Keep-Alive Caching for the Hawkes process

Abstract

We study the design of caching policies in applications such as serverless computing where there is not a fixed size cache to be filled, but rather there is a cost associated with the time an item stays in the cache. We present a model for such caching policies which captures the trade-off between this cost and the cost of cache misses. We characterize optimal caching policies in general and apply this characterization by deriving a closed form for Hawkes processes. Since optimal policies for Hawkes processes depend on the history of arrivals, we also develop history-independent policies which achieve near-optimal average performance. We evaluate the performances of the optimal policy and approximate polices using simulations and a data trace of Azure Functions, Microsoft's FaaS (Function as a Service) platform for serverless computing.

1 INTRODUCTION

Datacenters provide a variety of caching services in the interest of reducing latency. While traditional caches have a fixed size determined by the underlying hardware, datacenter resources are fungible and could be used for a variety of purposes. For example, in serverless computing (also known as Function-as-a-Service) the cloud provider handles all the provisioning and configuration of resources for each user, while the user simply pays for the amount of time the function is executing. The cloud provider could, in principle, keep a container in memory for every user to ensure that function calls can be executed immediately. However, this would be excessively expensive in terms of resources allocated but unused. Instead, user application code which has not been used recently may be kept in persistent storage [Shahrad et al., 2020]. Another example involves Time-to-Live (TTL) caches where the cache controller has a timer-based parameter for when to evict objects from the cache. Work on TTL-based caches for Content Delivery Networks (CDNs) has observed that hit rate guarantees can be provided by controlling the cache space used by a customer and suggested pricing based on this space consumption [Basu et al., 2018].

In contrast to traditional caches, in keep-alive caching the decision is not about *what object to evict from the cache when space is needed. Rather the relevant question is* when is it worth keeping the object in the cache. Such a decision trades off the opportunity cost of not putting those resources to other uses (which may be caching other objects or some entirely different purpose) against the cost of a cache miss. While prior work has examined the trade-off between cache misses and the overall cache size [Basu et al., 2018], we focus on precise answers regarding optimal and approximately optimal caching decisions for a single object.

We model this problem as trading off between the expected *time* an object is kept in the cache before it is next accessed and the probability of a *cache miss*. Since a given object has a fixed size, this lets us precisely quantify the two relevant costs. We characterize the optimal cache policy and show how this characterization yields simple policies for objects whose arrivals have a monotone hazard rate. This includes Poisson and Hawkes processes [Laub et al., 2021].

One downside of the optimal policy for Hawkes processes is that it depends on the history of arrivals for the object. So, it needs to be recalculated after every arrival. A natural alternative is a simple TTL-style policy which keeps the object in the cache for a fixed amount of time after each arrival before evicting it. Such policies have been used in practice in serverless systems such as AWS Lambda and Azure Functions [Shahrad et al., 2020]. The classic ski rental problem shows that, if the TTL is optimized solely based on costs, the resulting policy is a worst-case 2-approximation. We derive an approach to optimize the TTL based on the parameters of the Hawkes process (but still independent of the history of arrivals).

Submitted to the 39th Conference on Uncertainty in Artificial Intelligence (UAI 2023). **To be used for reviewing only**.

We simulate the average performance of all three policies (optimal, fixed-TTL, and optimized-TTL) on arrival requests that follow a Hawkes process. The simulations illustrate how an optimized-TTL in general and our specific optimization procedure in particular yield near-optimal performance.

We also evaluate these policies on Azure traces released by Shahrad et al. [2020]. Here, Hawkes processes naturally capture applications such as web servers where a recent function invocation makes it more likely for additional invocations to occur (perhaps because the same user makes another request). We show that applying the optimal policy to the 25% of applications best fit by a Hawkes process yields meaningful improvements over the fixed-TTL policy at the scale of a datacenter. Furthermore, our optimized-TTL approach again yields near-optimal performance.

2 RELATED WORK

Our modeling decisions draw motivation from serverless computing. Cloud providers that use Function as a Service (FaaS) serverless models, such as AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and Azure Functions, handle all the system administration and resource allocations for customers. As Jonas et al. [2019] highlight, one of the advantages of serverless computing is that the users have to pay solely for the usage of resources of their applications. The customers do not pay for maintaining and starting up the resources when their application is not running. On the other hand, they point out the challenges for customers in terms of the unpredictable latency of cold starts. Wang et al. [2018] measure the cold start latency of popular FaaS serverless platforms. They discuss the historical usage of fixed caching policies by these platforms and quantify the behavior assumed by our model: cloud providers regularly shutting down instances providing FaaS services to reallocate those resources to other uses. Lin et al. [2020] discuss the desirability of approaches, such as ours, that could allow customers to agree to pay a higher price in exchange for personalized warm-start performance guarantees.

Closest to our work, Shahrad et al. [2020] propose a keep-alive cache policy based on an approach to predict future arrivals. They show a significant reduction in memory use when compared with the standard fixed keep-alive policies. In contrast, we take the predictions as given and optimize decisions based on them. Other work has considered other approaches to mitigating cold starts including having multiple tiers of hardware [Roy et al., 2022], pre-warming just the networking components [Mohan et al., 2019], caching common Python libraries [Oakes et al., 2018], and overbooking [Kesidis, 2019]. Fuerst and Sharma [2021] build a system which allows the size of a FaaS cache to be scaled dynamically based on arrival rates, but they do not provide any theoretical analysis. Other work has looked at the caching of other aspects of function execution such as data access [Romero et al., 2021, Mvondo et al., 2021].

