IceBreaker: Warming Serverless Functions Better with Heterogeneity

Rohan Basu Roy
Northeastern University
Boston, Massachusetts, USA

Tirthak Patel
Northeastern University
Boston, Massachusetts, USA

Devesh Tiwari
Northeastern University
Boston, Massachusetts, USA

ABSTRACT

Serverless computing, an emerging computing model, relies on "warming up" functions prior to its anticipated execution for faster and cost-effective service to users. Unfortunately, warming up functions can be inaccurate and incur prohibitively expensive cost during the warmup period (i.e., keep-alive cost). In this paper, we introduce IceBreaker, a novel technique that reduces the service time and the "keep-alive" cost by composing a system with heterogeneous nodes (costly and cheaper). IceBreaker does so by dynamically determining the cost-effective node type to warm up a function based on the function's time-varying probability of the next invocation. By employing heterogeneity, IceBreaker allows for more number of nodes under the same cost budget and hence, keeps more number of functions warm and reduces the wait time during high load. Our real-system evaluation confirms that IceBreaker reduces the overall keep-alive cost by 45% and execution time by 27% using representative serverless applications and industry-grade workload trace. IceBreaker is the first technique to employ and leverage the idea of mixing expensive and cheaper nodes to improve both service time and keep-alive cost for serverless functions – opening up a new research avenue of serverless computing on heterogeneous servers for researchers and practitioners.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **n-tier architectures**; • **General and reference** → **Performance**.

KEYWORDS

Serverless Computing, Cloud Computing, Cold Start, Keep-alive Cost, Heterogeneous Hardware

ACM Reference Format:
Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507750

1 INTRODUCTION

Background and Motivation. Serverless computing paradigm is becoming increasingly prevalent in multiple domains including web applications, microservices, latency-critical workloads, data processing, and machine learning [18, 27, 32]. This is because serverless has lowered the barrier to entry for end users by enabling them to make use of cloud computing resources in a pay-per-use model with elastic scaling, and without worrying about the management and provisioning of computing resources.

Serverless functions are spawned on serverless instances (container or microVM [1]). They suffer from the problem of **cold starts**, which is the overhead (also known as **cold start time**) of setting up an instance, and loading an application logic into the memory [11, 31, 55, 68]. This overhead can be a significant fraction of a serverless function execution time as serverless mainly attracts short running tasks (order of seconds) due to limited resources of serverless instances.

Existing Methods for Mitigating Cold Starts. The most intuitive solution to avoid cold starts is to keep serverless functions instances set up and loaded in the memory so that when a function gets invoked, it can avoid the cold start overhead [57, 68]. This process of keeping a function alive in memory is called a **warm up**. However, keeping functions alive takes up memory and incurs a **keep-alive cost** for the serverless providers, increasing the capital budget. Keep-alive cost is defined as the product of three terms: the cost per unit memory per unit time for reserving the server, the memory required to accommodate the function instance, and the time for which the function is kept alive. If a function is invoked while it is kept alive in the memory, it undergoes a **warm start**, and thus executes without incurring the cold start overhead.

As serverless workloads are becoming diverse with different invocation patterns, it is becoming harder to decide how long to keep a function alive and when to warm it up to avoid cold start [57, 66]. Serverless providers usually follow a 10-minute keep-alive policy after a function invocation, but such a fixed policy does not ensure cold start mitigation for diverse invocation characteristics [57, 68]. To reduce the effect of cold starts, previous works have come up with strategies like overbooking server instances and pre-caching different parts of an application [3, 34, 50, 51]. However, these techniques increase the capital cost [2, 12]. Other strategies that predict the function inter-arrival time to accordingly warm up a function instance also increase the keep-alive cost [19, 57]. Even though complete servers are reserved for creating a serverless computing platform, reducing the keep-alive cost is important as it directly minimizes memory wastage. This helps the servers accommodate more functions.

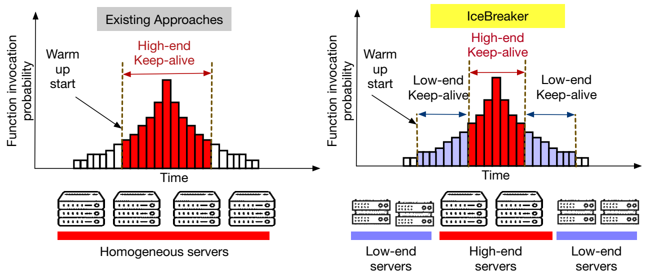

Figure 1: IceBreaker employs heterogeneity to reduce the keep-alive cost and service time.

It is in the best interest of the service provider to reduce the keep-alive cost. While the end-users want to reduce the **service time** (time to serve a complete serverless function request from invocation to the completion of execution). Depending on the scenario, the service time can be (1) just the execution time of a function (if a function undergoes a warm start), or (2) the sum of execution time and cold start time (if a function undergoes a cold start), or (3) the sum of the execution time, cold start time and wait time (if currently the servers are occupied with executing functions). There is a requirement for a technique that reduces both keep-alive cost and service time.

Limitations of Existing Approaches. The current approaches have two major limitations:

I. Fixed keep-alive cost throughout the keep-alive period. A function's probability of arrival (invocation) is not constant over time during its keep-alive period. But, the current approaches pay a constant amount of dollars per second during the time a function is kept alive after a warm-up. Unfortunately, this current approach leads to high keep-alive costs for serverless functions. If the probability of arrival is not constant during the warm-up period, an effective scheme should adjust itself dynamically to match the incurred keep-alive cost with the function arrival probability.

II. Lack of robustness to frequently changing patterns and concurrency degree: Existing approaches use histogram-based or ARIMA time-series-based mechanisms for predicting the next arrival (invocation) of a function [57]. While useful and effective for many functions, existing schemes are not robust to frequently changing patterns and hence, often mispredict function invocations, especially for hard-to-predict and infrequently functions [57, 65]. Furthermore, existing schemes optimistically assume that the concurrency degree of future invocations of a function is identical to its last invocation – concurrency degree refers to the number of simultaneous invocations of the same function. Our characterization, using a production-level workload trace, demonstrates that the inability to handle this leads to mispredictions, increased keep-alive cost, and higher service time.

IceBreaker: Key Insight and Approach. The key insight behind IceBreaker is the observation that a function's probability of arrival (invocation) is not constant over time during its keep-alive period, and hence, the cost of keeping it alive should not remain constant. Unfortunately, current approaches pay a constant amount of dollar per second during the whole time a function is kept alive after a warm-up. Unfortunately, this current approach leads to high keep-alive costs for serverless functions. IceBreaker introduces the idea of server-heterogeneity for keeping-alive functions based on their anticipated arrival probability. Specifically, IceBreaker warms up a function on a low-end server when the probability of arrival is relatively low and prioritizes warming functions with higher arrival probability on high-end servers.

On the positive side, keeping a function warm on low-end servers incurs relatively less cost to the service provider because the low-end servers are cheaper compared to high-end servers. Hence, functions with less arrival probability can be kept warm at a lower per unit time cost. When their arrival probability increases or others functions with higher arrival probability are available, they can be warmed on the high-end servers, and hence, high-end servers are utilized more cost-effectively (Fig. 1).

On the flip-side, low-end servers are slower and hence, when functions with low arrival probability receive an invocation on the low-end servers, the execution time will be relatively longer. This can be particularly undesirable for the end-users. However, since the low-end servers are relatively cheaper, the service provider can potentially afford more aggregated memory capacity to keep functions warm. This enables service providers to lower the waiting time for functions, and hence, decrease the overall service time despite of a potential increase in the execution time – thus, helping users.

Furthermore, our characterization of real-world serverless functions provides experimental evidence that for many functions, a warm start on a low-end server can be still faster than cold start execution on a high-end server. This is because cold-start is comparable to execution time for many functions, and hence, the ability to provide a warm start (zero cold start time) on low-end servers reduces the overall service time despite of an increase in the execution time. Overall, IceBreaker experimentally demonstrates the benefit of using a mixture of low-end and high-end servers for both the service providers and end-users, which is a new resource provisioning strategy for serverless platforms that has not been explored by prior works.

