FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching

Alexander Fuerst
Indiana University Bloomington
USA
alfuerst@iu.edu

Prateek Sharma
Indiana University Bloomington
USA
prateeks@iu.edu

ABSTRACT

Functions as a Service (also called serverless computing) promises to revolutionize how applications use cloud resources. However, functions suffer from cold-start problems due to the overhead of initializing their code and data dependencies before they can start executing. Keeping functions alive and warm after they have finished execution can alleviate the cold-start overhead. Keep-alive policies must keep functions alive based on their resource and usage characteristics, which is challenging due to the diversity in FaaS workloads.

Our insight is that keep-alive is analogous to caching. Our caching-inspired Greedy-Dual keep-alive policy can be effective in reducing the cold-start overhead by more than 3× compared to current approaches. Caching concepts such as reuse distances and hit-ratio curves can also be used for auto-scaled server resource provisioning, which can reduce the resource requirement of FaaS providers by 30% for real-world dynamic workloads. We implement caching-based keep-alive and resource provisioning policies in our FaasCache system, which is based on OpenWhisk. We hope that our caching analogy opens the door to more principled and optimized keep-alive and resource provisioning techniques for future FaaS workloads and platforms.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Functions as a Service, Serverless Computing, Cloud Computing, Caching

ACM Reference Format:
Alexander Fuerst and Prateek Sharma. 2021. 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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446757

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446757

1 INTRODUCTION

Functions as a Service (FaaS) is an emerging and popular cloud computing model, where applications use cloud resources through user defined "functions" that execute application code [38, 39, 53]. By handling all aspects of function execution, including resource allocation, cloud platforms can provide a "serverless" computing model where users do not have to explicitly provision and manage cloud resources (i.e., virtualized servers). FaaS employs a fine-grained pricing model allowing applications to pay for the resources they use, and provides many other advantages such as near-infinite horizontal scaling. FaaS services are being offered by all major cloud platforms (such as Amazon Lambda [10], Google Functions [12], and Azure Functions [11]), and are being used by diverse applications such as web services, API services, parallel and scientific computing [23, 31, 37], and in machine learning pipelines [20, 21].

The execution time of each function is typically short—in the range of a few milliseconds to a few seconds. This tight latency requirement and the wide diversity in function characteristics raises new challenges for FaaS providers. Cloud platforms execute each function invocation in a virtualized execution environment such as a container or a virtual machine (VM). Using virtualization techniques, a single physical server can execute many functions concurrently and safely [55]. Each function invocation entails creating and launching a container or a VM, and fetching and installing the necessary libraries and dependencies, before the function itself can be executed. This "initialization" phase can take non-negligible time, and adds to the overall function execution latency observed by the user. Reducing this function "startup overhead" is a key challenge in serverless computing [2, 6, 34, 46].

To mitigate the startup overhead, a common technique is to keep the execution environment alive or "warm" for a small duration, so that future invocations of the same function can run in the already initialized environment. Keeping functions warm can reduce the cold-start overheads and overall function latency by more than 5× [43]. However, keeping a container or a VM alive consumes computing resources on the physical servers, and increases the resource requirements for hosting FaaS platforms. Thus, while keep-alive can reduce the effective function execution latency, it can also reduce the overall system utilization and efficiency.

In this paper, we focus on how diverse FaaS workloads can be efficiently executed, by developing a new class of resource management techniques that balance the fundamental latency vs. utilization tradeoff. We argue that keep-alive *policies* can have a crucial impact on application performance, and thus must be integrated into resource allocation and provisioning. Current cloud platforms and FaaS systems employ simple keep-alive policies. For example, AWS Lambda will keep functions "warm" for $\sim 15 - 60$ minutes,

and users have to resort to ad-hoc "polling" techniques to keep functions warm to avoid the cold-start penalty [1, 5, 17]. Similarly, OpenWhisk sets a constant time to live (10 minutes) for function containers.

However, optimized keep-alive policies must balance the overhead of keeping a function warm with the likelihood that it will be invoked in the near future. Ideally, the duration of keep-alive should depend on the function's characteristics. This is challenging because of the diverse range of functions with different initialization overheads, resource footprints (i.e., CPU and memory consumption), and request frequencies.

Our primary insight is that the resource management of functions is equivalent to object caching. Keeping a function warm is equivalent to caching an object, and a warm function execution is equivalent to a cache hit. Terminating a function's execution environment means that the next invocation will incur the cold-start penalty, and is thus equivalent to evicting an object from a cache. The objective is to keep functions warm such that the effective function latency is reduced, which is equivalent to caching's goal of reducing object access time. *By mapping keep-alive to the exhaustively studied field of caching, we can leverage its principles and techniques, and apply them to serverless computing.*

Specifically, we use and adapt the Greedy-Dual caching framework [26], and develop keep-alive policies based on it. Our policies are cognizant of the memory footprint, access frequency, initialization cost, and execution latency of different functions. This caching-based approach can improve both the function latency and server utilization compared to the simple keep-alive policies found in current FaaS platforms.

The caching analogy allows us to use the vast set of caching algorithms and analytical models, and provides a new approach to resource provisioning for FaaS platforms. We use hit-ratio curves to determine the ideal size of servers required for handling FaaS workloads, and develop a vertical auto-scaling approach that dynamically adapts server size based on the workload characteristics. The dynamic scaling uses proportional control and hit-ratio curves to minimize both the required server resources, and cold-start overheads.

The rise of serverless computing and the challenges posed by its heterogeneity, workload diversity, and latency requirements, will require a new class of approaches to FaaS resource management. We argue that the vast collection of algorithms, analytical models, practical optimizations, and hard lessons from one of the most well studied fields in computer science, caching, can be customized to address many of these challenges. While bespoke solutions to serverless resource management will continue to be developed, our intent is to show the equivalence of caching and FaaS, and to highlight how naturally and easily caching techniques can be adapted instead. This paper makes an initial exploration into the world of caching-based approaches for resource management in serverless computing, and makes the following contributions:

(1) We show the equivalence between caching and function keep-alive, and develop a family of caching-based keep-alive policies for reducing function cold-start overhead. We use a Greedy-Dual based approach that is designed to work even with the diverse FaaS workloads.
(2) We implement our caching-based techniques in a our system, FaasCache, which is based on OpenWhisk. We conduct extensive trace-driven and empirical analysis of the tradeoffs of keep-alive techniques under different workload characteristics based on the Azure FaaS traces [48] and popular FaaS applications [40].
(3) Our resource provisioning policies use hit-ratio curves to determine the ideal server configuration (such as memory size) required to handle different function workloads. Our proportional-control based dynamic vertical-scaling can adjust server resources to reduce the cold-start probability, and reduce the average server size by more than 30%.
(4) Our experimental results indicate that caching-based keep-alive can reduce cold-start overheads by 3×, improve application-latency by 6×, and reduce system load to serve 2× more requests.

2 BACKGROUND

2.1 Function Keep-Alive

Serverless computing is now being provided by all large public cloud providers, and is increasingly popular way to deploy applications on the cloud. Functions as a Service (FaaS) can also be realized on private clouds and dedicated clusters using frameworks such as OpenWhisk [9], OpenFaas [14], OpenLambda [34], etc. In this new cloud paradigm, users provide functions in languages such as Python, Javascript, Go, Java, and others. The functions are executed by the FaaS platform, greatly simplifying resource management for the application.

FaaS functions cannot assume that state will persist across invocations, and function definitions must first import and load all code and data dependencies on each execution. Each function is run inside a container such as Docker [4], or a lightweight VM such as Firecracker [15]. By encapsulating all of the function state and any side-effects, the virtual execution environment provides isolation among multiple functions, and also allows for concurrent invocations of the same function. Due to the overhead of starting a new virtual execution environment (i.e., container or VM), and initializing the function by importing libraries and other data dependencies, function execution thus incurs a significant "cold-start" penalty. Table 1 shows the breakdown of initialization time (last column) vs. the total running time of different FaaS applications, and we can that the initialization overhead can as much as 80% of the total running time. Thus, FaaS can result in significant performance (i.e., total function execution latency) overheads compared to conventional models of execution where applications can reuse state and do not face the high initialization and cold-start overheads.

Once a container for a function is created and the function finishes execution, the container can be kept alive instead of immediately terminating it. Subsequent invocations of the function can then *reuse* the already running container. This *keep-alive* mechanism can alleviate the cold-start overhead due to container launching (which can be ∼ 100 ms).

However, keep-alive is not a panacea for all FaaS latency problems. Keeping a container alive consumes valuable computing resources on the servers. Specifically, a running container occupies memory, and "warm" containers being kept alive in anticipation

Table 1: FaaS workloads are highly diverse in their resource requirements and running times. The initialization time can be significant and is the cause of the cold-start overheads, and depends on the size of code and data dependencies.

Application	Mem size	Run time	Init. time
ML Inference (CNN)	512 MB	6.5 s	4.5 s
Video Encoding	500 MB	56 s	3 s
Matrix Multiply	256 MB	2.5 s	2.2 s
Disk-bench (dd)	256 MB	2.2 s	1.8 s
Web-serving	64 MB	2.4 s	2 s
Floating Point	128 MB	2 s	1.7 s

of future function invocations can reduce the multiplexing and efficiency of the servers. Thus, we develop keep-alive *policies* that reduce the cold-start overhead while keeping the server utilization high.