Most work on cache algorithms works to optimize their decisions about which items to evict when the cache is full. Closer to our work is work on TTL caches which evict objects after a fixed amount of time. These are studied both for their applications in settings like Content Delivery Networks (CDNs) as well as their use as a more tractable way of approximately analyzing the performance of traditional caches introduced by Che et al. [2002]. Berger et al. [2014] analyze TTL cache networks. They consider the inter-request arrival times of objects and TTL values to be two independent renewal processes. They study three types of TTL cache policies based on the TTL resets and eviction times (we show what they term $\mathcal{R}$ policies are optimal in our setting). Basu et al. [2018] design two TTL based caching algorithms for CDNs like Akamai, but focus on working within a fixed target cache size. Ferragut et al. [2016] study optimal TTL cache policies. Like us they examine the consequences of the monotonicity of the Hazard rate of the inter-arrival distributions. However, unlike us they focus on trading off hit rate and overall cache size. They formulate the TTL caching problem as a non-linear optimization problem with non-linear constraint, and prove that the convexity of the optimization problem is related to monotonicity of the Hazard rate. Ali et al. [2011], Balamash and Krunz [2004] give a broader introduction to and survey of web page caches.

Closer to our work, Dehghan et al. [2019] share our notion that users may have utilities which depend on the hit rate and they include the possibility of increasing the cache size at a cost. They assume the utility functions for each file to be concave and the arrival requests to follow a Poisson point process, while out model is more general. Babaie et al. [2019] extend this to a cache hierarchy network. Panigrahy et al. [2017] allow heterogeneity of user preferences for hit rates, but focus on network of capacity-constrained caches with simple Poisson arrivals.

There is substantial work in the AI literature on other optimization problems that arise in the context of cloud computing such as pricing Blocq et al. [2014], Friedman et al. [2015], Babaioff et al. [2017], Kash et al. [2019], Dierks and Seuken [2021], reservation scheduling Azar et al. [2015], Wang et al. [2015], information elicitation Ceppi and Kash [2015], Dierks et al. [2019], and fair division of resources Parkes et al. [2015], Kash et al. [2014], Friedman et al. [2014], Narayana and Kash [2021].

3 MODEL

We consider a cache system where there is a cost for an object to stay in the cache. There is also a cost for a cache miss. For concreteness we describe our model using terms from one natural application (serverless computing), but it is also relevant for other applications such as CDNs Fer-

ragut et al. [2016]. First, we describe a cache policy in this setting. Next, we detail the parameters of the cache associated with the cloud provider. Then, we express the cost of a cache policy. Since the time of arrival of future requests is unknown, we assume that the cloud provider has access to the distribution of the arrival requests, as they could be estimated by the past arrival requests.

Cache Policy We assume the cache has infinite capacity because the provider can always dedicate more resources to its serverless offering, which differentiates our model from those driven by capacity. Thus, for us a cache policy is not about which object should be evicted when space is needed but rather how long (or more generally when) we should keep it in the cache.

Let $\mathcal{H}_{m-1} = \{t_1,\ t_2,\ \cdots,\ t_{m-1}\}$ denote the history of $m-1$ previous requests for the application. Here, $t_1,\ t_2, \cdots$ and, t_{m-1} denote the time of 1st, 2nd , $\cdots$ and, (m-1)-st request respectively. Let $x_m = t_m - t_{m-1}$ denotge the *m*-th inter-arrival time. Our analysis of policies is based on these inter-arrival times.

A cache policy π of an application is a sequence of time windows during which a possible requested application is moved in and out of the cache. A keep-alive window is a time interval during which the application is kept in the cache. The policy is reset after each arrival request for an application. More formally, a policy $\pi(x|\mathcal{H}_{m-1})$ can be expressed as an indicator function of a sequence of keep-alive windows for the m-th inter-arrival as:

$$\pi(x|\mathcal{H}_{m-1}) = \begin{cases} 1\ , & x \in [L_0,\ L_1] \bigcup \cdots\ [L_{2k-2}, L_{2k-1}] \\ 0\ , & \text{otherwise} \end{cases}$$

Here, L_{2i} denotes amount of time after t_{m-1} (the time of the most recent request) where the i-th keep-alive window starts, while L_{2i+1} denotes the point where the i-th keep-alive window end for some k and all $\ i \in \{0,\ 1,\ 2, \cdots, k-1\}$.

A policy $\pi(\cdot)$ consisting of a single keep-alive window across the m-th inter-arrival can be represented by the length of pre-warming window and the length of keep-alive window. That is, if

$$\pi(x) = \begin{cases} 1\ , & \text{for} \quad x\ \in [\tau_{\text{pw}},\ \tau_{\text{pw}} + \tau_{\text{ka}}] \\ 0\ , & \text{otherwise} \end{cases}$$

then, the policy can be summarized by the parameters τ_{pw}, and τ_{ka}. The parameter τ_{pw} refers to the length of pre-warming window which is the time interval during which the policy waits before bringing in the possibly requested application into the cache. The parameter τ_{ka} denotes the length of keep-alive window which is the time interval when the application is kept in the cache. The pre-warming window is especially useful when the requests for an application occur at regular intervals, such as requests governed by a timer function.

Cost of a Cache Policy In the context of serverless computing, an application encounters a *warm* start (cache hit) if its invocation (arrival request) occurs when the keep-alive window is active. An application has a *cold* start (cache miss) when the keep-alive window is not active during its invocation. Thus, we can describe the costs of a policy using:

- c_{cs} denotes the cost associated with a cold start. This is primarily the latency cost experienced by the user when there is a cold start. It may also include the cost for the cloud provider to load the application image (requested object) into the cache.
- c_p denotes the cost per unit time for the cloud provider for keeping the application image (requested object) active in the cache.