Overall, IceBreaker makes the following key contributions:

- IceBreaker is a novel function pre-warming and keep-alive scheme for serverless functions that exploit server-heterogeneity to lower the keep-alive cost and reduce the service time. IceBreaker is the first work to propose using a heterogeneous mix of servers (high-end and low-end). IceBreaker appropriately decides where to warm up serverless functions to gain most benefit while reducing the keep-alive cost.
- The key to the beneficial realization of IceBreaker's heterogeneity idea requires an effective function-invocation prediction mechanism. Unfortunately, our results show that current state-of-the-art function-invocation prediction mechanisms (e.g., ARIMA time-series and histogram-based prediction) are not robust to frequent changes in the function invocation patterns, and do not

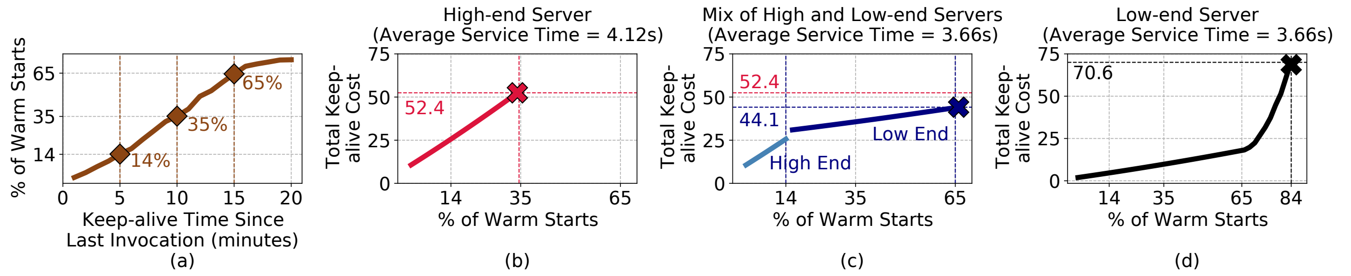

Figure 2: A heterogeneous mix of low and high-end servers can reduce both service time (due to more warm starts) and keep-alive cost than only high-end server. On only low-end server, the keep-alive cost increases for maintaining the same service time as that of the heterogeneous mix of servers (keep-alive cost expressed as a % of maximum on high-end server).

handle the invocation concurrency degree well. IceBreaker's designs and implement a novel Fast Fourier Transformation based prediction mechanism that removes these limitations from existing works – and is experimentally shown to outperform existing strategies even on homogeneous clusters by up to 27% for the most recent production-level serverless function trace.

- IceBreaker designs and implements a "utility-driven" warm-up and keep-alive strategy targeted for a heterogeneous mix of servers. IceBreaker assigns a "utility score" to each function based on several important factors that capture arrival probability, prediction of next arrival and expected concurrency degree, function's relative benefit in execution time across different server types, and keep-alive cost. IceBreaker combines its prediction mechanism and utility estimation to determine the appropriate location for warming a function and the duration of its keep-alive period.
- Our evaluation demonstrates that IceBreaker reduces the keep-alive cost by 60% and improves the service time by 40%, outperforming all competing state-of-the-art approaches [19, 57]. IceBreaker outperforms the next-best technique by 45% and 27% points in terms of keep-alive cost and service time. Our evaluation also demonstrates that IceBreaker is also more effective than existing schemes in predicting the invocations of infrequent and hard-to-predict functions by 10% – a notable challenge in serverless computing domain [57, 65].

IceBreaker's open-source artifact is available at: https://zenodo.org/record/5748667

2 ICEBREAKER: BACKGROUND AND MOTIVATION

In this section, first, we provide concrete quantitative examples to motivate the benefits of introducing heterogeneity in a serverless execution environment. Then, we discuss the challenges in achieving the full potential of heterogeneity – IceBreaker provides solutions for those challenges.

High-end and low-end servers for serverless functions have competing design trade-offs. High-end servers are computationally more powerful and hence, provide faster execution time – thereby, have higher potential to provide a lower service time. However, high-end servers are relatively more expensive. Consequently, using only high-end servers limits the number of servers the service provider can employ under a fixed capital budget. Introducing some low-end servers at the expense of high-end servers allows the service provider to buy additional nodes and higher aggregate memory capacity. This additional memory capacity enables the service provider to provide warm start to more functions. Also, higher node count can potentially reduce the wait time under the same budget constraint – thereby, potentially reduce the service time. Unfortunately, due to lower computational power of low-end severs, functions might be executed at lower speed and hence, undesirably result in longer execution times.

Table 1: A cold start on high-end can have higher service time than a warm start on low-end server (true for functions F_A & F_C, and, in general, true for more than 60% of the functions in ServerlessBench [71], all units are in seconds, CST= cold start time, ET = execution time, ST w/ CS = service time with cold start, ST w/ WS = service time with warm start, the metric checks if a warm start on low-end server is better than a cold start on high-end server).

Function	Server	CST	ET	ST w/ CS	ST w/ WS	Metric
F_A	Low-end	2.63	3.13	5.77	**3.13**	✓
	High-end	2.09	2.75	**4.85**	2.75	
F_B	Low-end	1.20	3.01	4.21	**3.01**	×
	High-end	0.66	0.77	**1.43**	0.77	
F_C	Low-end	1.11	2.09	3.2	**2.09**	✓
	High-end	0.81	1.62	**2.43**	1.62	

Next, we provide experimental evidence based on the production workload trace and representative benchmarks to demonstrate that a mixture of low-end and high-end servers has the potential to provide both lower service time and keep-alive cost, than what a homogeneous setting alone can offer.

Observation 1. Both low-end and high-end server types have the potential to lower service time, but it depends on the function type.

Table 1 shows the cold start time, execution time, and service time with warm start (execution time only) and without warm start (execution time plus cold start time) for three representative applications from the ServerlessBench suite [71] on different types of servers (refer to the Sec. 4 for methodological details). Experimentally, we observed the cold start overhead to be similar for different server types. These selected functions demonstrate the potential benefit of using heterogeneity.

As expected, all functions experience a slow-down when executed on the low-end server. But, interestingly, the overall service time on the low-end server can be lower for some functions if they experience a warm start, compared to a cold start on a high-end server (e.g., F_A & F_C,). This is likely to be true for functions where the cold start overhead is a significant fraction of the execution time – as is the case for many real-world serverless function [57, 71]. The reduction in execution time on the high-end server is not sufficient to outweigh the downside of cold start. Employing low-end servers can increase the number of nodes and hence, the likelihood of warm starts on low-end servers. Despite slower execution time, the overall service time can be smaller for some functions. However, as expected, this is not always true; some functions (F_B) may not experience any benefit despite a warm start on low-end servers.

We point out that this characterization is only focused on service time (primarily a user-centric metric) and does not discuss the impact on keep-alive cost – the expense incurred by the service provider to provide benefit to functions that have the potential to benefit from low-end servers. This leads us to our next discussion point. Via a simple experiment, we demonstrate that a mixture of low-end and high-end servers is beneficial for both end-users and service providers since it has the potential to reduce both service time and keep-alive cost.

Observation 2. A mixture of high-end and low-end servers have the potential to reduce both the service time and keep-alive cost.

To demonstrate this take-away, we pick a function's inter-arrival time from Microsoft Azure production-level serverless function workload trace [57] and estimate its keep-alive cost under various constant keep-alive periods after its execution.

Fig. 2(a) shows the fraction of warm starts as a function of the keep-alive period after the end of execution. For example, if a function is always kept alive (or warm) for 10 minutes after its executions, 35% of its invocation will receive warm starts. As expected, keeping the function alive for a longer period increases the fraction of warm starts (e.g., 15-minute window receives 65% warm start). But, note that the keep-alive cost and execution time change depending on the server type. Low-end servers type naturally incur lower \$/GB-unit-time expense to service providers, and high-end servers provide faster execution time.

Fig. 2(b) show the keep-alive cost and service time for this function alone if only the high-end server was used. In this figure, the keep-alive period is chosen to be 10 minutes – resulting in 35% warm start as expected from Fig. 2 (a). Hence, the average service time calculation adds the overhead of cold start for 65% invocations to the execution time. Since the Microsoft Azure function trace does not provide the function binary – we used application StatelessCost from ServerlessBench [71] to obtain the execution time and cold start overheads (other methodological details about server type and

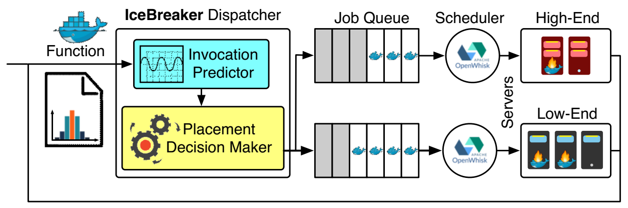

Figure 3: Overview of the design of IceBreaker

keep-alive cost rate covered in Sec. 4). The keep-alive cost is represented in a normalized form as a fraction of the total keep-alive which is the same across all server-type settings.

In Fig. 2(c), we slightly modify the keep-alive cost to demonstrate the potential of using heterogeneity. We keep the function alive for five minutes in the high-end server (14% warm starts as per Fig. 2(a)) and then, move it to a low-end server and keep-alive for 10 minutes (total 65% warm starts). This hand-constructed policy shows that using both types of servers reduces keep-alive cost and execution time. This is because using low-end servers reduces the cost of keeping alive a function per unit time, and hence, the function can be kept alive longer to increase warm starts (lower service time) such that the overall keep-alive cost is also lower. In fact, keeping alive first in the low-end server and then, shifting to a high-end server achieves a similar effect. Motivated by the reduction in both metrics, one may be tempted to employ only low-end servers.

Fig. 2(d) shows that this is not an effective strategy – to achieve the same service time as Fig. 2(c), one will need to keep the function alive for much longer (23 minutes after the execution) such that warm starts are sufficiently increased to a point (84%) to offset the slow-down in execution time. But, unfortunately, the keep-alive cost of keeping this function alive for too long results in 20% additional keep-alive cost compared to the heterogeneous case (Fig. 2(c)).

In summary, our motivational example demonstrates that a heterogeneous solution has the potential to offer improvements in both key metrics, but achieving this in an autonomic fashion for a large mix of functions is challenging – IceBreaker overcomes these challenges. We also point out in this motivating example, our heterogeneous setting has a higher capital budget because it employs both the servers, although it does not use them all the time. It is important to show that a heterogeneous solution can be effective when the capital budget for the infrastructure is fixed and the gains are not coming simply because of an increased capital budget. *Therefore, as we discuss next, IceBreaker is designed to provide effective improvements in both service time and keep-alive cost under fixed overall infrastructure capital budget (our evaluation in Sec. 5 also confirms this).*

3 ICEBREAKER: KEY IDEAS AND DESIGN

We begin by providing an overview of IceBreaker (Fig. 3).