Designing general keep-alive policies is challenging due to the extreme heterogeneity in the different function popularities, resource requirements, and cold-start overheads. For instance, a recent analysis of FaaS workloads from Azure [48] shows that function inter-arrival times and memory sizes can vary by more than three orders of magnitude. This workload heterogeneity magnifies the performance vs. utilization tradeoff faced by keep-alive policies, as we shall describe in the next section. Additionally, FaaS workloads also show a high temporal dynamism, which requires new approaches to resource provisioning and elastic scaling, which we also develop.

2.2 Caching

Our answer to solving the twin conundrum of keep-alive and provisioning that is robust to workload heterogeneity and dynamism, is to use concepts from a related, well-known field with the same challenges. Caching has a long history of robust eviction algorithms that use temporal locality such as LRU (Least Recently Used). The effectiveness of a caching algorithm depends on the workload's inter arrival time distribution, the relative popularities of different objects, and thus many variants of LRU such as LRU-k [47], segmented LRU [25], ARC [44], and frequency based eviction such as LFU [30], are widely used in caching systems. Because functions show a lot of diversity in their memory footprints, and since keep-alive is primarily constrained by server memory, we seek to use *size-aware* caching methods. While conventional caching algorithms and analytical models largely deal with constant-sized objects, many size-aware caching policies have been developed for web-pages and data [19]. In particular, we use the Greedy-Dual [58] online caching framework that deals with objects with different eviction costs, that are determined based on size and other factors. The Greedy-Dual family of eviction algorithms for non-identical objects can be extended in many ways. We use a common variant, Greedy-Dual-Size-Frequency [26–28], which considers the size and frequency of objects.

Caching has a rich collection of analytical and modeling techniques to determine the efficacy of caches for different workloads. Hit (or miss) ratio curves are widely used for cache sizing to achieve a target performance, and for understanding and modeling cache performance. Hit-ratio curves can be constructed both in an offline and online manner, using techniques involving reuse distances [60], eviction times [35], Che's approximation [24], footprint descriptors [51], and estimation techniques such as SHARDS [54], counterstacks [57], etc.

3 KEEP-ALIVE TRADEOFFS

In this section, we first present an empirical analysis of cold-start overheads of common serverless applications, followed by the tradeoffs in keep-alive policies.

System model. We assume that each function invocation runs in its own container. A FaaS platform may use a cluster of physical servers, and forward the function invocation requests to different servers based on some load-balancing policy. Our aim is to investigate general techniques that are independent of cluster-level load-balancing, and we therefore focus on *server-level* policies. Even on a single server, a function can have multiple independent and concurrent invocations, and hence containers. Each function has its own container disk-image and initialization code, and thus containers cannot be used by different functions. A function's containers are nearly identical in their initialization overheads and resource utilization, since they are typically running the same function code. When a function finishes execution, its container may be terminated, or be kept alive and "warm" for any future invocations of the same function. At any instant of time, each container is either running a function, or is being kept alive/warm. Thus, server resources are consumed by running containers, and containers being kept alive in anticipation for future invocations.

Keeping functions alive/warm presents a fundamental tradeoff: it can reduce application-latency and CPU and I/O overhead, but it increases memory pressure. Nevertheless, recycling the execution environment and keeping function containers alive is a useful performance optimization that is supported by large public cloud platforms [7, 8, 13]. In some scenarios, server resources may also be shared with long-running containers and VMs. In such cases, function keep-alive also influences the performance of other co-located applications and services, and the overall cloud efficiency. Therefore, understanding and optimizing this tradeoff is important, and we develop caching-based dynamic resource provisioning policies in Section 5. Our goal is to allow FaaS operators to understand the benefits of different levels of aggressive keep-alive policies.

Cold-start overheads in OpenWhisk. In order to understand the performance and latency implications of function cold-starts, we investigate the chain of events necessary to run function code in a popular FaaS platform, OpenWhisk [9]. A timeline of a function invocation request for a TensorFlow machine learning inference task is shown in Figure 1. The figure shows the major sources of cold-start overhead: from request arrival to the actual function execution. OpenWhisk first checks whether the function can be served from the pool of warmed containers it maintains, and if no container is found, a Docker container is launched, and the runtime for the function is initialized: which comprises of OpenWhisk and Python runtime initialization, as well as any specific *explicit* function initialization provided by the application. The total compulsory overhead, from the request arrival to the actual function execution, is significant: up to 2.5 seconds are spent loading all

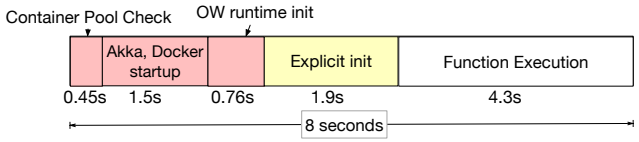


Figure 1: Timeline of function execution and sources of cold-start delay in OpenWhisk for an ML inference application.

```
1   #Initialization code
2   import numpy as np
3   import tensorflow as tf
4
5   m = download_model('http://model_serve/img_classify.pb')
6   session = create_tensorflow_graph(m)
7
8   def lambda_handler(event, context):
9       #This is called on every function invocation
10      picture = event['data']
11      prediction_output = run_inference_on_image(picture)
12      return prediction_output
```

Figure 2: Initializing functions by importing and downloading code and data dependencies can reduce function latency by hiding the cold-start overhead.

runtime dependencies, before the user-provided initialization and actual event handling code can begin execution.

Function Initialization. Function initialization refers to function-specific code for downloading and resolving code and data dependencies, which can be run before actual function execution (explicit-init component in Figure 1). For example, this can be used for downloading data dependencies ahead of time such as large neural network models for inference, or for runtime initialization such as downloading and importing package dependencies (e.g., Python packages). An example of function initialization is shown in Figure 2, which shows a pseudo-code snippet of a function that performs machine learning inference on its input. For ML inference, the function downloads an ML model and initializes the TensorFlow ML framework (lines 5 and 6). If the function's container is kept alive, then invocations of the function do not need to run the expensive initialization code (lines 2–6).

Workload Diversity and Dynamism. Designing keep-alive policies is not trivial due to the highly diverse and expanding range of applications that are using FaaS platforms. Conventionally, FaaS has been used for hosting web services, which is attractive because of the pay-per-use properties. Event handling functions for web responses typically have a small memory footprint but require low execution latency. Increasingly, FaaS is also being used for "heavy" workloads with high memory footprint and large initialization overheads such as highly parallel numerical computing (such as matrix operations [38], scientific computing [49], and machine learning [16]. The diversity of FaaS applications also results in a wide range of function memory footprints, running times, and initialization times, as seen in Table 1. Keep-alive policies must therefore balance the resource footprint of the containers with the benefits of keeping containers alive—and do so in manner that is applicable across a wide range of applications.

Furthermore, FaaS workloads show a high degree of dynamism and temporal effects. The Azure function [48] trace shows sharp diurnal effects: the function arrival rate is about 2× higher during the peak periods compared to the average. Function workloads are also heavy-tailed: a few "heavy hitting" functions are invoked much more frequently than others or consume a larger amount of computing resources, often by 2 or 3 orders of magnitude.

3.1 Policy Goals and Considerations

The primary goal of keep-alive is to reduce the initialization and cold-start latency, by keeping functions alive for different durations based on their characteristics. Because servers run hundreds of short lived functions concurrently, keep-alive policies must be generalizable and yield high server utilization. Functions can have vastly different characteristics, and keep-alive polices must work efficiently in highly dynamic and diverse settings. We use the following characteristics of functions for keep-alive policies.

The **initialization time** of functions can vary based on the code and data dependencies of the function. For example, a function for machine learning inference may be initialized by importing large ML libraries (such as TensorFlow, etc.), and fetching the ML model, which can be hundreds of megabytes in size and take several seconds to download. Functions also differ in terms of their **total running time**, which includes the initialization time and the actual execution time. Again, functions for deep-learning inference can take several seconds, whereas functions for HTTP servers and microservices are extremely short-lived (few milliseconds). The **resource footprint** comprises of the CPU, memory, and I/O use, and also differs widely based on the application's requirements. Finally, functions have different **frequencies** and invocation rates. Some functions may be invoked several times a second, whereas other functions may only be invoked rarely (if they are used to serve a very low-traffic web-site, for instance).

Because server resources are finite, it is important to prioritize functions which should be kept alive, based on the aforementioned characteristics. A function which is not popular and is unlikely to be called again in the near future, sees little benefits from keep-alive, and wastes server memory. Similarly, the resource consumption of the functions is also important: since keeping large-footprint functions alive is more expensive than smaller functions, smaller functions should be preferred and kept alive for longer. Finally, functions can also be prioritized based on their initialization overhead, since it is effectively wasted computation.

The problem of designing keep-alive policies is complicated by the fact that functions may have vastly different keep-alive priorities for the different characteristics. Consider a function with a large memory footprint (like those used in ML inference), high initialization overhead, and a low popularity. Such a function should have a low keep-alive priority due to its size, high priority due to large initialization overhead, and a low priority due to its low popularity. Thus, keep-alive policies must carefully balance all the different function characteristics and prioritize them in a coherent manner.