The cost of a policy with a single keep-alive window $(\tau_{\text{pw}}, \tau_{\text{ka}})$ when the inter-arrival time is x_m is given by $cost(x_m, (\tau_{\text{pw},\mathcal{H}_{m-1}}, \tau_{\text{ka},\mathcal{H}_{m-1}})) =$

$$\begin{aligned} &c_{cs},\ \text{if } x_m < \tau_{\text{pw},\mathcal{H}_{m-1}} \\ &c_p x_m,\ \text{if } \tau_{\text{pw},\mathcal{H}_{m-1}}\ \leq x_m\ \leq\ \tau_{\text{pw},\mathcal{H}_{m-1}} + \tau_{\text{ka},\mathcal{H}_{m-1}} \\ &c_p \tau_{\tau_{\text{ka},\mathcal{H}_{m-1}}} + c_{cs},\ \text{if } x_m >\ \tau_{\text{pw},\mathcal{H}_{m-1}} + \tau_{\text{ka},\mathcal{H}_{m-1}} \end{aligned}$$

The three cases for the cost of a policy correspond to that of a cold start if the invocation occurs during pre-warming, a warm start if the invocation occurs when the keep-alive window is active, and a cold start if the invocation is after the keep-alive window being active, respectively. In the more general case of multiple keep-alive windows the cost c_p must be paid for all prior windows as well (see Equation 1 in the proof of Lemma 1 for the full formula). This formula implicitly assumes that the time and costs associated with a decision to load the application into the cache (e.g. in the case of multiple windows where it may be moved in and out repeatedly) are zero, and so is in that sense a lower bound on the "true" cost. However, we show that for Hawkes processes in particular a single window which starts immediately ($\tau_{\text{pw}} = 0$) is optimal. Such a policy never has to pay such cost (except of course during cold starts, but those are already accounted for by c_{cs}).

To compute the cost of a cache policy, x_m must be known. However, the cloud provider does not have access to such information. Instead, they can estimate the distribution of x_m from past arrival requests $\mathcal{H}_{m-1}$. We model the application invocations as a point process. Let $f(x|\mathcal{H}_{m-1})$ and $F(x|\mathcal{H}_{m-1})$ denote the conditional probability distribution and the conditional cumulative distribution of an invocation at x units after the most recent invocation given the history $\mathcal{H}_{m-1}$ of previous invocations, respectively. We assume that both are continuous. Let the hazard rate of the based on the inter-arrival be expressed as $\lambda(x|\mathcal{H}_{m-1}) = \dfrac{f(x|\mathcal{H}_{m-1})}{1 - F(x|\mathcal{H}_{m-1})}$. We use these probability distributions to derive an expression for the expected cost of a cache policy.

4 CHARACTERIZATION OF OPTIMAL POLICIES

We start this section by deriving the expected cost of a caching policy over an inter-arrival. Then in Theorem 2, we characterize the optimal policy for application invocations in terms of the behavior of the Hazard rate. We apply this to derive optjmal policies when the arrival of application invocations follows a Poisson process or a Hawkes process, which are specific cases of arrival requests that have a constant Hazard rate and a monotone decreasing Hazard rate respectively.

Proofs omitted from this and subsequent sections of the paper can be found in the Appendix.

Lemma 1. *The expected cost of a cache policy over an inter-arrival is* $\mathbb{E}[cost(\pi(\cdot|\mathcal{H}_{m-1}))] =$

$$c_{cs} + \int_0^\infty \pi(x|\mathcal{H}_{m-1}) \cdot g(x|\mathcal{H}_{m-1})\, dx,$$

where the instantaneous cost at x units after the most recent arrival at t_{m-1} *is*

$$g(x|\mathcal{H}_{m-1}) = c_p \cdot \big(1 - F(x|\mathcal{H}_{m-1})\big) - c_{cs} \cdot f(x|\mathcal{H}_{m-1}).$$

Using the characterization of the cost of a policy from Lemma 1, Theorem 2 observes that the sign of g (which determines the optimal policy) is entirely determined by the hazard rate and costs.

Theorem 2. *The points* L_i *of the sequence of keep-alive windows over an inter-arrival for the optimal policy* $\pi_{opt}(x|\mathcal{H}_{m-1})$ *are at 0,* ∞*, or solutions to the equation* $c_p - (c_{cs} \cdot \lambda(x|\mathcal{H}_{m-1})) = 0$ *where the sign changes.*

We now examine several special cases of Theorem 2 with particularly natural structure. Our first, Corollary 2.1, describes the optimal policy when the hazard rate of the arrival requests are (weakly) decreasing. This case includes Poisson and Hawkes processes and is the main case we evaluate in our simulations.

Corollary 2.1. *If the hazard rate is weakly decreasing, the optimal policy* $\pi_{opt}(x|\mathcal{H}_{m-1})$ *is a single keep-alive window starting at* $\tau_{pw} = 0$*, and is given by*

$$\pi_{opt}(x|\mathcal{H}_{m-1}) = \begin{cases} 1\,, & \forall x \in [0,\ \tau_{opt,\mathcal{H}_{m-1}}] \\ 0\,, & \textit{otherwise} \end{cases} \quad \textit{where}$$

1. $\tau_{opt,\mathcal{H}_{m-1}} = \infty$*, i.e., to have the keep-alive window always be active when* $\forall x,\ \dfrac{c_p}{c_{cs}} < \lambda(x|\mathcal{H}_{m-1})$,
2. $\tau_{opt,\mathcal{H}_{m-1}} = 0$*, i.e., not cache and always have a cold start when* $\dfrac{c_p}{c_{cs}} > \lambda(x = 0|\mathcal{H}_{m-1})$
3. *a keep-alive window of length* $\tau_{opt,\mathcal{H}_{m-1}}$ *given by the solution to the equation* $\dfrac{c_p}{c_{cs}} = \dfrac{f(x = \tau_{opt,\mathcal{H}_{m-1}}|\mathcal{H}_{m-1})}{1 - F(x = \tau_{opt,\mathcal{H}_{m-1}}|\mathcal{H}_{m-1})}$*, otherwise.*

Next, Corollary 2.2, states the optimal policy when the hazard rate is instead (weakly) increasing.

