young. 2014. A nearly linear-time PTAS for explicit fractional packing and covering linear programs. *Algorithmica* 70, 4 (2014), 648–674.

[60] Conglong Li and Alan L Cox. 2015. GD-Wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS.* 1–15.

[61] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. 2016. Popularity-driven content caching. In *IEEE INFOCOM.* 1–9.

[62] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR* 45 (2015), 52–66.

[63] Valentina Martina, Michele Garetto, and Emilio Leonardi. 2014. A unified approach to the performance analysis of caching systems.. In *IEEE INFOCOM.* 12–24.

[64] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies.. In *IBM Systems journal*, Vol. 9. 78–117.

[65] John McCabe. 1965. On serial files with relocatable records. *Operations Research* 13 (1965), 609–618.

[66] Pierre Michaud. 2016. Some mathematical facts about optimal cache replacement. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 50.

[67] John Moriarty and Peter Neal. 2008. The Generalized Coupon Collector Problem. *Journal of Applied Probability* 45 (2008), 621–29.

[68] Giovanni Neglia, Damiano Carra, and Pietro Michiardi. 2018. Cache policies for linear utility maximization. *IEEE/ACM Transactions on Networking* 26, 1 (2018), 302–313.

[69] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *USENIX NSDI.* 385–398.

[70] Egerváry Research Group on Combinatorial Optimization. 2015. COIN-OR::LEMON Library. (2015). Available at http://lemon.cs.elte.hu/trac/lemon, accessed 10/21/17.

[71] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD* 22, 2 (1993), 297–306.

[72] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1999. An optimality proof of the LRU-K page replacement algorithm. *JACM* 46 (1999), 92–112.

[73] Antonis Panagakis, Athanasios Vaios, and Ioannis Stavrakakis. 2008. Approximate analysis of LRU in the case of short term correlations. *Computer Networks* 52 (2008), 1142–1152.

[74] Konstantinos Psounis and Balaji Prabhakar. 2001. A randomized web-cache replacement scheme. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Vol. 3. IEEE, 1407–1415.

[75] Konstantinos Psounis, An Zhu, Balaji Prabhakar, and Rajeev Motwani. 2004. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks* 45 (2004), 379–398.

[76] Eliane R Rodrigues. 1995. The performance of the move-to-front scheme under some particular forms of Markov requests. *Journal of applied probability* 32, 4 (1995), 1089–1102.

[77] Samta Shukla and Alhussein A Abouzeid. 2017. Optimal Device-Aware Caching. *IEEE Transactions on Mobile Computing* 16, 7 (2017), 1994–2007.

[78] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. 2014. Overlay networks: An Akamai perspective. In *Advanced Content Delivery, Streaming, and Cloud Services.* John Wiley & Sons.

[79] Storage Networking Industry Association (SNIA). 2008. MSR Cambridge Traces. http://iotta.snia.org/traces/388. (2008).

[80] David Starobinski and David Tse. 2001. Probabilistic methods for web caching. *Perform. Eval.* 46 (2001), 125–137.

[81] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47.

[82] Naoki Tsukada, Ryo Hirade, and Naoto Miyoshi. 2012. Fluid limit analysis of FIFO and RR caching for independent reference model. *Perform. Eval.* 69 (Sept. 2012), 403–412.

[83] Vijay V Vazirani. 2013. *Approximation algorithms.* Springer Science & Business Media.

[84] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *USENIX.* 487–498).

[85] Weiyu Xu and A. Kevin Tang. 2011. A Generalized Coupon Collector Problem. *Journal of Applied Probability* 48, 4 (2011), 1081–1094.

[86] Neal E Young. 2000. Online paging against adversarially biased random inputs. *Journal of Algorithms* 37 (2000), 218–235.

A PROOFS

A.1 Proof of equivalence of interval and classic ILP representations of OPT

Classic representation of OPT. The literature uses an integer linear program (ILP) representation of OPT [4]. Figure 18 shows this classic ILP representation on the example trace from Section 3. The ILP uses decision variables $x_{i,k}$ to track at each time i whether object k is cached or not. The constraint on the cache capacity is naturally represented: the sum of the sizes for all cached objects must be less than the cache capacity for every time i. Additional constraints enforce that OPT is not allowed to prefetch objects (decision variables must not increase if the corresponding object is not requested) and that the cache starts empty.