Current FaaS systems have shirked from this challenge and use primitive keep-alive policies that are not designed with the diversity and dynamism in mind. FaaS frameworks such as OpenWhisk, keep

all functions alive for a *constant* period of time (10 minutes). This is agnostic to different function characteristics such as resource footprint and initialization overheads, and only loosely captures popularity. More principled approaches are needed, which we provide next.

4 CACHING-BASED KEEP-ALIVE POLICIES

Formulating a keep-alive policy that balances priorities based on all competing characteristics of functions seems daunting.

> *The central insight of this paper is that keeping functions alive is equivalent to keeping objects in a cache.*

Keeping a function alive reduces its effective execution (or response) latency, in the same way as caching an object reduces its access latency. When all server resources are fully utilized, the problem of which functions *not* to keep alive is equivalent to which objects to *evict* from the cache. The high-level goal in caching is to improve the distribution of object access times, which is analogous to our goal of reducing the effective function latencies.

This caching analogy provides us a framework and tools for understanding the tradeoffs in keep-alive policies, and improving server utilization. Caching has been studied in wide range of contexts and many existing caching techniques can be applied and used for function keep-alive. Our insight is that we can use classic observations and results in object caching to formulate equivalent keep-alive policies that can provide us with well-proven and sophisticated starting point for understanding and improving function keep-alive.

In the rest of this section, we will show how cache eviction algorithms can be adapted to keep-alive policies. Caching systems typically seek to improve hit ratios (the fraction of accesses that are cache hits). However, focusing on hit-rates alone does not necessarily translate to improved *system* level performance, if the objects have different sizes and miss costs. For instance, caching all small objects may yield a high hit ratio, but the infrequent misses of larger objects results in higher miss costs and poor system throughput. Therefore, we will also focus on minimizing the overall cold-start overhead, which is equivalent to the "byte hit ratio" used in caching systems.

4.1 Greedy-Dual Keep-Alive Policy

While many caching techniques can be applied to the function keep-alive policies, we now present one such caching-inspired policy that is simple and yet captures all function characteristics and their tradeoffs. Our **GDSF** policy is based on Greedy-Dual-Size-Frequency object caching [26], which was designed for caches with objects of different sizes, such as web-proxies and caches. Classical caching policies such as LRU or LFU do not consider object sizes, and thus cannot be completely mapped to the keep-alive problem where the resource footprint of functions is an important characteristic. As we shall show, the Greedy-Dual approach provides a general framework to design and implement keep-alive policies that are cognizant of the frequency and recency of invocations of different functions, their initialization overheads, and sizes (resource footprints).

Fundamentally, our keep-alive policy is a function *termination* policy, just like caching focuses on eviction policies. Our policy is resource conserving: we keep the functions warm whenever possible, as long as there are available server resources. This is a departure from current constant time-to-live policies implemented in FaaS frameworks and public clouds, that are *not* resource conserving, and may terminate functions even if resources are available to keep them alive for longer.

Our policy decides which container to terminate if a new container is to be launched and there are insufficient resources availabile. The total number of containers (warm + running) is constrained by the total server physical resources (CPU and memory). We compute a "priority" for each container based on the cold-start overhead and resource footprint, and terminate the container with the lowest priority.

Priority Calculation. The GDSF keep-alive policy is based on Greedy-Dual caching [58], where objects may have different eviction costs. For each container, we assign a *keep-alive priority*, which is computed based on the frequency of function invocation, its running time, and its size:

$$\text{Priority} = \text{Clock} + \frac{\text{Freq} \times \text{Cost}}{\text{Size}} \tag{1}$$

On every function invocation, if a warm container for the function is available, it is used, and its frequency and priority are updated. Reusing a warm container is thus a "cache hit", since we do not incur the initialization overhead. When a new container is launched due to insufficient resources, some other containers are terminated based on their priority order—lower priority containers are terminated first. We now explain the intuition behind each parameter in the priority calculation:

Clock is used to capture the recency of execution. We maintain a "logical clock" per server that is updated on every eviction. Each time a container is used, the server clock is assigned to the container and the priority is updated. Thus, containers that are not recently used will have smaller clock values (and hence priorities), and will be terminated before more recently used containers.

Containers are terminated only if there are insufficient resources to launch a new container and if existing warm containers cannot be used. Specifically, if a container j is terminated (because it has the lowest priority), then Clock = Priority_j. All subsequent uses of other, non-terminated containers then use this clock value for their priority calculation. In some cases, *multiple* containers may need to be terminated to make room for new containers. If E is the set of these terminated containers, then Clock = $\max_{j \in E} \text{Priority(j)}$

We note that the priority computation is on a per-container basis, and containers of the same function share some of the attributes (such as size, frequency, and cost). However, the clock attribute is updated for each container individually. This allows us to evict the oldest and least recently used container for a given function, in order to break ties.

Frequency is the number of times a given function is invoked. A given function can be executed by multiple containers, and frequency denotes the *total* number of function invocations across all of its containers. The frequency is set to zero when all the containers of a function are terminated. The priority is proportional to the frequency, and thus more frequently executed functions are kept alive for longer.

Cost represents the termination-cost, which is equal to the total initialization time. This captures the benefit of keeping a container alive and the cost of a cold-start. The priority is thus proportional to the initialization overhead of the function.

Size is the resource footprint of the container. The priority is inversely proportional to the size, and thus larger containers are terminated before smaller ones. In most scenarios, the number of containers that can run is limited by the physical memory availability, since CPUs can be multiplexed easily, and memory swapping can result in severe performance degradation. Thus for ease of exposition and practicality, we consider only the container *memory* use as the size, instead of a multi-dimensional vector.

We can also use multi-dimensional resource vectors to represent the size, in which case we convert them to scalar representations by using the existing formulations from multi-dimensional *bin-packing.* For instance, if the container size is **d**, then the size can be represented by the magnitude of the vector $||\mathbf{d}||$. Other size representations can also be used. A common technique is to normalize the container size by the physical server's total resources (**a**), and then compute the size as $\sum_j \frac{d_j}{a_j}$ where d_j, a_j are the container size and total resources of a given type (either CPU, memory, I/O) respectively. Cosine similarity between **d** and **a** can also be used, as is widely used in multi-dimensional bin-packing.

FaaS-specific considerations. The application of cache eviction algorithms to FaaS keep-alive is fairly straight-forward. The various inputs Greedy-Dual (memory size, cold-start time, frequency) are available once a function has finished execution, and thus the keep-alive policy is completely online. Our policy calculates eviction priorities at the function level, but evicts at the container level. Recall that a particular function may have multiple containers associated with concurrent function invocations. We assume that all containers of a function are identical, i.e., they have the same initialization cost, footprint, etc. Thus, any one of the identical containers can be evicted.

4.2 Other Caching-Based Policies

The Greedy-Dual approach also permits many specialized and simpler policies. For instance, allowing for different parameters in Equation 1 results in different caching algorithms. If only the access clock is used as a priority, and other parameters are ignored, then we get **LRU**, with its ease of analysis and generality which has been well established with over half a century of empirical and analytical work. Using only frequency yields **LFU.** Similarly, a size aware keep-alive policy can be obtained by using 1/size as the priority, which would be useful in scenarios where memory size is at a premium.

Other size-aware online algorithms with tight online theoretical guarantees can also be applied. We also implement the **LANDLORD** [59] algorithm, which can be understood as a variant of the Greedy-Dual approach. Landlord also considers the frequency, size and initialization cost of functions. When the server is full and some container is to be evicted, a "rent" is charged from each function based on its size and initialization cost (specifically, it is equal to min $\frac{\text{initialization cost}}{\text{size}}$. This subtly differs from Greedy-Dual-Size-Frequency: the decrease in priority is computed based on the state of all the cached containers, and not independently applied. Upon a function invocation, its containers get a "credit", and their priority is set to their initialization cost. The containers with the lowest credits are evicted. Landlord has appealing and well-proven properties of its online performance: its competitive ratio (the performance compared to an optimal *offline* algorithm that knows future requests) has been well analyzed [59].

5 SERVER PROVISIONING POLICIES

Resource provisioning, i.e., determining the size and capacity of the servers for handling FaaS workloads, is a fundamental problem in serverless computing. In this section, we develop techniques that allocate the appropriate amount of resources to servers based on the characteristics of the function workloads. Resource provisioning policies must consider the rate of function invocations, the resource footprints of the functions, and the inter-arrival time between function invocations. To handle the interplay and tradeoffs between these factors, we use similar principles for provisioning that we used for developing our keep-alive policies. In case FaaS workloads are co-located with other applications such as long-running containers and VMs, our provisioning policies can also be used to determine the resource allocation of the combined running and warm function pool.

The fundamental challenge underlying resource provisioning for FaaS workloads is the performance vs. resource allocation tradeoff. Running a workload on large servers/VMs provides more resources for the keep-alive cache, which reduces the cold-starts and improves the application performance. However, we must also be careful to not *overprovision*, since it leads to wasted and underutilized resources. Additionally, since function workload can be dynamic, resource provisioning must be *elastic*, and be able to dynamically scale up or down based on the load. We therefore present a *static* provisioning policy that determines the server memory size for a given function workload, and then develop an elastic-scaling approach for handling workload temporal dynamics.