(1) Function Invocation Prediction Scheme (FIP). The FIP helps IceBreaker to predict if a function will be invoked within a particular time interval. In addition, it also predicts the invocation concurrency so that the appropriate number of function instances

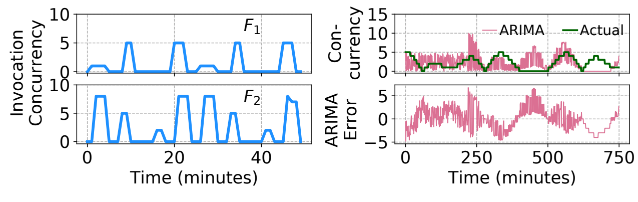

Figure 4: (a) Existing techniques using ARIMA are slow to converge as periodicity changes. (b) The invocation concurrency and inter-arrival time of a function can vary with time.

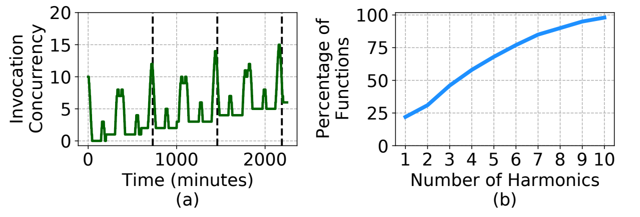

Figure 5: Invocation concurrency of most serverless functions have multiple frequency components or harmonics.

can be warmed up. IceBreaker designs a serverless-computing-specific time series prediction model to make accurate predictions about function invocations.

(2) Placement Decision Maker (PDM). If a function is predicted to be invoked by the FIP, the PDM decides where to warm up a function: the high-end server, or the low-end server, or no warm up at all. It makes this decision based upon a utility score, which is calculated by taking into consideration several factors like the accuracy of the FIP and the speedup gained from warming up on a high-end server.

Next, we delve into these two components in detail.

3.1 IceBreaker: The Function Invocation Predictor

First, we discuss why existing FIP approaches are insufficient.

What is missing from the state-of-the-art function invocation prediction schemes? Almost 98% of the serverless functions (from Microsoft Azure serverless trace) show some kind of periodic behavior in their invocation pattern (number of functions invoked per time interval), as shown in Fig. 4(a). The inter-arrival time between two successive function invocations can keep varying, making probability-histogram-based inter-arrival time prediction schemes (e.g., [21, 43, 49, 57]) unsuitable for serverless. In addition, all existing invocation prediction schemes only predict the inter-arrival time of a function, they do not predict the concurrency with which a function will be invoked [12, 19, 56]. If a function is predicted to be invoked in a time interval, these schemes simply warm up the number of instances of the function which were invoked in the previous time interval. However, as we see from Fig. 4(a), for functions F_1 and F_2, this approach is not suitable. This is because *(1) functions can have multiple invocation concurrency levels, and (2) the concurrency might change between two successive invocations.*

For example, the function shown in Fig. 4(b) changes its invocation concurrency over time. Previous state-of-the-art techniques like "Serverless in the Wild" [57] use auto-regressive integrated moving average (ARIMA) as a FIP. Fig. 4(b) shows that as periodicity changes, the predictions by ARIMA can be greater or lower by a considerable amount compared to the observed invocation concurrency, as reflected in the prediction error of ARIMA. Note that in this example, we enhance ARIMA from [57] to predict the number of invocations and tune its parameters (time lags, degree of difference, and order of moving average) for best results. Even still, because ARIMA uses prediction error and values of previous time intervals to predict an invocation concurrency of the future, it takes a long time to converge. As these kinds of invocation periodicity changes are common with serverless functions [28, 56, 66, 70], there is a need for a better prediction technique.

In IceBreaker, we design a function invocation prediction (FIP) scheme that predicts the concurrency of a function at every time interval, taking advantage of the periodic nature of serverless function invocations.

How does IceBreaker's function invocation prediction (FIP) work? At each time interval, IceBreaker predicts the concurrency with which a function will be invoked in that interval. It is also capable of predicting the inter-arrival time as it is simply the time between two non-zero concurrency predictions.

We observed in Fig. 4(a) that function invocation concurrency can be periodic, with periodicity varying over time. Fig. 5(a) shows that within one major period, the invocation concurrency can have multiple other periodic components (multiple frequencies or harmonics). This invocation concurrency pattern can be considered as a time domain series and hence, it can be described by its *amplitude* (the number of invocations at any time interval) and *period* (frequency with which the amplitude is repeated). The invocation concurrency of the function shown in Fig. 5(a), has three harmonics. The period of the major harmonic (the sinusoidal function with the maximum amplitude) is shown using dashed vertical lines. Fig. 5(b) we see that most of the serverless functions show some kind of periodicity in their invocation concurrency, and frequently they have more than one harmonic (25% of the functions have at least one harmonic, 98% of the functions have less than 10 harmonics). However, to predict the invocation concurrency of the next (future) time interval, we need to model and formalize the invocation pattern. For this purpose, we can take advantage of the Fourier theorem [7]:

Any time series, with multiple frequencies or harmonics, can be broken down into sinusoidal components with different amplitudes and frequencies using the Fourier Transformation.

IceBreaker uses the Fourier Transform to compute the harmonics and the amplitudes in the time domain (invocation concurrency). The time-dependency of invocation concurrency can be represented as a sum of sinusoidal components.

What if the invocation concurrency is non-periodic? When a time series in non-periodic, the Fourier Transform is also effective in breaking it down into multiple sinusoidal components. These components have low values of frequencies (large time period) as the time series is non-periodic. In addition, IceBreaker also needs to capture the overall trend of the invocation currency through polynomial regression. This is helpful for both periodic and non-periodic time series. For example, in Fig. 5(a) the invocation concurrency has an overall increasing slope along with a periodic trend. IceBreaker computes the coefficients (a, b, c) of a second order polynomial function to capture this overall trend of the invocation concurrency $f(t)$. The polynomial regression fit of $f(t)$ is given by $a \times t^2 + b \times t + c$.

We validate the effectiveness of IceBreaker's polynomial model by performing the Pearson χ^2 Goodness-of-Fit Test, which is a widely-used statistical test for polynomial model fit validation [25]. We observed that this polynomial model fits the function invocation trend with 99.2% confidence on average. Thus, we conclude this model is representative of the observed overall trend in invocation concurrency.

Note that $f(t)$ eventually needs to be *de-trended* by subtracting the polynomial regression fit from it in order to only capture the periodic variations of $f(t)$ using Fourier Transform. Hence, using a Fast Fourier Transform (FFT) of the de-trended version, IceBreaker computes the harmonics and their amplitudes as following:

$$FFT(f(t) - a \times t^2 + b \times t + c) = \sum_{i=1}^{n} cos(2\pi f_i t + \theta_i)$$

Here, f_i is the frequency of the i^{th} harmonic of $f(t)$, and θ_i is the phase. Now that we have the mathematical equations of overall trend and the periodic components of the invocation concurrency, IceBreaker uses this to predict the concurrency of a function invocation at a future time interval $t_k + 1$, given that the concurrency up to time interval t_k is known. The following equation gives the value of the concurrency.

$$f(t_k + 1) = a(t_k + 1)^2 + b(t_k + 1) + c + \sum_{i=1}^{n} cos(2\pi f_i(t_k + 1) + \theta_i)$$

IceBreaker considers the top ten major harmonics ($n = 10$) to make this prediction as the major periodic pattern is well captured by the major harmonics, and we observed a negligible difference in prediction performance (<0.75%) beyond ten harmonics. IceBreaker only considers the invocation concurrency in a *local time window* to make the concurrency prediction of the next time interval. This helps IceBreaker to maintain a low overhead of prediction (similar to the pre-existing inter-arrival prediction techniques) throughout all time intervals. IceBreaker selects the local time window to be one hour. However, the variation in IceBreaker's results are negligible (< 2%), when this time window is set to be anywhere below 10 hours. Next, we will discuss how IceBreaker uses this predicted concurrency to decide where to warm up a function.

3.2 IceBreaker: The Placement Decision Maker

IceBreaker's FIP predicts if a function will be invoked in a given time interval. This helps reduce the service time by avoiding cold starts. But to reduce cold starts, a function needs to be warmed up without a large increase in the keep-alive cost. Recall from Sec. 2, that maintaining a heterogeneous mix of low and high-end servers can be beneficial for both the service provider and the users as it reduces the overall keep-alive cost and service time. For the same allocation cost, low-end servers have more memory to warm up more functions than high-end servers because a larger number of low-end servers can be allocated than the number of high-end ones. Warming up more number of functions can potentially reduce cold starts. However, only having low-end servers is harmful for the average service time, while only having high-end servers increases the keep-alive cost. IceBreaker's placement decision maker (PDM) decides where to warm up a function in a heterogeneous system: high-end server, low-end server, no warm-up.

How does IceBreaker's Placement Decision Maker (PDM) make this decision? Because a warm up on a high-end server is costlier than a warm up on a low-end server, PDM gives the most *promising* functions a higher priority for a high-end server warm up.