Object	**a**	**b**	**c**	**b**	**d**	**a**	**c**	**d**	**a**	**b**	**a...**
Decision Variables	$x_{1,a}$	$x_{2,a}$	$x_{3,a}$	$x_{4,a}$	$x_{5,a}$	$x_{6,a}$	$x_{7,a}$	$x_{8,a}$	$x_{9,a}$	$x_{10,a}$	$x_{11,a}$
	$x_{1,b}$	$x_{2,b}$	$x_{3,b}$	$x_{4,b}$	$x_{5,b}$	$x_{6,b}$	$x_{7,b}$	$x_{8,b}$	$x_{9,b}$	$x_{10,b}$	$x_{11,b}$
	$x_{1,c}$	$x_{2,c}$	$x_{3,c}$	$x_{4,c}$	$x_{5,c}$	$x_{6,c}$	$x_{7,c}$	$x_{8,c}$	$x_{9,c}$	$x_{10,c}$	$x_{11,c}$
	$x_{1,d}$	$x_{2,d}$	$x_{3,d}$	$x_{4,d}$	$x_{5,d}$	$x_{6,d}$	$x_{7,d}$	$x_{8,d}$	$x_{9,d}$	$x_{10,d}$	$x_{11,d}$

Fig. 18. Classic ILP representation of OPT.

Lemma 2. *Under Assumption 1, our ILP in Definition 1 is equivalent to the classical ILP from [4].*

Proof sketch. Under Assumption 1, OPT changes the caching decision of object k only at times i when $\sigma_i = k$. To see why this is true, let us consider the two cases of changing a decision variable $x_{k,j}$ for $i < j < \ell_i$. If $x_{k,i} = 0$, then OPT cannot set $x_{k,j} = 1$ because this would violate Assumption 1. Similarly, if $x_{k,j} = 0$, then setting $x_{k,i} = 1$ does not yield any fewer misses, so we can safely assume that $x_{k,i} = 0$. Hence, decisions do not change within an interval in the classic ILP formulation.

To obtain the decision variables $x'_{p,i}$ of the classical ILP formulation of OPT from a given solution x_i for the interval ILP, set $x'_{\sigma_i,j} = x_i$ for all $i \le j < \ell_i$, and for all i. This leads to an equivalent solution because the capacity constraint is enforced at every time step. □

A.2 Proof of Lemma 1

This section proves Lemma 1 from Section 5.3, which bounds T_B, i.e., the first time after i that all intervals in B are completed.

Recall the definitions of the generalized CCP, where coupon probabilities follow a distribution p, and the definition of the classical CCP, where coupons have equal request probability.

We will make use of the following result, which immediately follows from [5, Theorem 4, p. 415].

Theorem 6. *For $b \ge 0$ coupons, any probability vector p, and the equal-probability vector $q = (1/b, \ldots, 1/b)$, it holds that $\mathbb{P}\left[T_{b,p} < l\right] \le \mathbb{P}\left[T_{b,q} < l\right]$ for any $l \ge 0$.*

We will use this result to prove a bound on T_B, which is defined as $T_B = \max_{j \in B} \ell_j - i$. The proof bounds T_B first using a generalized CCP and then a classical CCP.

Proof of Lemma 1. We first bound T_B via a generalized CCP with stopping time $T_{b,p}$ and $p = (p_1, \ldots, p_b)$ with

$$p_i = \mathbb{P}\left[\text{object } k \text{ is requested} \mid k \in B\right] \quad \text{under } \mathcal{P}^M \quad . \tag{26}$$

As T_B contains requests to other objects $j \notin B$, it always holds that $T_B \ge T_{b,p}$. Figure 19 shows such a case, where T_B is extended because of requests to uncached objects and large cached objects. $T_{b,p}$, on the other hand, does not include these other requests, and is thus always shorter or equal to T_B. This inequality bounds the probabilities for all $l \geq 0$.

$$\mathbb{P}\left[T_B < l\right] \leq \mathbb{P}\left[T_{b,p} < l\right] \tag{27}$$