5.1 Static Provisioning

In Section 2, we have seen how keeping function containers warm in a keep-alive cache can help mitigate the cold-start overheads. The effectiveness of any keep-alive policy depends on the size of this keep-alive cache, and thus the server resources available, i.e., the server size. Our *static* provisioning policy thus selects a server size for handling a given workload. We want to optimize the resource provisioning to avoid over and under provisioning, both of which are detrimental to cost and performance respectively.

Having established that keep-alive policies are equivalent to cache eviction in the previous section, we now extend the use of the caching analogy further, to develop a caching-based provisioning approach. We claim that the performance vs. resource availability tradeoff of serverless functions can be understood and modeled using cache hit (or miss) ratio curves. Hit-ratio curves are widely used in cache provisioning and modeling, since they give insights into cache performance at different sizes. Once a hit-ratio curve is obtained, it is used to provision the cache size based on system requirements. A common approach is to size the cache based on a target hit-ratio (say, 90%). Alternatively, the slope of a hit-ratio curve can be understood to be the marginal utility of the cache,

and a cache size that maximizes this marginal utility is picked. This entails choosing a cache size which corresponds to the *inflection point* of the hit-ratio curve.

Hit-ratio Curve Construction. We use a function hit-ratio curve for determining the percentage of warm-starts at different server memory sizes. The hit-ratio curve is constructed by using the notion of *re-use distances.* A function's reuse-distance is defined as the total (memory) size of the unique functions invoked between successive invocations of the same function. For example, in the request *reuse* sequence of `ABCBCA`, the reuse distance of function `A` is equal to `size(B) + size(C)`. The *distribution* of these reuse distances can yield important insights into the required cache size. If the cache size is greater than the reuse distances, then there will be no cache misses. This can be generalized to find the hit-ratio at cache size c:

$$\text{Hit-ratio}(c) = \sum_{x=0}^{c} P(\text{Reuse-distance} = x), \tag{2}$$

where the reuse distance probability is obtained by scanning the entire input function workload for all reuse sequences. Conveniently, the hit-ratio is the CDF (cumulative distribution function) of the reuse distances, which can be empirically determined based on all the computed reuse distances. We show one such hit-ratio curve constructed with reuse distances, for a representative sample of the Azure function workload in Figure 3. We can see that the hit-ratio curve of functions *also* follows the classic long-tailed behavior: the hit-ratio steeply increases with cache size up to an inflection point, after which we see diminishing returns.

This technique and observation informs our provisioning policy. We construct a hit-ratio curve based on reuse distances, and size the server's memory based on the inflection point. Alternatively, we can set a target hit ratio (say, 90%), and use that to determine the minimum memory size of the server. Finding the reuse-distances for an entire trace can be an expensive, one-time operation, and takes $O(N * M)$ time where N is the number of invocations and M is the number of unique functions. However, sampling techniques such as SHARDS [54] can be applied to drastically reduce the overhead, making this a practical and principled technique for resource provisioning.

Limitations of the Caching Analogy. The error in hit-ratios with the reuse-distance approach in Figure 3 highlights an important facet where caching does not fully map to FaaS. The main difference is due to the limitations on the concurrent execution of functions: caching deals with unique objects, whereas there can be multiple containers for a function. At lower cache sizes, a high miss rate results in higher server load, and hence a higher number of dropped requests, that the classical reuse-distance approaches do not capture. If all warmed containers of a function are in use, then a new invocation results in a cold-start—which would be counted as a cache "hit". Thus at lower sizes, the real hit-ratio is lower than the ideal. At larger sizes, *multiple* containers corresponding to concurrent invocations of a function will be present, which results in a deviation from the hit-rate curve. Reconciling these differences is an interesting area of future work. However, we note that hit-ratio curves are only used for coarse-grained allocation, and small deviations result in slight under or over provisioning. Moreover, our dynamic allocation policy described next can reduce these errors using proportional control.

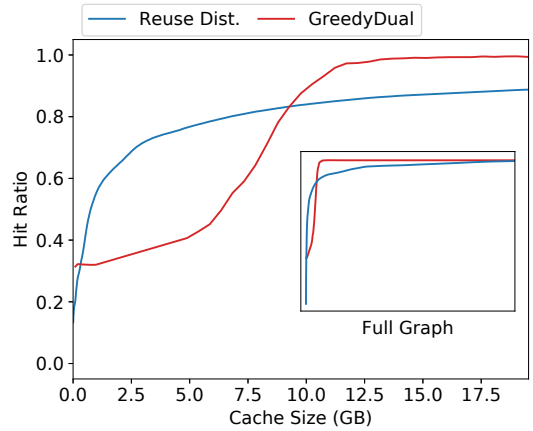


Figure 3: Hit ratio curve using reuse distances show slight deviations from the observed hit ratios due to dropped requests at lower sizes, and concurrent executions at higher sizes.

5.2 Elastic Dynamic Scaling

We also use the hit-ratio curve approach for a *dynamic* auto-scaling policy that adjusts the server size based on workload requirements. We assume that the FaaS server backend is running functions as containers either inside a virtual machine (VM), or is sharing the physical server with other cloud applications. In either case, it is important to be able to reclaim unused keep-alive cache resources and reduce its footprint, in order to increase the efficiency of the cloud platform.

Our vertical elastic scaling policy is simple and is intended to demonstrate the efficacy of a general caching based approach. We implement a proportional controller [3] which periodically adjusts the VM memory size based on the rate of cold-starts. Thus during periods of low rate of function invocations (i.e., arrival rate), the cache size can be reduced. This may *increase* the miss-ratio—but we care about the cold-starts (i.e., misses) per second, which is product of miss-ratio and invocations per second. Our controller monitors the arrival and cold-start rate, and uses the hit-ratio curve to decrease or increase VM size dynamically. We use VM resource deflation [50] to shrink or expand the VM by using a combination of hypervisor level page swapping, or guest-OS memory hot-plug and unplug.

Assume that we have a target miss speed (number of cold-starts/misses per second). For instance, this target value can be a product of the desired hit-ratio, h, and the average function arrival rate for the entire workload trace, $\bar{\lambda}$. Periodically, we monitor the exponentially smoothed arrival rate λ, and the observed miss speed. Our proportional controller adjusts the cache size in order to reduce the difference between the actual vs. target miss speed. This error is used to compute the new *miss rate*, m, and the associated cache size c' as follows:

$$\text{HR}(c') = 1 - m = 1 - h\frac{\bar{\lambda}}{\lambda} \tag{3}$$

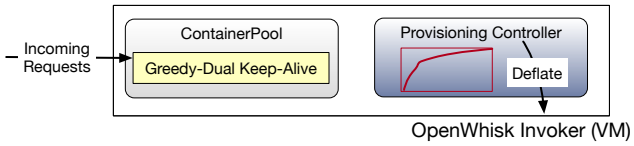


Figure 4: FaasCache system components. We build on OpenWhisk and augment it with new keep-alive policies and a provisioning controller.

The new cache size c' is then determined by inverting the hit-rate function HR. Our vertical scaling controller is designed for coarse-grained VM size adjustments, and only tracks the workload at time granularities of several minutes. Our intent with this policy is to not be overly aggressive with the capacity changes, but only to capture the coarse diurnal effects. Therefore, we use a large error deadband: the cache size is only updated if the error is more than 30%. Finally, the memory scaling can also be combined with cpu auto-scaling based on the function arrival rate, using classical predictive and reactive auto-scaling techniques found in web-clusters [32].

Online adjustments. Our policies rely on the *aggregate* function characteristics, which is used for constructing the hit-ratio curve. Once done, the traffic intensity (invocations per second) can change. We primarily assume that the probability distribution of function characteristics such as their frequency and size, does not significantly change. However, our dynamic scaling policy can adjust to changes in the traffic intensity (invocations per second). In other words, we assume that the future traffic is going to be similar to the past, which is the basis of the timeseries-forecasting based policies (such as in [48]), and is the fundamental principle underlying caching in general. Our provisioning policies are not completely online, since they have a preparation phase for constructing the hit-rate curves. A "drift" in function characteristics is fixed by periodically updating the hit-ratio curve, which we currently do once per week. Online hit-ratio curves can also be constructed, and adapting techniques such as [61] is part of our future work.

6 IMPLEMENTATION

We have implemented the keep-alive and the provisioning policies as part of our FaasCache framework built on top of OpenWhisk (Figure 4).

Keep-Alive. FaasCache replaces the default OpenWhisk TTL-based keep-alive policy with the Greedy-Dual-Size-Frequency approach. For each initialized container, we assign and maintain the keep-alive prioritized ContainerPool, which is only a 100-line Scala modification. Each invocation of a function (OpenWhisk action) in ContainerPool records the launch time and when results are returned.

If the container was prewarmed before the invocation arrived, we record it as the function's warm runtime. For new functions, the initialization overhead is caputred and assumed to be the worst-case runtime until a warmed invocation is recorded. In the subsequent invocations, the initialization overhead is computed by subtracting the cold from the warm time. The function's frequency and clock value are updated with each request. If the last container of a function is evicted, its cold and warm runtimes are stored and used to compute priority for its future invocations. To preserve the invocation fast-path, the ContainerPool is not kept sorted by priority. Instead, it is sorted by priorities only during evictions, when the lowest priority container(s) are terminated. We batch eviction operations to optimize the slow-path: we evict multiple containers to reach a certain free resource threshold (1000 MB is the current default).