Not all functions predicted to be invoked by IceBreaker's FIP are actually invoked. Even if they are invoked, the concurrency may not match. If the predicted concurrency of an invocation is more than the actual concurrency, then warming up will increase the keep-alive cost. If the predicted concurrency is less than the actual one, then even after warming up, some of the functions may undergo cold start. Also, there might be cases when warming up only one function can take up a considerable amount of memory which could have been utilized for warming up multiple other functions with lesser memory requirements. By taking these factors into consideration, IceBreaker calculates a utility score per function per time interval. Based on this score the placement decision is made. Next, we will discuss the individual components used by IceBreaker to compute this utility score.

True negative prediction rate. If IceBreaker warms up less number of function instances than the number of functions invoked, then the predicted concurrency by the FIP is less than the actual invocation concurrency. The FIP would have a high true negative rate in this case. Functions observing high true negative rates are prone to observe increased service time due to more cold starts. Hence, IceBreaker increases the utility score of these functions by a factor equal to the true negative rate (T_n), which is given by the ratio of the number of observed cold starts to the total number of invocation of a function within the local time window (*i.e.,* past one hour).

False positive prediction rate. This is the opposite case when IceBreaker warms up more number of instances of a function than the number of actual invocations. These functions increase the keep-alive cost, resulting from high false positive prediction rate. The false positive rate (F_p) is calculated as a ratio of the number of function instances warmed up but not invoked to the total number of invocations of a function within the local time window. IceBreaker assigns a lower utility score for functions with high false positive rates, and vice-versa.

Note, a function with a high false positive rate at a certain time interval may observe a high true negative rate at some other interval, and vice-versa. These rates are direct measurements of the performance of IceBreaker's FIP.

Inter-server speedup. Execution on high-end server affects functions differently. Some might experience a higher speedup than others. IceBreaker favors functions with higher speedups for a warm up on high-end servers. The inter-server speedup (I_s) is calculated as ratio of the sum of execution time and cold start time on a high-end server to their sum on a low-end server (wait time does not affect it).

Memory Footprint. It is costly to warm up functions that consume a large memory. This is because the same amount of memory could have been otherwise used to warm up multiple other functions with lower memory requirements. The memory footprint (M_r) is the memory used to warm up a function as a fraction of maximum allowed by the serverless provider (*e.g.,* 10 GB on AWS Lambda). Functions that consume large memory are given less priority in obtaining a high utility score. However, functions that have a large memory requirement and also experience a high speedup on high-end servers can still achieve a high utility score due to their inter-server speedup. IceBreaker designs safeguards (discussed in Sec. 3.3) to give a fair chance of warm ups to functions that have a large memory footprint but low speedup.

Using the above four components (true negative rate (T_n), false positive rate (F_p), inter-server speedup (I_s), and memory footprint (M_r)), the utility score (S_u) is calculated at every time interval for every function as follows:

$$S_u = \frac{T_n + (1 - F_p) + (1 - I_s) + (1 - M_r)}{4} \tag{1}$$

The components of S_u undergo min-max normalization to ensure that all of them range between 0 and 1.

How are functions warmed up based on their utility score? As shown in Algorithm 1, if S_u is greater than the *high-end cut-off* (H_E) for a function at a time interval, then it is warmed up on a high-end server. If S_u lies between *high-end cut-off* and *low-end cut-off* (L_E), then the function is warmed-up on a low-end server. Else, the function is not warmed up. IceBreaker sets the base values of high-end and low-end cut-offs to be 2/3 and 1/3, respectively. However, these are not hard cut-offs and they change dynamically. If IceBreaker observes that the memory on the high-end servers is vacant, but the low-end servers are getting filled with warm ups, it reduces the high-end cut-off, and vice versa. The cut-offs are changed in proportion to the fraction of vacant memory on the servers.

But, what if the utility scores of the functions are higher than the respective cut-offs, but there remains no memory left to warm them up? Based on the utility score, if such functions are supposed to be warmed up on a high-end server (or a low-end server), IceBreaker searches for vacant memory in a low-end server (or high-end server) to warm them up. Priority is given to the functions with higher utility scores.

How does PDM benefit serverless users? The PDM identifies the most promising functions in terms of their utility score. It identifies functions of which it can improve the service time using warm starts and high-end servers, while reducing the overall keep-alive cost. This directly relates back to the motivation for IceBreaker of having a heterogeneous mix of low and high-end servers, which benefits both serverless users (due to reduction in service time) and service providers (due to reduction in keep-alive cost).

Algorithm 1 IceBreaker's Placement Decision Maker

```
1:  Initialize the high and low-end servers, $f(t) \leftarrow$ Prediction
    concurrency, $S_u \leftarrow$ Utility score
2:  for Every time interval $t$ do
3:      for Every time function do
4:          $I_s \leftarrow$ Inter-server speedup, $M_r \leftarrow$ Memory
5:          $f(t) \leftarrow at^2 + bt + c + \sum_{i=1}^{n} cos(2\pi f_i t + \theta_i)$
6:          if $f(t) > 0$ then
7:              Update $F_p$, $T_n$, $H_E$, $L_E$
8:              $S_u \leftarrow \big[T_n + (1 - F_p) + (1 - I_s) + (1 - M_r)\big]/4$
9:              if $S_u > H_E$ then Warm up on high-end server
10:             else if $S_u < L_E$ then Do not warm up
11:             else Warm up on low-end server
12:     if A function is invoked and server is available then
13:         Execute it with warm or cold start
```

3.3 IceBreaker: Miscellaneous Design Considerations

Here we describe the reason for several design considerations of IceBreaker and how it handles corner cases.

IceBreaker provides safeguards to avoid ping-ponging of functions among the servers for warm up. If the utility score of a function changes in a manner that its warm up keeps alternating between low- and high-end servers, then this is called ping-ponging. This puts unnecessary stress on the memory sub-system, which is shared among all functions executing on a server. To mitigate this, IceBreaker ensures that the warm up location does not change in the next time interval if the utility score of a function in the local time window has not changed by more than 10%. At the end of the local time window, IceBreaker again decides on the server type for the function based on its utility score.

Functions with large memory requirements and low inter-server speedup rate are given chance to have warm starts by IceBreaker Recall from Sec. 3.2 that these kinds of functions may have a lower probability of receiving a high utility score. To mitigate this, IceBreaker ensures that functions which have high memory requirements and only underwent a warm up on a low-end server in the last local time window (if at all), then for the next local window they are always warmed up on a high-end server, even if they should be warmed up on a low-end server as per their utility score.

IceBreaker is scalable by design. IceBreaker avoids having any centralized controller in its design. The FIP and PDM are run per function in a parallel manner. The overhead to perform these steps does not scale up with the number of running functions or the number of low and high-end servers.

What is the life-cycle of a function invocation? If no warmed up instance of the invoked function is found in any of the servers, the function undergoes a cold start, else it has a warm start. If all the servers are fully occupied, the invoked function waits until a spot is available, resulting in some wait time. The combined effect of IceBreaker's FIP and PDM is to reduce the wait time as IceBreaker aims to maximize the number of functions that undergo warm starts and minimize memory wastage of warming up false positives.

3.4 IceBreaker: Implementation

IceBreaker uses Apache OpenWhisk [40], a widely used open-source serverless deployment for creating a serverless platform on reserved servers. IceBreaker uses two types of servers (low- and high-end) with a master node to control function placement. IceBreaker is implemented using a two level design stack: (1) Inter-server dispatcher, (2) OpenWhisk controller.

Inter-server dispatcher level. As shown in Fig. 3, this is the level where the logic of IceBreaker is implemented, and it takes place in the master node. This node has the information of all function characteristics and their corresponding FIP performance. From these, it calculates the utility score and decides where to warm up a function. When a function is invoked, the request arrives at the master node, which then dispatches it to execute on low- or high-end servers. The logic for this level is written in Python3.6, and includes *numpy* and *boto3* as the only external dependencies. It can be readily installed in the master node of a serverless deployment platform to control function execution and warm ups.

OpenWhisk controller level. If the inter-server dispatcher decides to execute or warm up a function on a high-end server, then the OpenWhisk controller decides where among all the high-end servers should the function be scheduled, and same for the low-end servers. This decision is made based on the capacity and utilization of the servers. IceBreaker's design has two OpenWhisk controllers, one for all the high-end servers, another for all the low-end servers. Using OpenWhisk, functions are executed on Docker containers. By default, OpenWhisk keeps these containers warm for 10 minutes after an invocation. However, we modified this to keep functions warm for any period of time as decided by the inter-server dispatcher.

4 EXPERIMENTAL METHODOLOGY

Experimental Platform. A real-system implementation of IceBreaker requires control over function placement, warm up timing, and function-specific keep-alive period. These controls are not exposed to users on commercial platforms (e.g., AWS Lambda, Google Functions and Azure Functions). Therefore, we implemented and evaluated IceBreaker on an OpenWhisk-based setup. OpenWhisk is one of the most widely used serverless workload manager and is adopted by several production serverless providers including IBM Cloud [10].