Corollary 2.2. *If the hazard rate is weakly increasing, the optimal policy* $\pi_{opt}(x|\mathcal{H}_{m-1})$ *is a single keep-alive window with* $\tau_{ka} = \infty$ *and a pre-warming window, and is given by*

$$\pi_{opt}(x|\mathcal{H}_{m-1}) = \begin{cases} 1,\ \tau_{pw,\mathcal{H}_{m-1}} \le x \\ 0,\ \textit{otherwise} \end{cases} \quad \textit{where,}$$

1. $\tau_{pw,\mathcal{H}_{m-1}} = 0$*, i.e., to have the keep-alive window always be active when* $\forall x,\ \dfrac{c_p}{c_{cs}} < \lambda(x|\mathcal{H}_{m-1})$,
2. $\tau_{pw,\mathcal{H}_{m-1}} = \infty$*, i.e., to always have a cold start when* $\forall x,\ \dfrac{c_p}{c_{cs}} > \lambda(x|\mathcal{H}_{m-1})$.
3. $\tau_{pw,\mathcal{H}_{m-1}}$ *satisfies the equation* $\dfrac{c_p}{c_{cs}} = \dfrac{f(x = \tau_{pw,\mathcal{H}_{m-1}}|\mathcal{H}_{m-1})}{1 - F(x = \tau_{pw,\mathcal{H}_{m-1}}|\mathcal{H}_{m-1})}$*, i.e., an infinite keep-alive window after a pre-warming window of length* $\tau_{pw,\mathcal{H}_{m-1}}$ *when* $c_p - c_{cs}\lambda(x = 0|\mathcal{H}_{m-1}) > 0$ *and changes sign.*

More generally, we can combine these to understand the optimal policy when the hazard rate has a single peak: it is first increasing and then decreasing. In this case using both τ_{pw} and τ_{ka} is optimal (apart from degenerate cases). This is the form used by Shahrad et al. [2020], and so our results characterize the class of applications for which their approach could be optimal if properly tuned as well as determining how to optimally tune the parameters.[1] One notable example of this class of application is applications triggered by a timer. We would expect λ to be 0 until the timer is due to elapse, rapidly increase as we approach our estimate of when the timer will trigger, and then eventually decrease if we appear to be wrong in our estimate of when the timer will next trigger.

We can also apply Theorem 2 to applications where the hazard rate of the arrival requests for application invocation has a single valley. This might be the case if the initial invocation is likely to trigger several others (in the same spirit as a Hawkes process) but then once the application finishes there is a gap before it is invoked again, perhaps due to a timer. As Figure 1 shows the optimal keep-alive policy has at-most two keep-alive windows, whose lengths can be computed using Theorem 2. Of course we can apply this approach to more complex situations resulting in even more windows if the distributional information available supports that (e.g. for a timer that only causes an arrival under certain additional conditions).

[1] They use a simple rule of pre-warming at the 5th percentile and ending the keep-alive at the 99th percentile while our approach would optimize those based on the costs and distribution characteristics.

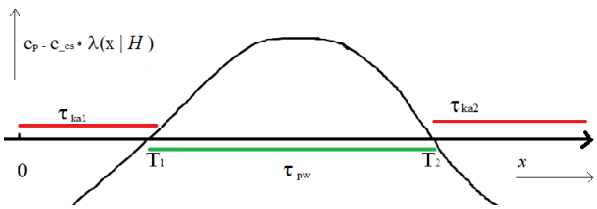


Figure 1: Optimal policy with a single valley hazard rate

4.1 POISSON PROCESS

Our first concrete application of Theorem 2 is when the distribution of the arrival requests follows a Poisson process. For the Poisson process, the relevant quantities are:

- $f(t) = \lambda \cdot e^{-\lambda \cdot t}$ for $t \geq 0$, where $\lambda > 0$ is a constant parameter.
- $F(t) = 1 - e^{-\lambda \cdot t}$.
- $\lambda(t) = \dfrac{f(t)}{1 - F(t)} = \lambda$.

From Corollary 2.1 we see that, barring degeneracy, the optimal policy for a Poisson process is one of two possibilities since the Hazard rate is constant. The optimal policy when $\dfrac{c_p}{c_{cs}} < \lambda$ is to have a keep-alive window active at all times, while when $\dfrac{c_p}{c_{cs}} > \lambda$ the optimal policy is to always experience a cold start. The cost of the optimal policy when $\dfrac{c_p}{c_{cs}} > \lambda$ is c_{cs}. Since the expected inter-arrival time is $1/\lambda$, the expected cost of the optimal policy when $c_p - (c_{cs} \cdot \lambda) \leq 0$ is $\dfrac{c_p}{\lambda}$.