In the future, we intend to implement a similar design that is found in the Linux kernel page eviction. A separate thread (analogous to kswapd) can be used to periodically sort the containerpool list and asynchronously evict containers, so that eviction is not on the critical path.

Provisioning. For the static provisioning, we compute the reuse distance distribution for a given workload trace, and assume stationarity — that it will be applicable on similar future workloads. We compute the reuse distances conventionally, by examining all reuse-sequences. The dynamic provisioning controller runs periodically (every 10 minutes), to deflate or inflate the VM size, if the cold start rate deviates from the target significantly (by more than 30%). When the VM has to be shrunk, we use cascade deflation [50]. We shrink the ContainerPool first, and reclaim the free memory using guest OS-level memory hot-unplug and hypervisor-level page swapping.

Keep-alive Simulator. We have implemented a trace-driven discrete event simulator for implementing and validating different keep-alive policies. Our simulator is written in Python in about 2,000 lines of code, and implements the various Greedy-Dual variants. It allows us to determine the cache hit ratios and the cold-start overheads for different workloads and memory sizes. Additionally, it also implements the static and dynamic provisioning policies for adjusting server size.

7 EXPERIMENTAL EVALUATION

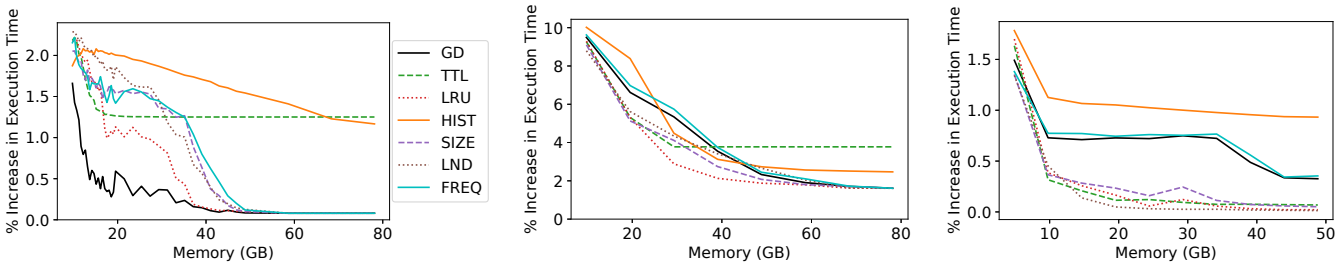
We now present the experimental evaluation of our caching-based keep-alive and provisioning techniques by using function workload traces and serverless benchmarks. Our goal is to investigate the effectiveness of the these techniques under different workload and system conditions.

Setup, Workloads, and Metrics. For evaluating different keep-alive performance with different workload types, we use different trace samples from the Azure Function trace [48], which contains execution times, memory sizes, and invocation-timestamps for more than 50,000 unique functions. Since our goal is to examine performance at a *server* level, we use smaller samples of this trace for realistic server sizes, and replay them in our discrete-event keep-alive simulator. This also allows us to examine the behavior with different *types* of workloads, which is important because our keep-alive policies are designed to be general and workload-agnostic. We use the following three trace samples (more details in the Table 2):

RARE: A random sample of 1000 of the rarest, most infrequently invoked functions. These functions will usually result in cold starts under a classic 10 minute TTL.

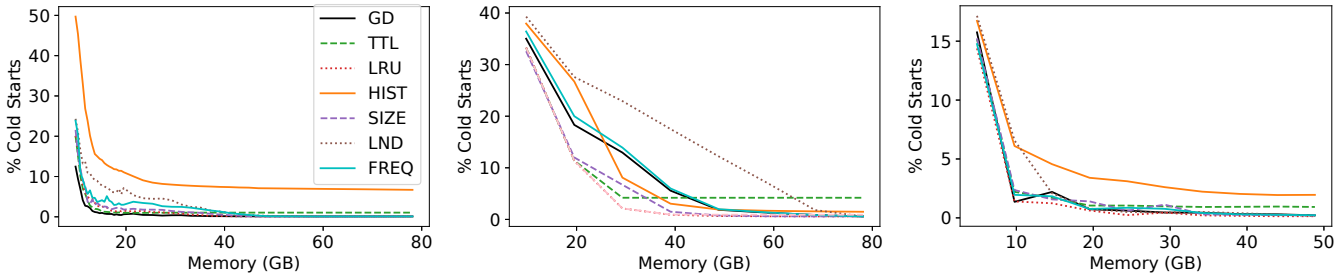
REPRESENTATIVE: A sample of 400 functions, sampled from each quartile of the dataset based on frequency—yielding a more representative sample with higher function diversity.

RANDOM: A random sample of 200 functions.



(a) Representative functions. (b) Rare functions. (c) Random sampling.

Figure 5: Increase in execution time due to cold-starts for different workloads derived from the Azure function trace.



(a) Representative functions. (b) Rare functions. (c) Random sampling.

Figure 6: Fraction of cold-starts is lower with caching-based keep-alive.

Table 2: Size and inter-arrival time (IAT) details for the Azure Function workloads used in our evaluation.

Trace	Num Invocations	Reqs per sec	Avg. IAT
Representative	1,348,162	190 /s	5.4 ms
Rare	202,121	30 /s	36 ms
Random	4,291,250	600 /s	1.8 ms

The FaasCache system is evaluated in Section 7.2. Functions from the FunctionBench [40] suite are used for generating a realistic workload. A single server with 250 GB RAM and 48-core Intel Xeon Platinum 2.10 GHz CPUs is used for running all functions. The server is running modified OpenWhisk (i.e., FaasCache), and Ubuntu 16.04.5.

Adapting the Azure Functions Trace. The format of the original Azure Function trace [48] requires some additional pre-processing and extrapolation for generating a workload. The full dataset consists of 14 days of function invocations, and billions of individual invocations. We use the first day's data, and do not consider functions that are never reused (i.e., with less than two invocations).

The original trace provides memory consumption at the *application* level—with the application made up of multiple functions. Therefore, we evenly split the memory allocation between all functions in an application. The dataset provides invocations in minute-wide buckets. When injecting/replaying the workload, if there is only one invocation in a minute-bucket, it is injected at the beginning of the minute. For multiple invocations, they are equally spaced throughout the minute.

The cold-start overhead of each function is estimated as `maximum - average runtime`, and the execution times provided in the dataset are used for this computation. The dataset does not account for certain important sources of cold-start overheads such as execution environment creation (e.g., Docker). This unfortunately underestimates the cold-start overheads. However, because it applies uniformly to all functions, it preserves the relative performance of the different keep-alive policies, and does not affect the cache hit ratios.

We are interested in two metrics: the cold-start ratio; and the average increase in the execution time due to cold-starts. The increase in execution time is computed by averaging across all function invocations.

7.1 Trace-Driven Keep-Alive Evaluation

In this subsection, we use the Azure function traces to evalaute different keep-alive policies in our discrete-event simulator. We compare all caching-based variants against the default keep-alive policy in OpenWhisk (10 minute TTL). When the server is full, this TTL policy evicts containers in an LRU order. We also evaluate different Greedy-Dual variants: GD is our GDSF policy described in Section 4.1. The others are the caching-based variants described in Section 4.2: LND is Landlord, and FREQ is LFU.

We also compare against the histogram based keep-alive policy in [48], which is the state of the art technique. We have reproduced this policy (HIST) from the details in the paper, and have implemented it in a "best-effort" manner without any knowledge of the optimizations in the actual implementation. This is effectively a "TTL+Prefetching" policy: it uses a histogram of *inter-arrival times* to predict future function invocations and eagerly evict warm functions. It uses timeseries-forecasting to capture temporal locality, but does not consider the other function characteristics such as function size and initialization cost. The IAT, computed by taking a function's execution time plus the subsequent idle time, between each actual invocation is recorded in minute granularity buckets, tracking up to four hours between executions. The policy uses ARIMA modeling for those invocations that fall outside this four hour window, we chose not to implement this specific feature due to its complexity, and the fact that it accounted for a minor fraction (~0.56%) of all invocations. From these buckets, a function's coefficient of variation (CoV) is calculated using Welford's online algorithm [56]. When the function's IAT is predictable (CoV ≤ 2), the function's historical/customized preload and TTL time are used. Otherwise the function has a generic TTL of two hours. When an invocation is anticipated, it is brought into memory and kept there until its TTL expires. A function is evicted when the policy predicts it will not have an invocation in the near future.

The increase in execution time for different traces and for different cache sizes is shown in Figure 5. The increase in execution time is the cold-start overheads averaged across all invocations of every function, and captures the user-visible response-time.

For the representative trace (Figure 5a), Greedy-Dual reduces the cold-start overhead by more than 3× compared to TTL for a wide range of cache sizes (15–80 GB). Interestingly, it is able to achieve a low overhead of only 0.5% at a much smaller cache size of 15GB, compared to other variants, which need 50 GB to achieve similar results—a reduction of cache size by more than 3×. For rare functions (Figure 5b), caching-based approaches such as LRU reduce the cold-start overhead by 2× compared to TTL for cache sizes of 40–50 GB. This shows that for rare functions, recency is a more pertinent characteristic, and the complex four-way tradeoff used in Greedy-Dual is not necessarily ideal in all workload scenarios. For this workload, the HIST policy outperforms TTL, as reported in [48]. However, it results in 50% higher cold-start overhead compared to caching-based approaches. Furthermore, because HIST uses only inter-arrival times, it is unable to perform well with heterogeneous representative workloads (Figure 5a).