IceBreaker spawns serverless functions via Docker containers on a heterogeneous mix of servers. This heterogeneous mix is composed of two types of servers: high-end (more expensive and higher performing) and low-end (cheaper and low performing). High-end servers correspond to AWS general purpose m5n instances (usage cost: \$0.01475/GB/hour) and low-end servers correspond to AWS general purpose t4g instances (usage cost:\$0.0084/GB/hour). However, IceBreaker's results are not dependent on the type of instances, and we evaluate it to show similar benefits on other general purpose instances (a1, t3a, and t2) with different cost ratios. We ensure that we do not report results during the burst period of the VMs, by evaluating only after the burst period is exhausted.

For simplicity, in our default evaluation set-up, we divide the total capital budget equally between high-end and low-end servers – this corresponds to ten high-end and eighteen low-end servers. For completeness, in our evaluation, we demonstrate that (1) IceBreaker is effective in the full range of high-end and low-end server configurations (i.e., their relative numbers in the cluster), and (2) IceBreaker benefits are portable to other ratios of their relative cost. Our evaluation includes 11 different configurations, consisting of different numbers of high and low-end servers, including the two extreme cases where all servers are of one type (homogeneous case) – one configuration with 20 high-end servers and another with 35 low-end servers. Experimentally, we observed that the amount of memory consumed and slowdown due to cold start remains similar for high-end and low-end servers. However, IceBreaker benefits are not dependent on them.

Workloads. IceBreaker evaluation is driven by representative serverless applications and workload traces. In particular, we use applications from the ServerlessBench benchmark suite [71]. ServerlessBench consists of benchmarks covering a wide range of real-world serverless applications including image processing, data analytics, online compiling, and linear algebra operations. These benchmarks also cover a wide-variety of serverless-specific characteristics, including the cold-start overhead. To represent production-like workload arrival pattern and characteristics, we use Microsoft Azure Function trace [57]. It consists of serverless invocation data of two weeks on Microsoft production systems. The trace provides the inter-arrival time of functions, the memory allocated for the function and its execution time. IceBreaker follows the same inter-arrival pattern of functions as the Azure trace. From the memory allocated and execution time information of all the functions in the trace, we find the nearest match of a corresponding benchmark from our benchmark pool to represent the corresponding function behavior.

Competing Strategies. We compare IceBreaker with the following state-of-the-art serverless function warm up strategies: (a) **Serverless in the wild (Wild) [57] (ATC' 20):** It employs a hybrid histogram-based approach for predicting the inter-arrival time of different serverless functions, including the ARIMA model for functions with heavy histogram tail; (b) **FaasCache [19] (ASPLOS' 21):** FaasCache uses greedy caching approach to predict and decide which functions to keep-alive; (c) **OpenWhisk [40]:** The native policy in OpenWhisk is to keep-alive the function for 10 minutes after its invocation. But, it does not predict when the next invocation will occur. Similar strategy is followed by other commercial serverless providers like AWS, Google, and Azure functions. *All the performance metrics are reported as a percentage improvement over OpenWhisk.*

We note that the competing techniques were not developed to employ heterogeneity to reduce the keep-alive cost – this is central to IceBreaker only. But, we modified the competing techniques to make them heterogeneity-aware such that it improves their effectiveness. We experimentally observed that the competing techniques achieve their best performance when functions are

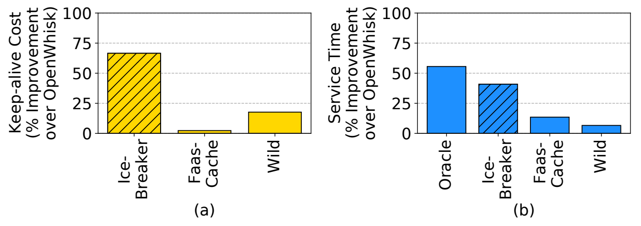

Figure 6: IceBreaker improves overall keep-alive cost (a), and service time (b), significantly more than the competing techniques.

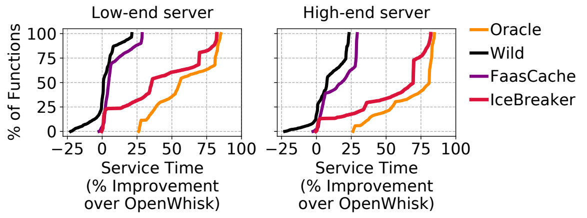

Figure 7: IceBreaker improves the service time on both low-end servers and high-end servers.

prioritized to execute or warm-up on high-end servers. Only when the high-end servers do not have vacancy, it places the function on the low-end servers. This strategy is better than placing functions randomly.

Finally, we compare IceBreaker with the **Oracle** strategy. This strategy has the Oracle knowledge of all future invocations of all functions (i.e., 100% prediction accuracy). It warms up all functions just-in-time before its invocations, and hence, incurs zero keep-alive cost. This results in the minimum service time of a function as all invocations always experience a warm start, if the system has the required unoccupied memory to run the function. While it is not feasible to implement this strategy online on a real system, but this offline strategy is designed to estimate the upper bound on the service time improvement.

Metrics. Two major metrics of interest are the total service time and the keep-alive cost. Total service time is the sum of cold-start time, execution time, and wait time. If the function is invoked while it has a warmed up instance, it incurs zero cold-start time. The execution time depends on the type of server (low or high-end). If the system has enough capacity to execute new functions, the wait time is zero. Keep-alive cost is the total monetary cost that the service provider incurs in keeping the function alive (in memory). Note that the keep-alive cost is dependent on whether the server is low-end or high-end (e.g., keeping a function alive on high-end servers incurs relatively more $/GB/hr expense).

5 EVALUATION AND ANALYSIS

In this section, we analyze IceBreaker and compare its performance with the competing strategies.

Keep-alive cost. *IceBreaker reduces the overall keep-alive cost incurred by the service provider by more than 60% over the baseline, and outperforms existing state-of-the-art approaches by more than 45% points.*

Fig. 6(a) shows the improvement in overall keep-alive cost for different competing schemes on the base heterogeneous configuration compared to the baseline scheme (OpenWhisk's static ten-minute keep-alive policy after a function execution ends; no warm up). First, we observe that prior schemes including Wild and FaasCache provide some improvement over the baseline strategy. This is because, unlike OpenWhisk's static policy, these schemes become adaptive to the behavior of different applications and perform intelligent function warm ups. Second, we observe that IceBreaker significantly outperforms both state-of-the-art schemes (Wild and FaasCache) and provides 66% improvement over the baseline. There are two key contributing factors. First, IceBreaker is able to reduce the keep-alive cost because IceBreaker attempts to make the keep-alive cost for each function invocation proportional to its utility score (e.g., warming and keeping alive functions with lower utility scores on the low-end servers). Second, IceBreaker's function invocation prediction scheme is more effective at predicting the next invocation. Hence, IceBreaker reduces the number of warm ups of functions which are not invoked, compared to the competing strategies. This further reduces the keep-alive cost.

Service time. *IceBreaker reduces the average service time of functions by over 40% over the baseline and outperforms all existing state-of-the-art approaches.*

Fig. 6(b) show the average service time normalized to OpenWhisk's baseline policy. IceBreaker outperforms existing state-of-the-art techniques and performs similar to the Oracle scheme. Recall that the Oracle scheme is practically infeasible since it requires future knowledge that is not available to IceBreaker. Despite that, the gap between IceBreaker and Oracle is small (54% vs 40% improvement over the baseline). The next best technique provides only 13% improvement in service time. This improvement in service time is because IceBreaker's function invocation prediction scheme is better at avoiding cold starts than the competing techniques. The success of IceBreaker's utility score based scheduling is dependent on function properties (e.g., memory footprint and inter-server speedup) being consistent across invocations. This holds true in practice – on an average, memory footprint of a function changes by only 0.77% and inter-server speedup by only 1.1% across all invocations.

Tail and median latency improvements. IceBreaker's benefits are not only limited to the mean values of the metrics. IceBreaker provides 68% and 42% improvement over baseline in median latency (versus 20% and 18% improvement by the next best technique, FaasCache) for keep-alive cost and service time, respectively. *IceBreaker also provides 53% and 36% improvement in tail latency (95th percentile), versus 17% and 19% improvement by the next-best technique (FaasCache) for keep-alive cost and service time, respectively.*

Why does IceBreaker outperform existing strategies?

To understand the reason behind IceBreaker's performance improvement over the competing techniques, let us look into the distribution of service time in the heterogeneous mix of servers

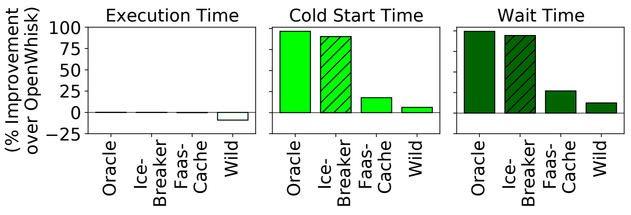

Figure 8: IceBreaker primarily improves the cold start time and wait time components of the service time.

(Fig. 7). We observe that IceBreaker consistently provides reduction in service time to almost all functions (>98%) compared to the baseline and its service time improvement CDF curve is closer to the Oracle scheme. In comparison, other competing techniques (Wild and FaasCache) result in service time degradation for more than 25% functions and magnitude of improvement even for the most-benefiting functions are lower compared to IceBreaker.