We then apply Theorem 6.

$$\leq \mathbb{P}\left[T_{b,q} < l\right] \tag{28}$$

□

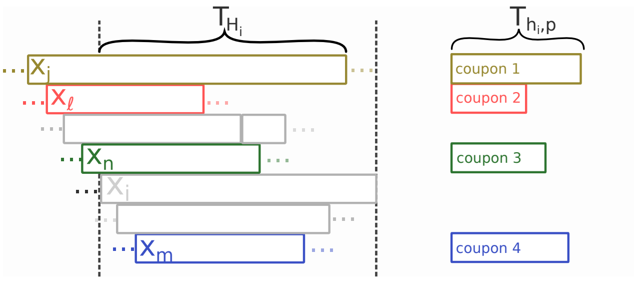


Fig. 19. Translation of the time until all B objects are requested once, T_B into a coupon-collector problem (CCP), $T_{b,p}$. As the CCP is based on fewer coupons (only objects $\in B$), the CCP serves as a lower bound on T_B.

A.3 Proof of Theorem 5

This section proves Theorem 5 from Section 5.4. Recall the definition of large and unpopular objects (Definition 5). At a high level, the proof shows that the remaining, "typical" objects are almost always part of a precedence relation. As a result, the probability of non-integer decision variables for typical objects vanishes as the number of objects M grows large.

Before we state the proof of Theorem 5, we introduce two auxiliary results, which are proven in Appendices A.4 and A.5).

The first auxiliary result shows that the number of cached objects h_i goes to infinity as the cache capacity C and the number of objects M go to infinity.

Lemma 3 (Proof in Section A.4). *For $i > N^*$ from Definition 5, $\mathbb{P}\left[h_i \to \infty\right] = 1$ as $M \to \infty$.*

The second auxiliary result derives an exponential bound on the lower tail of the distribution of the coupon collector stopping time as it gets further from its mean (roughly $b_i \log b_i$).

Lemma 4 (Proof in Section A.5). *The time $T_{b,q}$ to collect $b > 1$ coupons, which have equal probabilities $q = (1/b, \ldots, 1/b)$, is lower bounded by*

$$\mathbb{P}\left[T_{b,q} \leq b \log b - c\, b\right] < e^{-c} \quad \textit{for all } c > 0 \ . \tag{29}$$

With these results in place, we are ready to prove Theorem 5. This proof proceeds by using elementary probability theory and exploits our previous definitions of L_i, T_{B_i}, and $T_{b_i,q}$.

Proof of Theorem 5. We know from Theorem 4 that the probability of non-integer decision variables can be upper bounded using the random variables L_i and T_{B_i}.

$$\mathbb{P}\left[0 < x_i < 1\right] \leq \mathbb{P}\left[L_i > T_{B_i}\right] \tag{30}$$

We expand this expression by conditioning on $L_i = l$.

$$= \sum_{l=1}^{\infty} \mathbb{P}\left[T_{B_i} < l | L_i = l\right] \mathbb{P}\left[L_i = l\right] \tag{31}$$

We observe that $\mathbb{P}\left[T_{B_i} < l\right] = 0$ for $l \leq b_i$ because requesting b_i distinct objects takes at least b_i time steps.

$$= \sum_{l=b_i+1}^{\infty} \mathbb{P}\left[T_{B_i} < l | L_i = l\right] \mathbb{P}\left[L_i = l\right] \tag{32}$$

We use the fact that conditioned on $L_i = l$, events $\{L_i = l\}$ and $\{T_{B_i} < l\}$ are stochastically independent.

$$= \sum_{l=b_i+1}^{\infty} \mathbb{P}\left[T_{B_i} < l\right] \mathbb{P}\left[L_i = l\right] \tag{33}$$

We split this sum into two parts, $l \leq \Lambda$ and $l > \Lambda$, where $\Lambda = \frac{1}{2} b_i \log b_i$ is chosen such that Λ scales slower than the expectation of the underlying coupon collector problem with $b_i = \delta h_i$ coupons. (Recall that $\delta = |B_i|/|H_i|$ is the largest fraction of objects in H_i, defined in Definition 5.)