Finally, the randomly sampled trace has a large number of infrequent functions because of the low probability of selecting the heavy-hitting functions. In Figure 5c, the recency component again dominates, and we see LRU outperforming other variants. The equivalence of LRU and TTL-based caching for rare objects has been noted [18, 36], which explains their similar behavior seen in Figure 5c.

Result: *For representative, diverse workloads, our GD policy can improve the performance and shrink cache sizes by up to* 3×. *For more homogeneous workloads, LRU can outperform current TTL-based approaches by* 2×.

We can observe from Figure 5 that the increase in execution time is generally small (< 10%). This is because of two main factors: the evaluation metric chosen, and the properties of the workload trace. The execution time is averaged across *all* function invocations. However, serverless workloads consists of a large number of very frequently invoked functions. The performance of these functions is generally not affected by keep-alive policies, since any policy is going to keep them in the cache because of their high frequency. Thus, the difference between non work-conserving policies such as TTL and Greedy-Dual is masked because of the frequent and popular functions. For instance, the average inter-arrival time for all three workloads is less than 36ms, or about 27 function invocations per second. Thus the server is overloaded, and TTL does well even though it is not work-conserving. As the IAT grows, the effectiveness of work-conserving caching-based approaches increases compared to TTL, as we shall see in the next subsection.

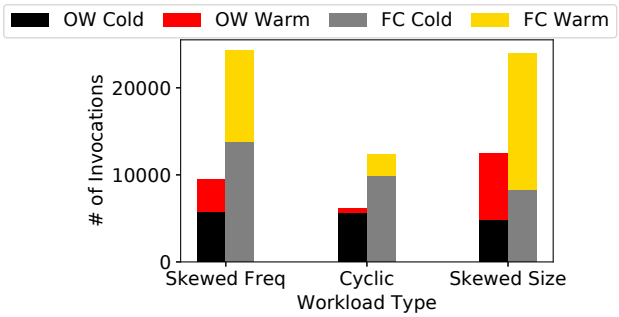


Figure 7: FaasCache runs 50 to 100% more cold and warm functions, for skewed workload traces.

We see a similar relation and behavior in the miss-ratio curves shown in Figure 6. Due to function heterogeneity, the cold-start overheads are not strictly correlated with cache miss ratios, and thus the differences between policies is different compared to the previously described actual cold-start overheads. Classic miss-ratio curves do not consider the miss *cost* (i.e., initialization cost), which is an important metric that is optimized by the Greedy-Dual approach. Thus in general, even in object caching contexts, miss-ratio curves deviate from the actual performance—a behavior that we also observe.

7.2 OpenWhisk Evaluation

In this subsection, we evaluate the performance of the FaasCache system on real functions. We focus on the performance of FaasCache's Greedy-Dual keep-alive implementation, and compare it to the vanilla OpenWhisk system which uses a 10 minute TTL.

In contrast to the previous subsection in which we showed the average performance for different cache sizes, we will now also focus on the inverse problem: for a fixed server size, how much more load can be handled with FaasCache? By leveraging Greedy-Dual caching, FaasCache is able to reduce cold-starts. This also reduces the number of *dropped* requests.

OpenWhisk buffers and eventually drops requests if it cannot fulfill them. Because FaasCache more effectively selects evictions, its higher hit rate results in functions finishing faster, allowing more functions to be executed in the same time frame. To examine the effect of Greedy-Dual keep-alive on cold-start and dropped

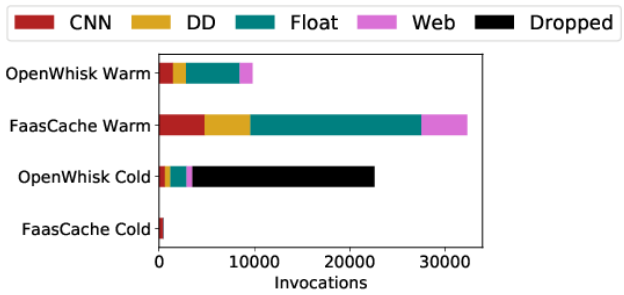


Figure 8: FaasCache increases warm-starts by more than 2×, which also reduces system load and dropped functions.

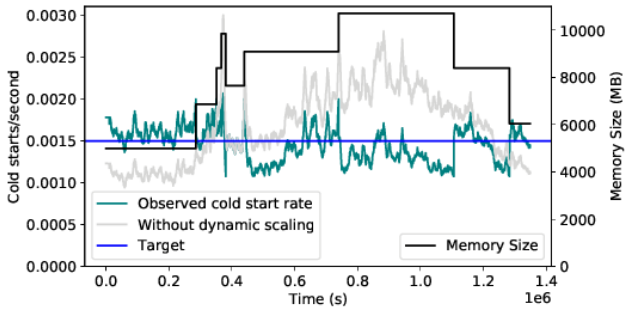


Figure 9: With dynamic cache size adjustment, the cold starts per second are kept close to the target (horizontal line), which reduces the average server size by 30%.

requests, we use a workload trace comprising of four different functions: Disk-bench, ML inference, Web-serving, and Floating-point from Table 1.

In Figure 7, we use different kinds of *skewed* workloads: with a single function having a different frequency, a cyclic access pattern, and a skewed workload with 2 sizes. We see that FaasCache's keep-alive can increase the number of warm invocations by between 50 to 100% compared to OpenWhisk's TTL. The difference in the total number of requests served (warm+cold) is because OpenWhisk drops a significant number of requests due to its high cold-start overhead and resultant system load. Thus with FaasCache, the total number of requests that are served also increases by 2×.

Next, we use the skewed frequency workload and use functions from Table 1 to evaluate the impact on real applications. To generate the workload, the CNN, DD, and Web-serving functions have an inter-arrival time of 1500 ms, and the Floating-point function has a lower IAT of 400 ms. Figure 8 shows the breakdown of different function invocations for this workload on a 48 GB server. Interestingly, OpenWhisk drops a significant number (50%) of requests due to the its high cold-start overheads. FaasCache increases the warm requests by more than 2×. Interestingly, the *distribution* of warm starts is also different. FaasCache's Greedy-Dual policy prioritizes functions with higher initialization times, but penalizes those with large memory footprints. Because the floating-point function has a high initialization overhead (Table 1), it sees a 3× increase in hit-ratio compared to OpenWhisk. *In practical terms, the improvement in keep-alive results in a 6× reduction in the application latency.*
Result: *FaasCache can increase the number of warm-starts by 2× to 3× depending on the function initialization overheads and workload skew. This results in lower system load, which increases the number of requests FaasCache can serve by 2×.*

7.3 Effectiveness of Provisioning Policies

All our previous results have been with a statically allocated server, and we now illustrate the effectiveness of our dynamic vertical scaling policy described in Section 5.2. The goal is to dynamically adjust the cache size based on the workload. Our policy seeks to keep the miss speed (cold starts per second) close to a pre-specified target. This is shown in Figure 9—the target is 0.0015 misses per second. In this experiment, the cache resizing is done only when the miss speed error exceeds 30%, and we can see that the cache size increases with the miss speed, and decreases with it. Without the dynamic scaling, a conservative provisioning policy would result in a constant, 10,000 MB size. In contrast, the average cache size with our proportional controller is less than 7,000 MB. This 30% reduction means that FaaS providers can reduce their provisioned resources without compromising on performance. The freed-up resources can be used to accommodate additional cloud workloads (such as co-located VMs and containers). Our dynamic scaling is extremely conservative: increasing its agressiveness by reducing the error tolerance below 30% will reduce average server size, but we seek to avoid the resultant small changes to memory-size to minimize fragmentation.

8 RELATED WORK

Function Keep-alive. Mitigating cold-starts is one of the central performance problems in FaaS, and has received commensurate attention in both academia and industry. The initialization or startup time of functions can be reduced by reducing container startup overheads [16, 45, 46], or deploying functions inside ultra-light containers, VMs, or unikernels [15, 42]. While these mechanisms can reduce the cold-start overhead associated with the virtual environment creation, the other sources of overheads remain, such as losing all application initialized variables, cached files, etc. As we have shown, keep-alive essentially serves the role of caching, and fast startup only reduces the "miss" penalty, and does not eliminate it.

Catalyzer [29] implements new mechanisms for checkpointing and restoring application and sandbox state, which significantly reduce the initialization cost of functions deployed in their gvisor-based sandbox environment. Our approach is complementary to these techniques, since we focus on retaining the entire execution environment instead of optimizations for restoring/recreating it. Keep-alive policies can be combined with these optimized mechanisms to improve system-wide performance even further.

Principled keep-alive policies for functions have recently gained attention: the recent dataset and policy from the Azure function trace [48] shows the importance and effectiveness of keep-alive policies. In contrast to our work, their policy does not take the function size into consideration, and uses a time-series predcition approach (effectively capturing recency and frequency), and combines it with a predictive "prefetching" approach. As we have shown,

function sizes are a crucial characteristic, and the use of caching allows the use of advanced analytical and modeling approaches for serverless computing in general. Earlier work has focused on simple "warm container pools" [41], in which Kubernetes cluster runs a certain number of warm containers for functions. Our caching-based policies take this one step further and decide *which* container to keep-alive, and for how long. Polling to keep cloud functions warm has also been a popular method [2, 6].