To dig deeper, we analyze the service time distribution of functions individually in the high and low-end servers of the heterogeneous mix. From Fig. 7, we observe that the improvement in service time is higher for the high-end servers than the low-end servers (through the maximum possible improvement is similar in both the server types). This is because IceBreaker's function placement is guided by the utility score – which aims to place the most promising functions on the high-end servers. Functions that have higher chances of invocation and achieve most benefit from warm ups on high-end servers are considered to be the most promising functions. These functions contribute the most toward improving the average service time, and help IceBreaker to reduce the overall keep-alive cost. However, IceBreaker also consistently improves the service time for most of the functions (> 98%) in both type of clusters.

Next, we analyze the reason for IceBreaker's effectiveness toward reducing the service time. Recall that the service time is made up of three components – execution time, cold start time, and wait time. The execution time depends on the type of cluster where a function is running, and since all the competing techniques and IceBreaker uses both the high and low-end servers, there is only minor difference in execution time among all the techniques (Fig. 8). However, IceBreaker improves the net service time over the competing techniques as it achieves a higher improvement in cold start time and wait time. The improvement in cold start time is observed because IceBreaker's invocation prediction mechanism correctly identifies the functions to be invoked. This helps to avoid cold starts and hence, IceBreaker achieves a performance almost similar to Oracle (only 6% difference). Wait time is improved because IceBreaker's function warm up is based on utility score that selects functions that are more likely to be invoked.

To understand the reason behind the keep-alive cost reduction, we look at two key metrics: (1) *Successful warm-up cost*: It is the cost of warming up a function that does get invoked. (2) *Wasteful warm-up cost*: It is the cost of the memory wasted when a function is warmed up but not invoked.

From Fig. 9(a), we observe that IceBreaker reduces the cost of successful warm-up by more than 10% on both low- and high-end servers compared to the competing techniques. This is because

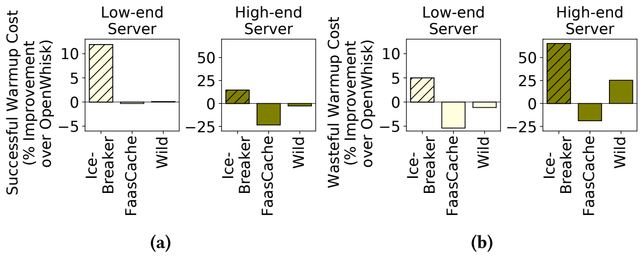

Figure 9: IceBreaker reduces the cost of successful (a), and wasteful (b), warm-ups on both high- and low-end servers.

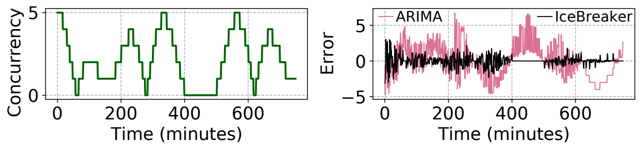

Figure 10: IceBreaker's prediction works better than ARIMA.

IceBreaker's function invocation prediction scheme is better than the competing techniques (improves service time by more than 40% over the baseline v/s only 13% improvement with the next best technique). From Fig. 9(b), we see that IceBreaker reduces the cost of wasteful warm-up on both types of servers. This result also supports that IceBreaker's invocation prediction scheme is better than competing techniques in detecting functions that are not invoked. For high-end servers, the improvement in wasteful warm-up cost is more than 65% over OpenWhisk. This large improvement is because IceBreaker's utility score ensures to place functions with the highest probability of invocation on high-end servers.

This reduction in keep-alive cost helps IceBreaker reduce the memory wastage due to wasteful warm up. On high-end servers, IceBreaker reduces more than 20% memory wastage compared to the next best technique (Wild). This free memory helps IceBreaker to quickly allocate and execute other functions and reduce their wait time. This memory can also be used by the cloud provider to allocate more functions in the cluster.

In addition, recall that in IceBreaker's design (Sec. 3) we observed that existing techniques for invocation concurrency prediction using ARIMA (Wild [57]) has significant error when a function's invocation periodicity changes. Using the same example as Fig. 4, we confirm that IceBreaker is more effective at prediction, achieving lower prediction error, and faster convergence (Fig. 10).

Overhead. *IceBreaker incurs minimal overhead. This increases its suitability for practical deployment.*

IceBreaker has two sources of overhead at the start of every time interval (configured to be one minute in our setup): (1) prediction of whether the function should be warmed up, and (2) where it should be warmed up. The second decision may involve moving a function between two different types of servers. In our experiments, we measured that IceBreaker incurs an overhead of less than 30ms for the first part. This overhead is on the critical path only if the

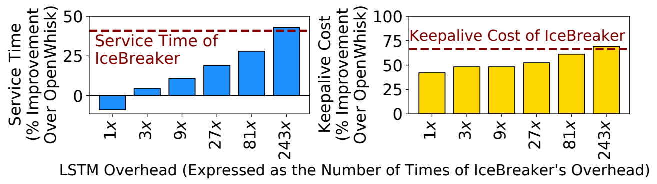

Figure 11: Complex learning-based prediction mechanism provide marginal improvement, but incur prohibitive overhead.

function is invoked in the first 30ms of the one-minute-long time interval (1000 ms), otherwise its latency can be hidden. However, to provide a more conservative estimate, all our results include this overhead and treat this on the critical path even if the function is invoked after 30ms during the same minute. Competing schemes (Wild and FaasCache) have overhead in the same order (10-20ms), but provide lower benefit.

The second overhead is similar to cold-start and occurs when the function is invoked before IceBreaker's prediction mechanism can warm up a function. This overhead component is also accounted for in our evaluation. Our evaluation trace consists of 1.2% of functions that execute for less than 100 milliseconds on high-end servers. Even these extremely short-running functions observe more than 22% reduction in average service time after accounting for all overheads pessimistically – confirming that IceBreaker's overhead is practical even for millisecond-range applications.

While it is possible to achieve higher reduction in overall keep-alive cost and service time by using more complex deep learning techniques, they have very high overhead in the context of serverless platforms. For example, Fig. 11 shows that replacing IceBreaker's relative simple prediction mechanism with a complex long short term memory (LSTM) model yields higher benefits. However, the overheads are higher (243×), but for only marginal improvement over the benefits IceBreaker already provides.

Effectiveness on different cluster compositions and cost ratios. *IceBreaker is effective across different compositions of the cluster and different cost ratios of high-end to low-end servers.*

Earlier, we had demonstrated that IceBreaker provides service time and keep-alive cost improvements for our base heterogeneous configuration which uses the low-end and high-end servers such that the aggregate cost of each type of servers is equal. To further evaluate IceBreaker's robustness, we performed a complete sweep of configurations – that is, we experiment with different combination of servers from only high-end servers (leftmost bars in Fig. 12) to only low-end servers (rightmost bars in Fig. 12) such that the total expense of building the heterogeneous mix of servers remains equal to the capital cost of high-end homogeneous servers. All configurations in between the two homogeneous ends correspond to a heterogeneous configuration. We have purposely kept the capital cost of building the server configurations the same during this sweep to demonstrate that IceBreaker's observed benefit in reducing the keep-alive cost is not a side-effect of changing the total capital cost of the server itself.

Fig. 12 shows that IceBreaker is effective across all compositions of low-end and high-end servers. In particular, this result shows that IceBreaker consistently outperforms all existing schemes and provides close-to-optimal service time improvements. The fact that IceBreaker provides benefit over competing schemes even in the two homogeneous ends reflects the fact that IceBreaker's function invocation prediction scheme is better at detecting function invocations. IceBreaker also outperforms all existing schemes and provides close-to-optimal service time improvements in terms keep-alive cost.

The only exception is observed on high-end homogeneous servers. This is because the high-end homogeneous configuration has lower number of servers and lower amount of memory to warm up functions than other configurations. When the memory is lower, it is harder to achieve more warm starts. IceBreaker's invocation prediction scheme works better than the competing techniques and it achieves a higher improvement in service time (53% on homogeneous high-end v/s 32% on homogeneous low-end servers) on the high-end servers. However, IceBreaker has to pay the price for this large improvement in service time by increasing the keep-alive cost by 25% on homogeneous high-end servers. Overall, we note that availing IceBreaker's benefits does not require determining the optimal ratio of low-end to high-end servers. While targeting optimal capacity planning of low and high-end servers, we learned that keeping the heterogeneity ratio similar to the cost ratio is quite effective; although not always optimal, it provides a good first-order estimate for planning.

Similar to cluster composition, we perform sensitivity analysis w.r.t. cost ratio of high-end to low-end servers. Recall that in our default setup, it was 1.8× (that is, the high-end server is 1.8× more expensive than a low-end server). This ratio was not actively chosen by us, but instead determined via various constraints (e.g., number of nodes in the real-system cluster, budget for the experiment, available nodes instances on AWS, etc.). Therefore, to test robustness, during our sensitivity analysis, we varied this ratio from 1.3× to 2.4×.