4.2 HAWKES PROCESS

In this subsection, we explore the more interesting case of optimal policies when the distribution of the application invocations follow a Hawkes process. The Hawkes process is a point process whose hazard rate, or conditional intensity, is given by

$$\lambda(t|\mathcal{H}) = \frac{f(t|\mathcal{H})}{1 - F(t|\mathcal{H})} = \lambda_0 + \sum_{t_j \in \mathcal{H}} \mu(t - t_j)$$

where $\mathcal{H} = \{t_1, t_2, \cdots, t_{i-1}\}$ denotes the history of invocations of the function, λ_0 refers to the background intensity, and μ is the excitation function. We limit our analysis to the exponential excitation function for which $\mu(t) = \alpha e^{-\beta \cdot t}$. The constant $\alpha > 0$ captures the increase in the intensity from an arrival, while the constant $\beta > 0$ is the decay rate of the arrival's influence. The conditional intensity of the self-exciting Hawkes process with an exponential excitation function is thus

$$\lambda(t|\mathcal{H}) = \lambda_0 + \sum_{t_j \in \mathcal{H}} \alpha \cdot e^{-\beta \cdot (t - t_j)} \quad .$$

Corollary 2.1 characterizes the optimal policy when the distribution of arrival requests follow the Hawkes process to be one of the following policies,

- The keep-alive window is to always be active with $\tau_{\text{opt},\mathcal{H}_{m-1}} = \infty$ when, as in the Poisson case, the background intensity is sufficiently high: $\dfrac{c_p}{c_{cs}} < \lambda_0$.
- Experience a cold start with $\tau_{\text{opt},\mathcal{H}_{m-1}} = 0$ when $\dfrac{c_p}{c_{cs}} > \lambda(x|\mathcal{H}_{m-1}))$, after the most recent request.
- The keep-alive window is given by the expression

$$\tau_{\text{opt},\mathcal{H}_{m-1}} = \frac{1}{\beta}\Big(\log \alpha + \log\Big(\sum_{j=1}^{m-1} e^{\beta(t_j - t_{m-1})}\Big) - \log\Big(\frac{c_p}{c_{cs}} - \lambda_0\Big)\Big)$$

  otherwise. This expression is obtained by substituting the conditional intensity of the Hawkes process in Corollary 2.1 and solving for $\tau_{\text{opt},\mathcal{H}_{m-1}}$ as detailed in the Appendix.

4.3 OPTIMIZED-TTL KEEP-ALIVE WINDOWS FOR HAWKES PROCESSES

Corollary 2.1 provides the optimal history-dependent policy for Hawkes processes. We conclude this section by showing how it also provides motivation for a history-independent heuristic for Hawkes processes. In Section 5 we show this heuristic has strong empirical performance.

Our proposal is to empirically determine a window length that works well for "typical" windows. Intuitively, this can be done by sampling a number of histories, computing the optimal policy for each history, and computing a summary statistic. We find that the average of the optimal policies works well. We discuss the simulation of a Hawkes process from its parameters in Section 5

Corollary 2.3. *When the parameters of the Hawkes process are such that $c_p - (c_{cs} \cdot \lambda(x|\mathcal{H})) = 0$ has a solution, the optimal policy has a history independent lower bound, and an upper bound expressed as follows*

$$\tau_{opt,\mathcal{H}} \geq \frac{1}{\beta} \cdot \left(\log \alpha - \log\left(\frac{c_p}{c_{cs}} - \lambda_0\right)\right)$$

$$\tau_{opt,\mathcal{H}} \leq \frac{1}{\beta} \cdot \left(\log \alpha + \log \delta + 1 - \log\left(\frac{c_p}{c_{cs}} - \lambda_0\right)\right)$$

where δ satisfies

$$\sum_{i=m-\delta}^{m-1} e^{\beta \cdot (t_i - t_{m-1})} \geq \frac{1}{2} \sum_{i=1}^{m-1} e^{\beta \cdot (t_i - t_{m-1})}$$

That is, the most recent δ arrivals provide at least half the total weight of the history dependent term. This can be thought of as only having δ arrivals that are recent enough to matter. Apart from rare scenarios where δ is much larger or smaller than typical given the Hawkes process parameters, this bound is relatively insensitive to the exact history due to the log. In our simulations, rather than estimate δ we simply directly estimate a threshold via sampling.

We also considered other approaches to deriving history-independent policies. The solution to the classic ski rental problem shows that setting $\tau_{ka} = c_{cs}/c_p$ always has a cost withing a factor of 2 of that of the optimal history-dependent policy for Hawkes processes. This bound can be tightened by setting τ_{ka} in a way that depends on the parameters of the Hawkes process. However, we found that in practice this approach was overly conservative, so we defer analysis of it to the appendix.

5 SIMULATIONS

5.1 PERFORMANCES ON SIMULATED HAWKES PROCESSES

Our theoretical results derived the optimal policy and argued that optimized-TTL, which uses averaging, provided a good heuristic which is independent of history. To better understand its expected performance, we evaluate both policies on simulated application invocations governed by a Hawkes process. We use *Ogata's modified thinning algorithm* Ogata [1981] to generate the samples of the Hawkes process. We generate 600 sample points of function invocations in a single realization of the Hawkes process. We evaluate the policies by taking the mean over 100 realizations.

Figure 2 shows the average cost of the policies for the three possible cases of the optimal policy given in Corollary 2.1. It does so by varying c_cs with the value of c_p is normalized to 1. The red curve shows the performance of the fixed policy as the length of the fixed keep-alive window is varied, while the blue horizontal line indicates the performance of the optimal history dependent policy. The red dot indicates the cost of the fixed policy evaluated at the length of the average optimal keep-alive window (i.e optimized-TTL).

In Figure 2 (a), since c_p/c_{cs} is large compared to λ_0, the optimal keep-alive policy is to have a keep-alive window length of 0. In other words, the optimal policy is to encounter a cold start for every invocation. Here, the optimized-TTL policy is optimal since the optimized-TTL window is 0.