$$\leq \sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{B_i} < l\right] \mathbb{P}\left[L_i = l\right] \tag{34}$$

$$+ \sum_{l=\Lambda+1}^{\infty} \mathbb{P}\left[T_{B_i} < l\right] \mathbb{P}\left[L_i = l\right] \tag{35}$$

$$\leq \sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{B_i} < l\right] + \sum_{l=\Lambda+1}^{\infty} \mathbb{P}\left[L_i = l\right] \tag{36}$$

We now bound the two terms in Eq. (36), separately. For the first term, we start by applying Lemma 1.

$$\sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{B_i} < l\right] \leq \sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{b_i,q} \leq l\right] \tag{37}$$

We rearrange the sum (replacing l by c).

$$= \sum_{c=\frac{1}{2}\log b_i}^{1+\log b_i} \mathbb{P}\left[T_{b_i,q} \leq b_i \log b_i - c\, b_i\right] \tag{38}$$

We apply Lemma 4.

$$< \sum_{c=\frac{1}{2}\log b_i}^{1+\log b_i} e^{-c} \tag{39}$$

We solve the finite exponential sum.

$$= \frac{e^2}{e^2 - e} \frac{1}{\sqrt{b_i}} \tag{40}$$

For the second term in Eq. (36), we use the fact that L_i's distribution is Geometric(ρ_{σ_i}) due to Assumption 2.

$$\sum_{l=\Lambda+1}^{\infty} \mathbb{P}\left[L_i = l\right] = \sum_{l=\Lambda+1}^{\infty} \left(1 - \rho_{\sigma_i}\right)^{l-1} \rho_{\sigma_i} \tag{41}$$

We solve the finite sum.

$$= \left(1 - \rho_{\sigma_i}\right)^{\Lambda} \tag{42}$$

We apply Definition 5, i.e., $\rho_{\sigma_i} \geq \frac{1}{b_i \log\log b_i}$.

$$\leq \left(1 - \frac{1}{b_i \log\log b_i}\right)^{\frac{1}{2} b_i \log b_i} \tag{43}$$

Finally, combining Eqs. (40) and (43) yields the following.

$$\mathbb{P}\left[L_i \geq T_{b_i,q}\right] < \frac{e^2}{e^2 - e} \frac{1}{\sqrt{b_i}} + \left(1 - \frac{1}{b_i \log\log b_i}\right)^{b_i \log b_i} \tag{44}$$

As $b_i \log\ b_i$ grows faster than $b_i \log\log b_i$, this proves the statement $\mathbb{P}\left[L_i \geq T_{b_i,q}\right] \to 0$ as $h_i \to \infty$ (implying that $b_i = \delta h_i \to \infty$) due to Lemma 3. □

A.4 Proof of Lemma 3

This section proves that, for any time $i > N^*$, the number of cached objects h_i grows infinitely large as the number of objects M goes to infinity. Recall that N^* denotes the time after which the cache needs to evict at least one object. Throughout the proof, let $\overline{E}$ denote the complementary event of an event E.

Intuition of the proof. The proof exploits the fact that at least $h^* = C/\max_k s_k$ distinct objects fit into the cache at any time, and that FOO finds an optimal solution (Theorem 1). Due to Assumption 3, $M \to \infty$ implies that $C \to \infty$, and thus $h^* \to \infty$. So, the case where FOO caches only a finite number of objects requires that $h_i < h^*$. Whenever $h_i < h^*$ occurs, there cannot exist intervals that FOO could put into the cache. If any intervals could be put into the cache, FOO would cache them, due to its optimality.

Our proof is by induction. We first show that the case of no intervals that could be put into the cache has zero probability, and then prove this successively for larger thresholds.

Proof of Lemma 3. We assume that $0 < h^* < M$ because $h^* \in \{0, M\}$ leads to trivial hit ratios $\in \{0, 1\}$.

For arbitrary $i > N^*$ and any x that is constant in M, we consider the event $X = \{h_i \leq x\}$. Furthermore, let $Z = \{h_i \leq z\}$ for any $0 \leq z \leq x$. We prove that $\mathbb{P}[X]$ vanishes as M grows by induction over z and corresponding $\mathbb{P}[Z]$.