Our results (Fig. 13) confirmed that IceBreaker continues to provide significant benefit at other ratios too. For a 1.28× ratio between m5a and t4g instances, IceBreaker achieves 22% improvement in keep-alive cost and 18% improvement in service time over the baseline (the next best technique, FaasCache, provides 11% and 8% point improvement in keep-alive cost and service time, respectively). IceBreaker also outperforms competing techniques at lower ratios. For a cost ratio of 1.23× between t3 (high-end) and t4g (low-end), IceBreaker achieves 17% improvement in keep-alive cost and 14% improvement in service time over the baseline (FaasCache provides 1% and 6% point improvement in keep-alive cost and service time, respectively). For a 1.5× cost ratio, IceBreaker achieves 33% and 21% points-improvement in keep-alive cost and service time, respectively over the next best technique, FaasCache. As expected, when the ratio is closer to one (i.e., homogeneous cluster), the improvements come only from the invocation prediction scheme.

Hard-to-predict and infrequent functions. *IceBreaker is effective for both hard-to-predict and infrequent function types.*

Previous works have shown that a sizeable fraction of functions invocation are infrequent (e.g., occurring only once a day) and have

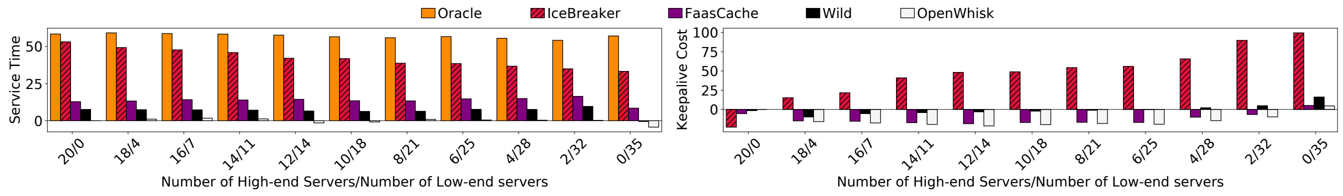

Figure 12: IceBreaker is effective across different compositions of the servers (% improvement over OpenWhisk at 20/0).

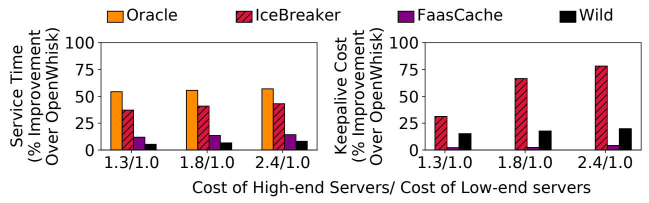

Figure 13: IceBreaker is effective across different cost ratios of high-end to low-end servers.

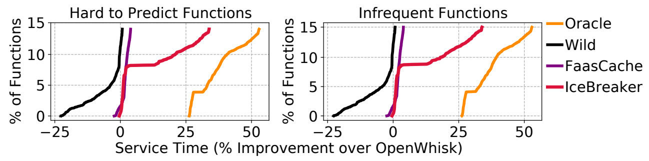

Figure 14: IceBreaker reduces service time and keep-alive cost for both hard-to-predict and infrequent functions.

difficult to detect re-invocation patterns (e.g., missed by time-series based ARIMA predictions [57, 65]). These works have noted the importance of improving the quality of prediction and reducing the keep-alive cost for such functions. Hence, we evaluated IceBreaker's effectiveness for both hard-to-predict and infrequent functions.

Hard-to-predict functions are the ones whose invocations receive the least successful warm-ups and hence, incur the relatively highest cold-start times. For demonstration, 15% of all the functions which have the highest average cold-start time are treated as hard-to-predict functions. Similarly, infrequent functions are the 15% of all the functions which are invoked least frequently. Fig. 14 shows that IceBreaker provides service time improvement and reduces the keep-alive cost for such functions too. On average, existing schemes yield better service time compared to OpenWhisk's baseline, but they are not always effective for hard-to-predict and infrequent functions as observed in the left tail in Fig. 14. IceBreaker improves this situation due to its better prediction scheme, resulting in lesser number of cold starts for such functions. Also, by warming up such functions judiciously on low-end servers, IceBreaker reduces the overall keep-alive cost. Among all schemes, IceBreaker is the closet to the Oracle strategy.

One might think that what prevents IceBreaker from experiencing oscillations by constantly scheduling incorrectly between low-end and high-end servers, especially for hard-to-predict functions with invocation spikes? The answer to that is IceBreaker makes scheduling decisions based on its utility score, which considers inter-server speedup and memory footprint. These components prevent the utility score from changing drastically for even hard-to-predict functions. Among the hard-to-predict functions, the utility score changes by more than 10% in the local time window only 13% of the times (mean value of utility score change for hard-to-predict functions is only 6.3%).

We clarify that IceBreaker's scope is not limited to functions that are hard-to-predict. IceBreaker is even more effective for functions that are invoked frequently (e.g., 51% and 33% average points improvement in keep-alive cost and service time, respectively over the next best technique, FaasCache, for the top 15% most frequent functions). IceBreaker also outperforms competing techniques under most challenging scenarios – functions with unexpected invocation concurrency spikes. For example, for the top 15% of the functions with the highest spikes in invocation concurrency, IceBreaker has 18% and 11% points improvement in keep-alive cost and service time, respectively over the next best technique, FaasCache, compared to baseline.

Effect of heterogeneity on economy of scale and hardware resource fungibility.

By design, IceBreaker relies on heterogeneity in server nodes to reduce the keep-alive cost and service time. However, we note this is not a new constraint, rather seizing an existing opportunity. With prevalent heterogeneity in processors, data centers often already have heterogeneous and multi-generational computer hardware [8, 14, 64] – IceBreaker demonstrates how to leverage this existing opportunity. Besides reducing keep-alive cost, IceBreaker also reduces the average service time. Even a small reduction in service time allows businesses to generate significant revenue – potentially compensating the capital and operating expense of heterogeneous systems and potential reduction in hardware resource fungibility [26, 39, 46]. IceBreaker opens up a new research avenue of heterogeneous serverless computing. For example, determining optimal capacity planning to maximize IceBreaker's benefits.

6 RELATED WORKS

In addition to the works that IceBreaker is evaluated against, there are other works that characterize serverless platforms.

Workload characterization. There have been studies that have captured the behavioral trends of commercial serverless platforms, both from the user side by running representative benchmarks [24, 41, 45, 65, 71], and also from the serverless provider side by examining workload traces [56, 57, 66, 68]. These studies have captured several trends of serverless computing, like scheduling effects, invocation patterns, and I/O patterns [20, 36–38, 44, 48]. Even local deployments of serverless results in service time degradation due to cold starts [9, 42]. Overall, prior works confirm the severity of cold start time and keep-alive cost [16, 24, 50, 59, 62].

Cold start and keep-alive cost mitigation. Multiple works have attempted to address the cold start challenges in various ways [2, 12, 19, 23, 33, 48, 58–61]. For example, Mohan et al. [50] propose pre-allocating network interfaces and bind them to function containers to start them faster. When an application consists of multiple functions, SAND [3] uses sand-boxing techniques to warm up subsequent invocations of a function within the application. SOCK [51] caches commonly used Python libraries, and Kesidis et al. [34] overbook function instances to ensure warm starts. Previous works have demonstrated some similarities and distinguishing features of why cold start overhead cannot be appropriately mitigated by caching solutions alone – primarily due to the keep-alive cost.

Unlike conventional caching, serverless computing's charge model is based on resource allocated per unit time [22, 35, 47, 53]. Thus, cold start mitigation strategies should prioritize unloading functions from memory when they are less likely to be invoked, rather than unloading them only when other functions need space. Hence, as demonstrated by other approaches on managing variable sized caches, Time-to-Live caches cannot be applied [4–6, 52, 69]. Also, traditional caching algorithms depends on the hit and miss ratios of all objects [13, 17], whereas such a centralized control will not scale well for serverless functions. Nevertheless, we quantitatively compare with the most recent work in this area [19] which is primarily based on caching principles but with additional improvements for the serverless-specific case. Finally, prior approaches attempting to mitigate the cold start overhead, tend to either increase the capital cost, or increase the keep-alive cost [1, 15, 20, 29, 30, 54, 63, 67]. In contrast, IceBreaker tries to address cold starts by reducing keep-alive cost, while maintaining the same capital expenditure.

7 CONCLUSION

IceBreaker is a novel mechanism that reduces the service time and keep-alive cost of serverless functions by accurately predicting the invocation concurrency and arrival time in a heterogeneous server system. To the best of our knowledge, this is the first work to introduce heterogeneous servers for the execution of serverless functions to improve both performance and cost-effectiveness. IceBreaker reduces keep-alive cost by 45% and service time by 27% over the state-of-the-art approaches. IceBreaker can be deployed in any cluster with a serverless workload manager and is beneficial for both end-users and service providers. IceBreaker is open-sourced for community adoption: https://zenodo.org/record/5748667.

ACKNOWLEDGMENTS

We thank David Meisner (our shepherd) and anonymous reviewers for their valuable feedback. This work is supported by NSF Award 1910601, 1920020, and 2124897, and Northeastern University.