Our work considers functions individually—function scheduling with DAG based approaches [22] is effective for function-chains, and are orthogonal and complementary to our work. Hiding function latency using data caching (such as redis) for database applications is investigated in [33]. The ENSURE [52] system handles keep-alive and resource provisioning for CPU resources using queueing theory techniques. Our focus is on memory-constrained keep-alive and provisioning, and CPU-focused approaches are complementary to our work.

9 DISCUSSION

In this section, we reflect on how our ideas fit into the broader serverless computing ecosystem.

Impact on colocated applications. The short execution lifecycles of serverless functions makes them a good workload to colocate with other long-running containers, VMs, batch-jobs, etc. Such workload management architectures result in an additional layer of performance tradeoffs: the keep-alive policies not only influence function performance, but also the performance of other colocated applications. Our provisioning policies in Section 5 can be used to find the appropriate keep-alive cache size based on the performance vs. memory-consumption tradeoff captured in the hit-rate curves. Ultimately, the tradeoff between function and other colocated application performance is determined by their utility and revenue for the cloud provider. Nevertheless, our provisioning policies can provide a principled way to examine these tradeoffs, and is part of our future work.

Cluster-level analysis. Cluster-level function load-balancing and scheduling also affects keep-alive. Load-balancing policies determine the function load and distribution of function characteristics on servers. The function workload characteristics have a major influence on performance, as we have extensively discussed in Section 7 (e.g., Figure 5). For instance, a stateful load-balancing policy which runs a function on the same subset of servers will result in better temporal locality, which in turn improves keep-alive effectiveness. On the other hand, randomized load-balancing is simpler to implement and scale, but offers worse temporal locality to individual servers. We have deliberately focused our techniques and evaluation on a single-server setting, in order to provide modular and easily reproducible policies.

Explicit initialization. There are many techniques for reducing the initialization cost, which can be combined with our policies. The cold-start overhead can also be addressed by *explicit initialization* of functions, in which the initialization code is provided as a separate, explicit call-back. For instance, OpenWhisk supports an `init` call into the function runtime, which can be executed before the function is triggered with the `run` call. This explicit initialization allows functions to be pre-warmed, and can be used to reduce the cold-start overhead. However, explicit initialization is not common—our empirical investigation into FaaS benchmarks [40] and official examples showed that applications do not use this functionality. We speculate that the slow adoption is due to the subtle differences in the various cloud function APIs, serverless platforms, and runtimes. Nevertheless, it can be a powerful technique to amortize expensive operations such as package imports and downloading data dependencies, and increase the effectiveness of keep-alive policies even further. By separating out the initialization code and actual function code, explicit initialization can also increase the potency of function prefetching [48].

10 CONCLUSION

The main insight in this paper is the equivalence between function keep-alive and object caching. This can have far reaching consequences for cloud resource management policies. We showed that classic size and frequency-aware caching algorithms such as Greedy-Dual can be adapted to yield effective and principled keep-alive policies. The tradeoff between server memory-utilization and cold-start overheads can also be analyzed through hit-ratio curves, which can also be used for dynamic resource allocation. FaasCache, our OpenWhisk-based system, implements these caching-based techniques. We hope that our caching analogy opens the door to more caching-based serverless systems and analysis.

ACKNOWLEDGMENTS

Our sincere thanks and appreciation goes to the ASPLOS reviewers and our shepherd Ana Klimovic, for their extremely insightful and helpful comments and suggestions that have significantly improved the quality of this paper. Comments from Mohammad Shahrad and other authors of [48] helped us understand and clarify some of the nuances of the Azure dataset. Thanks also to the artifact evaluation committee and its volunteers for comments which have improved the usability of the FaasCache software artifact. This work was supported by startup funds from Indiana University.

A ARTIFACT APPENDIX

A.1 Artifact Check-List (Meta-Information)

- **Program:** FaasCache
- **Data set:** see A.2.4
- **Run-time environment:** Ubuntu 16.04.5
- **Hardware:** 250 GB RAM, 48 cores
- **Experiments:** Simulation & OpenWhisk implementation
- **How much disk space required (approximately)?** 10 GB
- **How much time is needed to prepare workflow?**2 hours
- **How much time is needed to complete experiments (approximately)?** 6 hours
- **Publicly available?** Yes
- **Archived (provide DOI)?** 10.5281/zenodo.4321766

A.2 Description

We have two main software artifacts.

The first is a discrete-event simulator for FaaS workloads written in Python. This simulator implements various keep-alive policies,

all of which are described in the full paper. Its inputs are workload trace files that are publicly available at the Azure trace site, and serialized into a custom format by scripts in *code/split/gen*. The simulator takes server memory size, keep-alive policy, input trace file as arguments, and outputs various statistics on warm and cold starts, memory usage, and other accounting information. These outputs are run through the data-graphing scripts in *code/split/plotting*, to produce figures such as Figure 5 in the paper.

The second artifact is a custom OpenWhisk (i.e., FaasCache). It is a drop-in replacement of OpenWhisk with the same installation procedures. It optimizes OpenWhisk scheduling with GD keep-alive policy. FaasCache or vanilla OpenWhisk can both be used to generate the information from Table 1 in the paper.

Our artifact also includes some FaasCache performance tests in *faas-keepalive/code/wsk-actions/load-test/traces*. These can be run with scripts in *faas-keepalive/code/wsk-actions/load-test* that invoke LookBusy work-simulating FaaS functions at regular intervals to load test the FaasCache design.

Full details on how to run these two artifacts are in Section A.4 below. For brevity, portions talking about the simulator or OpenWhisk load generation will be referencing folder *faas-keepalive*. Those sections on running the customized OpenWhisk server are based in folder *openwhisk-caching*.

A.2.1 How to Access. 10.5281/zenodo.4321766

The code for both experiments is in the zenodo zip, but also available on GitHub.
Simulator code is here: https://github.com/aFuerst/faascache-sim
FaasCache OpenWhisk implementation:
https://github.com/aFuerst/openwhisk-caching/commit/38ff898d45da57726da38c00f735cb449e7f8595

A.2.2 Hardware Dependencies. To reproduce the FaasCache load test results, it is recommended to use sufficient RAM and CPU cores. We used 64 GB RAM, and 48 CPU cores. The simulator needs 1 GB RAM per core, and is embarrassingly parallel and is mainly limited by total system memory.

A.2.3 Software Dependencies.

(1) Python 3.7+
(2) Docker
(3) Java

A.2.4 Data Sets. The original data is pulled from the dataset described in this markdown file:
https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md

The representative trace used in the paper is in the zip, called *392-b.pckl*. Pre-computed simulator results are also in the zip, in the folder *392-pckls*. These are high-resolution results, memory-wise, as the simulation is compute-intensive (i.e. slow).

A.3 Installation

A.3.1 Simulator Setup. The original trace dataset can be found here:
https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md
The simulator code is available here, as well as in the zip:
https://github.com/aFuerst/faascache-sim

The script *./code/split/trace_split_funcs.py* will combine the first day's info into one file. Edit *datapath* and *store* in the script to adjust input and output locations.

./code/split/gen_representative_trace.py will create traces that are generally representative of the larger trace sample. Edit *datapath* and *store* in the script to adjust input for trace CSVs and split pickles (made by *trace_split_funcs.py* above) respectively. *save_dir* points to the folder where the resulting combined trace(s) will be saved. Change lines 131-133 if you want specific trace sizes.

./code/split/gen_rare.py will create traces using the rarest half and quarter of functions. You can edit line 99 to adjust which quartiles it picks functions from.

A.3.2 FaasCache OpenWhisk Setup. FaasCache OpenWhisk implementation:
https://github.com/aFuerst/openwhisk-caching/commit/38ff898d45da57726da38c00f735cb449e7f8595
Install the OpenWhisk CLI:
https://github.com/apache/openwhisk#quick-start
OpenWhisk testing code (a subset of FaasCache sim repo):
https://github.com/aFuerst/faascache-sim/tree/master/code/wsk-actions

All the edits made to the OpenWhisk source for FaasCache are located in the file *core/containerpool/ContainerPool.scala*.

Build the dockerfile located at *code/wsk-actions/py/Dockerfile* with the name/tag alfuerst/wsk-py-pybuild. This name is not required, but you will have to change the next script to use the name you pick. *./code/wsk-actions/py/build.sh* will create zip packages for all the actions that OpenWhisk can use.

To run the load tests on OpenWhisk you will also have to build the LookBusy Docker container in *./code/wsk-actions/load-test/lookbusy*. Make sure the new Docker container is added as a runtime to the OpenWhisk *ansible/files/runtimes.json*. The default AI container OpenWhisk uses still may be missing packages, if this is the case you will also have to build the Dockerfile at *wsk-actions/py/cust_ai* and add it as a custom runtime.

Edit the *./sample-app.conf* in the root and put it in *./bin/*. Rename it to *application.conf*. OpenWhisk requires this configuration file to run and allow function creation. You can adjust *container-pool.user-memory* depending on local resources.

./openwhisk-caching/blob/master/run.sh will build and run the custom OpenWhisk.

Make sure the *.conf whisk.user* info matches between the scripts that talk to OpenWhisk.

code/wsk-actions/load-test/wsk_interact.py contains helper functions that interact with the OpenWhisk CLI to set up functions and authentication. If you use a different username or auth key then you will need to edit this file.