Figure 20 sketches the number of objects over a time interval including i. Note that, for any $z \leq x$, we can take a large enough M such that $z < h^*$, because $h^* \to \infty$. So the figure shows $h^* > z$. The figure also defines the time interval $[u, v]$, where u is the last time before i when FOO cached

h^* objects, and v is the next time after i when FOO caches h^* objects. So, for times $j \in (u, v)$, it holds that $h_j < h^*$.

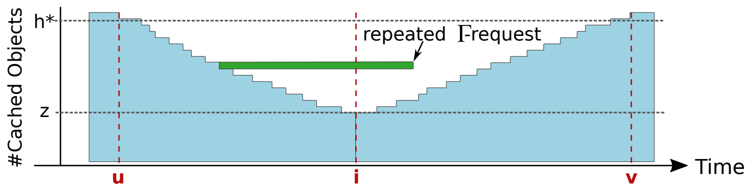


Fig. 20. Sketch of the event $Z = \{h_i \leq z\}$, which happens with vanishing probability if z is a constant with respect to M. The times u and v denote the beginning and the end of the current period where the number of cached objects is less than $h^* = C/\max_k s_k$, the fewest number of objects that fit in the cache. We define the set Γ of objects that are requested in $(u, i]$. If any object in Γ is requested in $[i, v)$, then FOO must cache this object (green interval). If such an interval exists, $h_i > z$ and thus Z cannot happen.

Induction base: $z = 0$ and event $Z = \{h_i \leq 0\}$. In other words, Z means the cache is empty. Let Γ denote the set of distinct objects requested in the interval $(u, i]$. Note that the event Z requires that, during $(u, i]$, FOO stopped caching all h^* objects. Because FOO only changes caching decisions at interval boundaries, Γ must at least contain h^* objects. Using the same argument, we observe that there happen at least h^* requests to distinct objects in $[i, v)$.

A request to any Γ object in $[i, v)$ makes Z impossible. Formally, let A denote the event that any object in Γ is requested again in $[i, v)$. We observe that $A \Rightarrow \overline{Z}$ because any Γ-object that is requested in $[i, v)$ must be cached by FOO due to FOO-L's optimality (Theorem 1). By inverting the implication we obtain $Z \Rightarrow \overline{A}$ and thus $\mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right]$.

We next upper bound $\mathbb{P}\left[\overline{A}\right]$. We start by observing that $\mathbb{P}[A]$ is minimized (and thus $\mathbb{P}\left[\overline{A}\right]$ is maximized) if all objects are requested with equal popularities. This follows because, if popularities are not equal, popular objects are more likely to be in Γ than unpopular objects due to the popular object's higher sampling probability (similar to the inspection paradox). When Γ contains more popular objects, it is more likely that we repeat a request in $[i, v)$, and thus $\mathbb{P}[A]$ increases ($\mathbb{P}\left[\overline{A}\right]$ decreases).

We upper bound $\mathbb{P}\left[\overline{A}\right]$ by assuming that objects are requested with equal probability $\rho_k = 1/M$ for $1 \leq k \leq M$. As the number of Γ-objects is at least h^*, the probability of requesting any Γ-object is at least h^*/M. Further, we know that $v - i \geq h^*$ and so

$$\mathbb{P}\left[\overline{A}\right] \leq \left(1 - \frac{h^*}{M}\right)^{h^*} .$$

We arrive at the following bound.

$$\mathbb{P}[\{h_i \leq z\}] = \mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right] \leq \left(1 - \frac{h^*}{M}\right)^{h^*} \longrightarrow 0 \quad \text{as } M \rightarrow \infty \tag{45}$$

Induction step: $z - 1 \rightarrow z$ for $z \leq x$. We assume that the probability of caching only $z - 1$ objects goes to zero as $M \rightarrow \infty$. We prove the same statement for z objects.

As for the induction base, let Γ denote the set of distinct objects requested in the interval $(u, i]$, excluding objects in H_i. We observe that $|\Gamma| \geq h^* - z$, following a similar argument.