In Figure 2 (b), the red dot indicating the average window length used by the optimal policy is also near the point where the average cost of the fixed window keep-alive policy curve changes from a decreasing function of window length to an increasing function of window length. Intuitively, this point on the red curve corresponds to the point t_{opt} where the averaged expected cost function $\overline{g}(t) = c_p(1 - \overline{F}(t) - c_{cs}\overline{f}(t)$, where the averages are taken over the different histories and runs, changes from negative to positive. It is suggestive of some of the underlying regularities of Hawkes processes that the average of the optimal solutions and the optimal solution to the average problem are close to each other. Thus, optimized-TTL finds a near-optimal point in the space of fixed policies and the performance there is close to the true optimum.

In Figure 2 (c), we see that the optimal keep-alive policy is to always be active. Here, there is no indication of a red dot as it is out of bounds for the plot, where the red curve asymptotes toward the blue.

In the appendix we provide results for additional paramater settings showing the strong perfromance of optimized-TTL across a wide range of paramter settings.

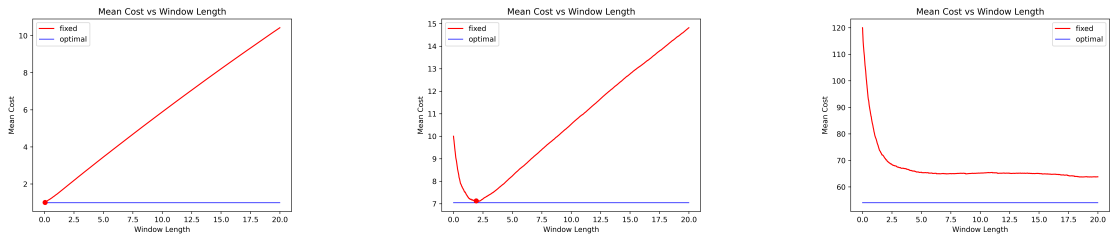
5.2 AZURE DATA TRACE EXPERIMENTAL SETUP

We evaluate the performance of the optimal policy and the optimized-TTL policy by comparing them with the fixed keep-alive policy on a subset of Azure traces released by Shahrad et al. [2020].[2]. The traces collect invocation counts of functions binned in 1-minute intervals. In Azure Functions, an application comprises of multiple functions where each function performs a specific task for the application. Since allocation of resources is based on applications (which are the unit of caching), we aggregate the bin counts of the function invocations belonging to the same application.

We evaluate performance using two metrics. The first is the amount of memory time that is wasted, normalized to the amount wasted by the default policy of a 10 minute keep-alive window. We accumulate the wasted memory time across all applications for a given policy. We assume the function execution times to be zero, to quantify the worst case wasted memory time. For this calculation we assume that all the applications use the same amount of memory. The second is the number of cold starts. We evaluate the cold start behavior by computing the average number of cold starts per application. We assume the first invocation of an application to be a cold start. These modeling decisions are generally consistent with those of Shahrad et al. [2020].

The fixed keep-alive policy was implemented by adding a fixed keep-alive window after an application invocation of, 5 minutes, 10 minutes, 20 minutes, 30 minutes, 45 minutes, 60 minutes, 90 minutes, or 120 minutes. The other two policies required fitting a Hawkes Process to the invocation pattern of an application. The Azure traces collect the data of application invocations for 14 days from July 15th to July 28th 2019. To avoid horizon effects or assuming

[2]These traces are available at `https://github.com/Azure/AzurePublicDatset`



(a) $c_p = 1, c_{cs} = 1$ (b) $c_p = 1, c_{cs} = 10$ (c) $c_p = 1, c_{cs} = 120$

Figure 2: Comparisons between optimal and fixed policies with Hawkes process parameters $\lambda_0 = 0.01, \alpha = 0.5, \beta = 1.0$

unreasonable amounts of prior data about an application We estimated the Hawkes process parameters based on the application invocations on day 8. We evaluate the policies for the application invocations during day 9.

We know that the data contains applications triggered by timers and other patterns are quite different from Hawkes processes. Therefore, we applied our policies only to those applications which were a good fit to a Hawkes process. The estimates of the Hawkes process parameters were computed as the minimum of the negative log-likelihood as described by Laub et al. [2021]. The log-likelihood for the estimation of Hawkes process parameters $\lambda_0,\ \alpha,$ and β is

$$\sum_{i=1}^{k} \log(\lambda_0 + \alpha A(i)) - \lambda t_k + \frac{\alpha}{\beta} \sum_{i=1}^{k} \left[e^{-\beta(t_k - t_i)} - 1 \right]$$

where $t_1,\ t_2, \cdots,\ t_k$ are the k invocations of the application, and $A(i) = \sum_{j=1}^{i-1} e^{-\beta(t_i - t_j)}$. The appropriateness of the application invocations being modeled by a Hawkes process is then determined by applying the Random time change theorem on the estimated parameters as detailed by Laub et al. [2021]. The similarity measure of the distribution of the application invocations to a Hawkes process was determined via the Kolmogorov–Smirnov (KS) test. After testing various thresholds, we applied the optimal keep-alive policy based on the estimated Hawkes process parameters to the 25% of processes which had the best goodness of fit. The default fixed policy was used for the remaining 75%.

From Section 4.2, we know that, apart from degenerate cases, the optimal policy is given by $\tau_{\text{opt},\mathcal{H}} = \frac{1}{\beta} \cdot \left(\log \alpha + \log \left(\sum_{j=1}^{m-1} e^{\beta(t_j - t_{m-1})} \right) - \log \left(\frac{c_p}{c_{cs}} - \lambda_0 \right) \right)$. In order to compute the optimal keep-alive window efficiently for applications with frequent arrivals we consider no more than 200 previous arrivals. We normalize $c_p = 1$ and compute the optimal policy for $c_{cs} = 5$, 10, 20, 30, 45, 60, 90, and 120. These values of c_{cs} are chosen in order to match the lengths used by the fixed policy.