A ARTIFACT APPENDIX

A.1 Abstract

IceBreaker is technique that reduces the service time and keep-alive cost of serverless functions, which are executed on a heterogeneous system consisting of costly and cheaper nodes. IceBreaker's design consists of two major components: (1) Function Invocation Prediction Scheme (FIP), and (2) Placement Decision Maker (PDM). The FIP uses a Fourier transform based approach to determine the invocation concurrency of a function. The PDM decides where to warm up a serverless function: on a high-end server, or on a low-end server, or no warm up at all. This decision is made based upon an utility score which is calculated by considering several factors like probability of function invocation, speedup of a function on a high-end server, etc. Our artifact packages the scripts for setting up and invoking IceBreaker. It also contains the data obtained in our experimentation. The artifact is available at the following link:

https://zenodo.org/record/5748667

It includes the following:

- The framework to set up IceBreaker on AWS EC2 virtual machines (VMs).
- Scripts to set up the invocation scheme of IceBreaker which is ran from a local environment.
- Scripts and instructions to set up the benchmarks and serverless invocation trace.
- Keep-alive cost data of executing IceBreaker and other competing techniques.
- Service time data of executing IceBreaker and other competing techniques
- Data of executing IceBreaker and other competing techniques for different number of combinations of high-end and low-end servers.

Multiple runs are performed for each of the experiments. IceBreaker is evaluated based on the Microsoft Azure function invocation trace. IceBreaker can be set up for all or any individual function from the trace.

A.2 Artifact check-list (meta-information)

- **Algorithm:** A Fourier transform based FIP. PDM decides the location of a function warm up.
- **Program:** Benchmarks from SeverlessBench were used to evaluate IceBreaker.
- **Data set:** The data set consists of Microsoft Azure serverless function trace. This trace provides 2 weeks of serverless invocation data. From this trace, per function invocation concurrency is obtained, per time interval.
- **Run-time environment:** Python3.6 with *boto3*, *paramiko* and *awscli*. AWS is used for spawning serverless functions and setting up EC2 VMs from a Ubuntu 18.04.4 LTS server. EC2 VMs are installed with *docker* for the operation of IceBreaker.
- **Hardware:** AWS EC2 VMs with different node counts, ranging upto 35.
- **Metrics:** Service time, and keep-alive cost are the major metrics of evaluation. Along with these, the prediction accuracy of FIP is important for the operation of IceBreaker.
- **Output:** Service time and keep-alive cost.
- **Experiments:** The experiments measure the keep-alive cost and service time per serverless function, per time interval. In our evaluation, we have experimented with the entire sweep of server configurations – from 35 low-end servers and 0 high-end servers, to 20 high-end servers and 0 low-end servers. The configurations are chosen to maintain an equal capital cost across all of them.
- **How much disk space required (approximately)?:** 7 GB

- **Publicly available?:** Yes
- **Archived (DOI)?:** https://zenodo.org/record/5748667

A.3 Description

This artifact provides the framework of IceBreaker, which proposes the idea of having a heterogeneous cluster for intelligently warming up serverless functions. The goal of IceBreaker is to jointly minimize service time and keep-alive cost by avoiding cold starts. As an input, IceBreaker requires the benchmarks and the invocation trace which will be used to spawn the benchmarks. The user also needs to specify the family of the EC2 VMs to use as high-end and low-end servers. Different parameters related to IceBreaker, like the number of harmonics to consider for the Fourier transform of the FIP, how much ping-ponging of functions can be allowed, etc. are other parameters that can be modified in the implementation of IceBreaker.

A.4 Methodology

IceBreaker is implemented in Python3.6 and it can be easily portable to use with multiple cloud providers. As a dependency, it requires the command line interface (CLI) of the respective cloud provider to be installed and configured with the user account credentials. According to the user's chosen high-end and low-end VM families, IceBreaker sets up EC2 servers to form a heterogeneous cluster. Then IceBreaker spawns the serverless benchmarks on those VMs based on an invocation trace of serverless functions. These serverless functions run as docker containers. The FIP of IceBreaker predicts which of the serverless functions to warm up, along with the concurrency. Based on FIP's prediction accuracy and several factors like relative speedup of functions on high-end servers, decision is made by the PDM on where (high-end or low-end) to warm up a serverless function. When a function is warmed up, its docker image is already fetched and a container is already created, ready for execution. If a function undergoes cold start, the whole process of fetching the docker image and creating a container from that image occurs, resulting in an increased function start-up time.

A.5 Installation

To set up the trace, benchmarks, and the framework of IceBreaker, *boto3* must be installed and *awscli* should be configured with the user's AWS account credentials. This will allow the user's local environment to directly communicate with AWS. This will aid in the process of setting up EC2 VMS, and running serverless functions on them. Before running IceBreaker, *docker* must be installed and configured as the serverless functions run as docker containers on the EC2 VMs. The installation process consists of setting up the user's local environment, and the AWS EC2 VMs. More details on installation is provided in the *README.txt* file in the artifact.

REFERENCES

[1] 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.

[2] Siddharth Agarwal, Maria A Rodriguez, and Rajkumar Buyya. 2021. A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 797–803.

[3] 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. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*. 923–935.

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

[5] Abdullah Balamash and Marwan Krunz. 2004. An overview of web caching replacement algorithms. *IEEE Communications Surveys & Tutorials* 6, 2 (2004), 44–56.

[6] 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.

[7] René Johan Beerends, Henricus G ter Morsche, JC Van den Berg, and EM Van de Vrie. 2003. *Fourier and Laplace transforms.*

[8] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. 2018. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 316–331.

[9] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the" micro" back in microservice. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 645–650.

[10] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2658–2659.

[11] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (2019), 44–54.

[12] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.

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

[14] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 621–637.

[15] 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.

[16] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (2020), 32–39.

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

[18] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 475–488.

[19] 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*. 386–400.

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[21] Alim Ul Gias and Giuliano Casale. 2020. COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.

[22] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. 2019. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 199–208.

[23] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*. 280–295.

[24] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidamabaram Nachiappan, Ram Srivatsa Kannan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Characterizing Bottlenecks in Scheduling Microservices on Serverless Platforms. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1197–1198.

[25] MG Habib, DR Thomas, et al. 1986. Chi-Square Goodness-if-Fit Tests for Randomly Censored Data. *The Annals of Statistics* 14, 2 (1986), 759–765.

[26] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018.

Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[27] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[28] Sanghyun Hong, Abhinav Srivastava, William Shambrook, and Tudor Dumitraș. 2018. Go serverless: Securing cloud via serverless design patterns. In *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[29] MohammadReza HoseinyFarahabady, Javid Taheri, Zahir Tari, and Albert Y Zomaya. 2017. A dynamic resource controller for a lambda architecture. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 332–341.

[30] M Reza HoseinyFarahabady, Albert Y Zomaya, and Zahir Tari. 2017. A model predictive controller for managing QoS enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Transactions on Parallel and Distributed Systems* 29, 7 (2017), 1442–1455.

[31] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.

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

[33] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2019. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*. 158–164.

[34] George Kesidis. 2019. Overbooking Microservices in the Cloud. *arXiv preprint arXiv:1901.09842* (2019).

[35] Young Ki Kim, M Reza HoseinyFarahabady, Young Choon Lee, and Albert Y Zomaya. 2020. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems* 31, 10 (2020), 2289–2301.

[36] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 759–773.

[37] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 789–794.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 427–444.

[39] Ashwin Kumar Kulkarni and B Annappa. 2019. Context aware VM placement optimization technique for heterogeneous IaaS cloud. *IEEE access* 7 (2019), 89702–89713.

[40] Aleksandr Kuntsevich, Pezhman Nasirifard, and Hans-Arno Jacobsen. 2018. A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In *Proceedings of the 19th International Middleware Conference (Posters)*. 3–4.

[41] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 442–450.

[42] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. 2020. Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 399–408.

[43] Changyuan Lin and Hamzeh Khazaei. 2020. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 615–632.

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

[45] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.

[46] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-performance heterogeneity-aware asynchronous decentralized training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 401–416.

[47] Kunal Mahajan, Daniel Figueiredo, Vishal Misra, and Dan Rubenstein. 2019. Optimal pricing for serverless computing. In *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–6.

[48] Nima Mahmoudi and Hamzeh Khazaei. 2020. Performance Modeling of Serverless Computing Platforms. *IEEE Transactions on Cloud Computing* (2020).

[49] Nima Mahmoudi and Hamzeh Khazaei. 2020. Temporal Performance Modelling of Serverless Computing Platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 1–6.

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

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

[52] Stefan Podlipnig and Laszlo Böszörmenyi. 2003. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)* 35, 4 (2003), 374–398.

[53] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 193–206.

[54] Aakanksha Saha and Sonika Jindal. 2018. EMARS: efficient management and allocation of resources in serverless. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 827–830.

[55] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. 2021. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM* 64, 5 (2021), 76–84.

[56] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1063–1075.

[57] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 205–218.

[58] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 419–433.

[59] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference*. 1–13.

[60] Khondokar Solaiman and Muhammad Abdullah Adnan. 2020. WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 144–153.

[61] Amoghvarsha Suresh and Anshul Gandhi. 2019. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*. 19–24.

[62] Davide Taibi, Josef Spillner, and Konrad Wawruch. 2020. Serverless computing-where are we now, and where are we heading? *IEEE Software* 38, 1 (2020), 25–31.

[63] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 311–327.

[64] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[65] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[66] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FAASNET: Scalable and Fast Provisioning of Custom Serverless ContainerRuntimes at Alibaba Cloud Function Compute. *arXiv preprint arXiv:2105.11229* (2021).

[67] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[68] 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 ({USENIX}{ATC} 18)*. 133–146.

[69] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 996–1008.

[70] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. 2019. Adaptive function launching acceleration in serverless computing platforms. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 9–16.

[71] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 30–44.