We define $\mathbb{P}[A]$ as above and use the induction assumption. As the probability of less than $z-1$ is vanishingly small, it must be that $h_i \geq z$. Thus, a request to any Γ object in $[i, v)$ makes $h_i = z$ impossible. Consequently, $Z \Rightarrow \overline{A}$ and thus $\mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right]$.

To bound $\mathbb{P}[A]$, we focus on the requests in $[i, v)$ that do not go H_i-objects. There are at least $h^* - x$ such requests. $\mathbb{P}[A]$ is minimized if all objects, ignoring objects in H_i, are requested with equal popularities. We thus upper bound $\mathbb{P}\left[\overline{A}\right]$ by assuming the condition requests happen to objects with equal probability $\rho_k = 1/(M-z)$ for $1 \leq k \leq M - z$. As before, we conclude that the probability of requesting any Γ-object is at least $(h^* - z)/(M - z)$ and we use the fact that there are at least $h^* - x$ to them in $[i, v)$.

$$\mathbb{P}[\{h_i \leq z\}] = \mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right] \tag{46}$$

$$\leq \left(1 - \frac{h^* - z}{M - z}\right)^{h^* - z} \tag{47}$$

We then use that $z \leq x$ and that x is constant in M.

$$\leq \left(1 - \frac{h^* - x}{M}\right)^{h^* - x} \longrightarrow 0 \quad \text{as } M \to \infty \tag{48}$$

In summary, the number of objects cached by FOO at an arbitrary time i remains constant only with vanishingly small probability. Consequently, this number grows to infinity with probability one. □

A.5 Proof of Lemma 4

Proof of Lemma 4. We consider the time $T_{b,q}$ to collect $b > 1$ coupons, which have equal probabilities $q = (1/b, \dots, 1/b)$. To simplify notation, we set $T = T_{b,q}$ throughput this proof.

We first transform our term using the exponential function, which is strictly monotonic.

$$\mathbb{P}[T \leq b \log b - c\, b] = \mathbb{P}\left[e^{-sT} \leq e^{-s(b \log b - c\, b)}\right] \quad \text{for all } s > 0 \tag{49}$$

We next apply the Chernoff bound.

$$\mathbb{P}\left[e^{-sT} \leq e^{-s(b \log b - c\, b)}\right] \leq \mathbb{E}\left[e^{-sT}\right] e^{s(b \log b - c\, b)} \tag{50}$$

To derive $\mathbb{E}\left[e^{-sT}\right]$, we observe that $T = \sum_{i=1}^{b} T_i$, where T_i is the time between collecting the $(i-1)$-th unique coupon and the i-th unique coupon. As all T_i are independent, we obtain a product of Laplace-Stieltjes transforms.

$$\mathbb{E}\left[e^{-sT}\right] = \prod_{i=1}^{b} \mathbb{E}\left[e^{-sT_i}\right] \tag{51}$$

We derive the individual transforms.

$$\mathbb{E}\left[e^{-sT_i}\right] = \sum_{k=1}^{\infty} e^{-s\,k} p_i (1 - p_i)^{k-1} \tag{52}$$

$$= \frac{p_i}{e^s + p_i - 1} \tag{53}$$

We plug the coupon probabilities $p_i = 1 - \frac{i-1}{b} = \frac{b-i+1}{b}$ into Eq. (51), and simplify by reversing the product order.

$$\prod_{i=1}^{b} \mathbb{E}\left[e^{-sT_i}\right] = \prod_{i=1}^{b} \frac{(b-i+1)/b}{e^s + (b-i+1)/b - 1} = \prod_{j=1}^{b} \frac{j/b}{e^s + j/b - 1} \tag{54}$$

Finally, we choose $s = \frac{1}{b}$, which yields $e^s = e^{1/b} \geq 1 + 1/b$ and simplifies the product.

$$\prod_{j=1}^{b} \frac{j/b}{e^s + j/b - 1} \leq \prod_{j=1}^{b} \frac{j/b}{1/b + j/b} = \frac{1}{b+1} \tag{55}$$

This gives the statement of the lemma.

$$\mathbb{P}\left[T \leq b \log b - c\, b\right] \leq \frac{1}{b+1}\, e^{\frac{b \log b - c\, b}{b}} < e^{-c} \tag{56}$$

□

Received February 2018; revised April 2018; accepted June 2018