After obtaining the applications that are a good fit for the Hawkes process with its estimated parameters, we apply the average optimal keep-alive window policy over simulated arrivals. We generate a single simulation of arrivals for each application based on the estimated Hawkes process parameters for 24 hours (1440 minutes). The average of the optimal keep-alive windows were computed for each application based on the simulated arrivals for the various values of c_{cs},

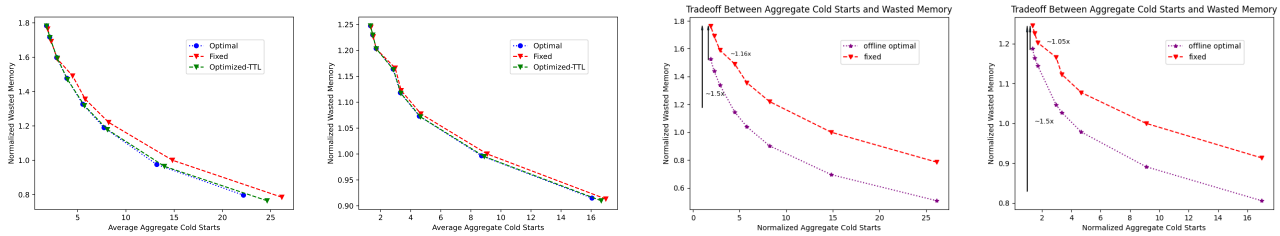
5.3 AZURE DATATRACE PERFORMANCE RESULTS

Overall, our results show that applying the optimal policy to applications fit by a Hawkes process yields benefits that are economically significant at the scale of a datacenter. Furthermore, the optimized-TTL policy yields near-equivalent results with no need to update based on history.

Figure 3 (a,b) plots the trade-off curve between the average number of cold starts per application and the normalized wasted memory for optimal, optimzed-TTL, and fixed policies. Figure 3 (a) includes only the treated (Hawkes-process-like) applications, while Figure 3 (b) includes all applications. We observe from Figure 3, that the optimal policy Pareto dominates the fixed keep-alive policy. We see that for high values of c_{cs} both metrics are almost the same for all policies. This is because, the optimal policy for many applications was selected to be the upper bound. Similarly, for low values of c_{cs} many applications use the lower bound. For intermediate values of c_{cs} we see that the optimal policy has almost the same number of normalized cold starts but lower amount of wasted memory. Furthermore, the performance of the Optimized-TTL policy is very similar to the optimal policy so we can get these benefits. Of course, since we only treat 25% of applications, these benefits are attenuated when considering all applications.

While the benefits are small in absolute terms we argue they are still economically significant. We quantify them by computing the area between the Pareto curve of the fixed and the other two policies. By dividing the area between the Pareto curves with the maximum number of average aggregate cold starts encountered by an application (effective x-axis length) or maximum normalized wasted memory (effective y-axis length). This gives a sense of the average improvement across the curve. We record the results in Table 1.

They show that the improvement of the optimized-TTL policy over the fixed policy is around 80% of that of the optimal policy indicating that the optimized-TTL policy performs



(a) Hawkes process applications (b) All applications (c) Hawkes process applications (d) All applications

Figure 3: Trade-off curve of average number of cold starts vs normalized wasted memory for various policies

Policy	Avg. Cold Start Savings (Hawkes)	(All)	Avg. Memory Savings (Hawkes)	(All)
Optimal	0.779	0.0959	0.0519	0.0070
Optimized-TTL	0.653	0.0834	0.0435	0.0061
Offline-Optimal	4.038	1.29	0.269	0.0948

Table 1: Average performance improvement over fixed policy

effectively close to the history dependent policy. Given that wasted memory in the existing system is normalized to 1, average improvements of approximately 0.05 on treated applications and 0.007 overall represent an overall decrease in resource use for the cache on the order of 5% on treated applications (0.7% overall).[3] While small in absolute terms, percentage improvements of this scale translate to tens or hundreds of millions of dollars in cost savings for a major cloud provider Dierks et al. [2019]. Alternatively, this same improvement could be used to improve customer experience, reducing the number of cold starts by an average of about 0.75 per day for treated customers.

To put these results in further context, recall that we are comparing against the fixed policy *tuned optimally per our theory*. In terms of average cost, simply using the default choice of 10 minutes would be substantially worse in many cases. Furthermore, our theory shows that this fixed policy is not a weak baseline but has strong theoretical properties in its own right. (See discussion at the end of Section 4.3.)

Finally, to give a bound on how much of the possible performance improvement our approach achieves, we also evaluate the performance of the offline optimal policy in Figure 3 for (c) only the applications that have a Hawkes process distribution (d) and all applications. Here, the offline optimal policy chooses to have a cold start if $c_p x_m \geq c_{cs}$, otherwise the keep-alive window is of length x_m. Here, x_m is the inter-arrival time for the m-th arrival of the application invocation. That is, this policy cheats in that it is tuned to the actual realized pattern of arrivals rather than any prediction. In this sense it provides a bound on how much we could hope to achieve. The results from Figure 3 (c), and also quantified in Table 1, show that the optimal Hawkes policy achieves a meaningful fraction of it. Figure 3 (d) includes a comparison to the improvements reported by Shahrad et al. [2020]. They report a 1.5x improvement in wasted memory at essentially no cost in cold starts, which is substantially larger than what even the offline optimal can achieve (1.05x). This highlights how much of their improvement comes from pre-warming apps, for example timers, which our theoretical results show is not necessary for Hawkes processes. They combine sophisticated predictive modeling with a simple rule for determining pre-warming and keep-alive decisions. In contrast, our results provide a sophisticated rule for these decisions, making them complementary.