A.4 Experiment Workflow

A.4.1 FaasCache Simulation. The *./code/run_sim.sh* file will run a trace in the simulator and graph the results.

Step 1. trace_dir => folder where trace file needs to be
Step 2. trace_output_dir => folder where sim results will end up
Step 3. log_dir => file where sim log data will end up

The location of the trace output and log output **must** be different.

Edit the number of functions in *./code/run_sim.sh* to match the number of functions in the trace you want to run (this number will be in the file name). Make sure the trace file letter (-b-, etc.) matches line 65 of *./code/split/many_run.py*, this is set by the trace generation script.

Running the Simulator. *many_run.py* executes instances of the simulator (LambdaScheduler) in parallel and saves the results of each simulation to a pickle file. The inputs and outputs of many_run.py are documented in the file itself. *LambdaScheduler.py* contains all the major simulator code, and the helper classes are located in:

Step 1. *LambdaData.py* - represents a Serverless function, holding a name, memory size, and cold and warm runtime lengths
Step 2. *Container.py* - holds a LambdaData and tracks if it is warm or cold and when it was last accessed

Each function execution enters with *runActivation* which does the following (function names are emphasized)

Step 1. *cleanup_finished* - remove containers that have exceeded their TTL (if applicable)
Step 2. *PreWarmContainers* - prewarm containers when calculated by the histogram policy (if that policy is being simulated)
Step 3. *track_activation* - book-keeping for the histogram policy (if that policy is being simulated)
Step 4. *find_container* - search for a container that matches the function being invoked that currently isn't running another function
  (a) *cache_miss*
    (i) create a new *Container* for the function
    (ii) evict low-priority containers to make room if necessary
    (iii) assign the function to it, and set the cold running time
  (b) Cache hit - a requisite container was found, simply assigns the function to it and sets the warm running time
Step 5. *calc_priority* - update the priority of the function that was just called and any functions of that type that are currently in-memory

Simulation Analysis. These two scripts compute the number of cold/warm starts, and the global increase in execution time.
compute_policy_results.py
compute_mem_usage.py

Plotting Results. These two perform the plotting for figures 5 & 6 respectively: *plot_run_across_mem.py*, and *plot_cold_across_mem.py*.

The folder contains several other plotting scripts for analyzing the simulation results, but we chose not include those plots in the paper.

A.4.2 FaasCache OpenWhisk. You can follow the items in run.sh to run individual actions or run

./code/wsk-actions/load-test/testing/find_avgs.py to get average run times for all the different actions.

./code/wsk-actions/load-test/gen_litmus.py will generate the litmus test pckls for the full OpenWhisk tests. Then run *code/wsk-actions/load-test/sub_litmi.py* to invoke the litmus test.

Cold vs warm hit metrics are output to OpenWhisk log (stdout from sh/jar). Make sure to pipe the output to a file. You can grep on *cold hits:* to look at current results.

If you run the any of the litmus test, stop the test after 2 hours. OpenWhisk may or may not complete all invoked actions, stopping it significantly late is ok, just wastes time. Then grep for the first hit event, record the time, then grep for the event closest to 2 hours later. This will match how the paper results were gathered.

A.5 Evaluation and Expected Results

Results should be similar to those in the paper.

The simulator results will not be identical if a new trace sampling is used, and if a coarser grained memory step is used. We used 500 MB steps, but this dramatically increases needed simulation time.

Timings and numbers for the FaasCache OpenWhisk implementation will vary marginally due to the stochastic nature of web request handling and the inner workings of OpenWhisk.

REFERENCES

[1] [n.d.]. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.
[2] [n.d.]. Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues. https://serverless.com/blog/keep-your-lambdas-warm/.
[3] [n.d.]. PID Controllers. https://en.wikipedia.org/wiki/PID_controller.
[4] 2015. Docker. https://www.docker.com/.
[5] 2017. How long does AWS Lambda keep your idle functions around before a cold start? https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810.
[6] 2018. Lambda Warmer: Optimize AWS Lambda Function Cold Starts. https://www.jeremydaly.com/lambda-warmer-optimize-aws-lambda-function-cold-starts/.
[7] 2019. AWS Lambda predictable start-up times with provisioned concurrency. https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/.
[8] 2019. Azure Functions Warm-up trigger. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-warmup.
[9] 2020. Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.
[10] 2020. AWS Lambda. https://aws.amazon.com/lambda/.
[11] 2020. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[12] 2020. Google Cloud Functions. https://cloud.google.com/functions.
[13] 2020. Google Cloud Functions Tips and Tricks. https://cloud.google.com/functions/docs/bestpractices/tips.
[14] 2020. OpenFaaS : Server Functions, Made Simple. https://www.openfaas.com.
[15] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 419–434.
[16] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. *USENIX ATC* (2018), 14.
[17] Erwan Alliaume and Benjamin Le Roux. 2018. Cold start / Warm start with AWS Lambda. https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/.
[18] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. 2017. Adaptive TTL-based caching for content delivery. In *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*. 45–46.
[19] Pei Cao and Sandy Irani. 1997. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*. 15.
[20] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, Vol. 2018.
[21] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing - SoCC '19*. ACM Press, Santa Cruz, CA, USA, 13–24. https://doi.org/10.1145/3357223.3362711
[22] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. 2019. In Search of a Fast and Efficient Serverless DAG Engine. *arXiv:1910.05896 [cs]* (Oct. 2019). http://arxiv.org/abs/1910.05896 arXiv: 1910.05896.
[23] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 65–76. https:

//doi.org/10.1145/3369583.3392683

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

[25] Kai Cheng and Yahiko Kambayashi. 2000. LRU-SP: a size-adjusted and popularity-aware LRU replacement algorithm for web caching. In *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000*. IEEE, 48–53.

[26] Ludmila Cherkasova. 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. In *HP Labs Technical Report 98-69 (R.1)*.

[27] Ludmila Cherkasova and Gianfranco Ciardo. 2001. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *High-Performance Computing and Networking*, G. Goos, J. Hartmanis, J. van Leeuwen, Bob Hertzberger, Alfons Hoekstra, and Roy Williams (Eds.). Vol. 2110. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–123. https://doi.org/10.1007/3-540-48228-8_12 Series Title: Lecture Notes in Computer Science.

[28] Ludmila Cherkasova and Gianfranco Ciardo. 2001. Role of aging, frequency, and size in web cache replacement policies. In *International Conference on High-Performance Computing and Networking*. Springer, 114–123.

[29] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[30] Gil Einziger, Roy Friedman, and Ben Manes. 2017. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[31] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, and Shuvo Chatterjee. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. *USENIX ATC* (2019), 15.

[32] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2012. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–26.

[33] Bishakh Chandra Ghosh, Sourav Kanti Addya, Nishant Baranwal Somy, Shubha Brata Nath, Sandip Chakraborty, and Soumya K. Ghosh. 2019. Caching Techniques to Improve Latency in Serverless Architectures. *arXiv:1911.07351 [cs]* (Nov. 2019). http://arxiv.org/abs/1911.07351 arXiv: 1911.07351.

[34] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[35] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16))*. 351–364.

[36] Bo Jiang, Philippe Nain, and Don Towsley. 2018. On the convergence of the ttl approximation for an lru cache under independent stationary request processes. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 4 (2018), 1–31.

[37] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. 2019. SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion - UCC '19 Companion*. ACM Press, Auckland, New Zealand, 43–50. https://doi.org/10.1145/3368235.3368839

[38] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 445–451.

[39] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv:1902.03383 [cs]* (Feb. 2019). http://arxiv.org/abs/1902.03383 arXiv: 1902.03383.

[40] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504. https://doi.org/10.1109/CLOUD.2019.00091 ISSN: 2159-6182.

[41] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *arXiv:1903.12221 [cs]* (March 2019). http://arxiv.org/abs/1903.12221 arXiv: 1903.12221.

[42] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[43] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, Zurich, 181–188. https://doi.org/10.1109/UCC-Companion.2018.00054

[44] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *USENIX FAST*, Vol. 3. 115–130.

[45] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Vadim Sukhomlinov, and Naren Nayak. 2019. Agile Cold Starts for Scalable Serverless. *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2019), 6.

[46] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. USENIX ATC (2018), 14.

[47] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[48] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. (July 2020), 205–218. http://arxiv.org/abs/2003.03423

[49] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[50] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 33, 17 pages. https://doi.org/10.1145/3302424.3303945

[51] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. 2017. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 55–67.

[52] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. (2020), 10.

[53] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thommes. 2017. The SPEC cloud group's research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*. ACM Press, Las Vegas, Nevada, 1–4. https://doi.org/10.1145/3154847.3154848

[54] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 95–110.

[55] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference*. 133–146.

[56] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4, 3 (1962), 419–420. https://doi.org/10.1080/00401706.1962.10490022 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022

[57] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. 2014. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 335–349.

[58] N. Young. 1994. The K-server dual and loose competitiveness for paging. *Algorithmica* 11, 6 (June 1994), 525–541. https://doi.org/10.1007/BF01189992

[59] Neal E Young. 2002. On-line file caching. *Algorithmica* 33, 3 (2002), 371–383.

[60] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 785–798. https://www.usenix.org/conference/atc20/presentation/zhang-yu

[61] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference*. 785–798.