[3]Because of the shape of the curves in Figure 3, the benefits may be modestly larger in practice because current operating point is toward the right end of the plots where the gap tends to be larger.

6 CONCLUSION

Motivated by applications such as serverless computing, we presented a model of caching policies which captures the trade-off between the cost of keeping objects in the cache and the cost of cache misses. We characterized optimal caching policies and examined the optimal policies in detail for Hawkes processes. Since optimal policies for Hawkes processes depend on the history of arrivals, we also developed history-independent policies based on the heuristic of averaging the optimal keep-alive window from simulated predictions of arrivals. Evaluation on Hawkes process simulations provided insights into the tuning and expected performance of these approximations. Evaluation on a data trace of Azure functions showed this approach can yield small,yet economically meaningful improvements at the scale of a datacenter. Our results point to several avenues for future work. Since our approach allows us to characterize optimal policies on a per-item basis, it is naturally suited for exploring customization based on individual customer utilities rather than an overall system average as done in our experiments and most prior work. Another direction would be to use our model to examine optimal policies for more complex scenarios, such as a hierarchy of caches.

References

Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, 2011.

Yossi Azar, Inna Kalp-Shaltiel, Brendan Lucier, Ishai Menache, Joseph Naor, and Jonathan Yaniv. Truthful online scheduling with commitments. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, pages 715–732, 2015.

Pariya Babaie, Eman Ramadan, and Zhi-Li Zhang. Cache network management using big cache abstraction. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 226–234. IEEE, 2019.

Moshe Babaioff, Yishay Mansour, Noam Nisan, Gali Noti, Carlo Curino, Nar Ganapathy, Ishai Menache, Omer Reingold, Moshe Tennenholtz, and Erez Timnat. Era: A framework for economic resource allocation for the cloud. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 635–642, 2017.

Abdullah Balamash and Marwan Krunz. An overview of web caching replacement algorithms. *IEEE Communications Surveys & Tutorials*, 6(2):44–56, 2004. doi: 10.1109/COMST.2004.5342239.

Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive ttl-based caching for content delivery. *IEEE/ACM transactions on networking*, 26(3):1063–1077, 2018.

Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of ttl cache networks. *Performance Evaluation*, 79:2–23, 2014.

Gideon Blocq, Yoram Bachrach, and Peter B Key. The shared assignment game and applications to pricing in cloud computing. In *AAMAS*, pages 605–612. Citeseer, 2014.

Sofia Ceppi and Ian Kash. Personalized payments for storage-as-a-service. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):83–86, 2015.

Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Sadoc Menasche, and Yong Chiang Tay. A utility optimization approach to network cache design. *IEEE/ACM Transactions on Networking*, 27(3):1013–1027, 2019.

Ludwig Dierks and Sven Seuken. The competitive effects of variance-based pricing. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 362–370, 2021.

Ludwig Dierks, Ian Kash, and Sven Seuken. On the cluster admission problem for cloud computing. In *Proceedings of the 14th Workshop on the Economics of Networks, Systems and Computation*, pages 1–6, 2019.

Andrés Ferragut, Ismael Rodríguez, and Fernando Paganini. Optimizing ttl caches under heavy-tailed demands. *ACM SIGMETRICS Performance Evaluation Review*, 44(1): 101–112, 2016.

Eric Friedman, Ali Ghodsi, and Christos-Alexandros Psomas. Strategyproof allocation of discrete jobs on multiple machines. In *Proceedings of the fifteenth ACM conference on Economics and computation*, pages 529–546, 2014.

Eric Friedman, Miklós Z Rácz, and Scott Shenker. Dynamic budget-constrained pricing in the cloud. In *Advances in Artificial Intelligence: 28th Canadian Conference on Artificial Intelligence, Canadian AI 2015, Halifax, Nova Scotia, Canada, June 2-5, 2015, Proceedings 28*, pages 114–121. Springer, 2015.

Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

Ian Kash, Ariel D Procaccia, and Nisarg Shah. No agent left behind: Dynamic fair division of multiple resources. *Journal of Artificial Intelligence Research*, 51:579–603, 2014.

Ian A Kash, Peter Key, and Warut Suksompong. Simple pricing schemes for the cloud. *ACM Transactions on Economics and Computation (TEAC)*, 7(2):1–27, 2019.

George Kesidis. Overbooking microservices in the cloud. *arXiv preprint arXiv:1901.09842*, 2019.

Patrick J Laub, Young Lee, and Thomas Taimre. The elements of hawkes processes, 2021.

Xiayue Charles Lin, Joseph E Gonzalez, and Joseph M Hellerstein. Serverless boom or bust? an analysis of economic incentives. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. Ofc: an opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 228–244, 2021.

Sushirdeep Narayana and Ian A Kash. Fair and efficient allocations with limited demands. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 5620–5627, 2021.

Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

Yosihiko Ogata. On lewis' simulation method for point processes. *IEEE transactions on information theory*, 27 (1):23–31, 1981.

Nitish K Panigrahy, Jian Li, Faheem Zafari, Don Towsley, and Paul Yu. What, when and where to cache: A unified optimization approach. *arXiv preprint arXiv:1711.03941*, 2017.

David C Parkes, Ariel D Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation (TEAC)*, 3(1):1–22, 2015.

Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa $ t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 122–137, 2021.

Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

Changjun Wang, Weidong Ma, Tao Qin, Xujin Chen, Xiaodong Hu, and Tie-Yan Liu. Selling reserved instances in cloud computing. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.