ProPack: Executing Concurrent Serverless Functions Faster and Cheaper

Rohan Basu Roy
Northeastern University

Tirthak Patel
Rice University

Richmond Liew
Northeastern University

Yadu Nand Babuji
Argonne National Laboratory

Ryan Chard
Argonne National Laboratory

Devesh Tiwari
Northeastern University

ABSTRACT

The serverless computing model has been on the rise in recent years due to a lower barrier to entry and elastic scalability. However, our experimental evidence suggests that multiple serverless computing platforms suffer from serious performance inefficiencies when a high number of concurrent function instances are invoked, which is a desirable capability for parallel applications. To mitigate this challenge, this paper introduces ProPack, a novel solution that provides higher performance and yields cost savings for end users running applications with high concurrency. ProPack leverages insights obtained from experimental study to build a simple and effective analytical model that mitigates the scalability bottleneck. Our evaluation on multiple serverless platforms including AWS Lambda and Google confirms that ProPack can improve average performance by 85% and save cost by 66%. ProPack provides significant improvement (over 50%) over the state-of-the-art serverless workload manager such as Pywren, and is also, effective at mitigating the concurrency bottleneck for FuncX, a recent on-premise serverless execution platform for parallel applications.

CCS CONCEPTS

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

KEYWORDS

Serverless Computing, Cloud Computing, Scalability

ACM Reference Format:
Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. 2023. ProPack: Executing Concurrent Serverless Functions Faster and Cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3588195.3592988

1 INTRODUCTION

Background. Serverless computing paradigm is becoming increasingly prevalent in multiple domains including embarrassingly data-parallel scientific computation, machine learning, and parallel video processing [8, 30, 35, 46, 49, 54]. Serverless computing has lowered the barrier to entry for end users by enabling them to make use of cloud computing resources in a pay-as-you-go model, without worrying about the management and provisioning of computing resources. Another attractive aspect of the serverless computing model is its ability to scale the computing resources elastically and quickly. An application can spawn multiple "concurrent" function instances in parallel, which is useful for data-parallel applications (e.g., sorting). Serverless computing is particularly attractive for end users in the high throughput computing (HTC) and HPC domains as it provides *on-demand HPC* like capability with elastic scalability, without the overhead of provisioning/managing the cluster resources. Consequently, serverless computing for parallel applications is becoming an emerging area of interest [59, 71, 90]. For example, even on-premise serverless execution platforms for parallel applications are being developed (e.g., FuncX [11]) and scientific computing-focused serverless workload managers such as Pywren [34] have been developed for high performance data analytics and visualization applications, such as distributed sorting using map reduce [59].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPDC '23, June 16–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592988

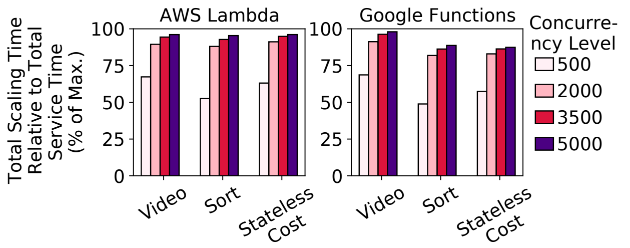

Figure 1: Scaling time can account for a significant fraction of the total service time across different commercial serverless providers, especially at high concurrency levels.

Serving multiple concurrent function instances is also a critical requirement for other types of parallel applications [48]. High concurrency helps embarrassingly parallel serverless applications to achieve high throughput [28]. Resource-intensive large-scale applications are frequently broken down into multiple steps, where each of the steps is processed in parallel by a large number of serverless functions [14]. High concurrency allows us to achieve shorter execution time for compute-intensive parallel processing tasks.

Problem Motivation and Why Existing Solutions Are Not Sufficient. Serverless computing is designed to offer elastic scalability – a desirable feature for parallel computing applications. However, in practice, as our experiments reveal, scaling a serverless application to spawn a large number of concurrent function

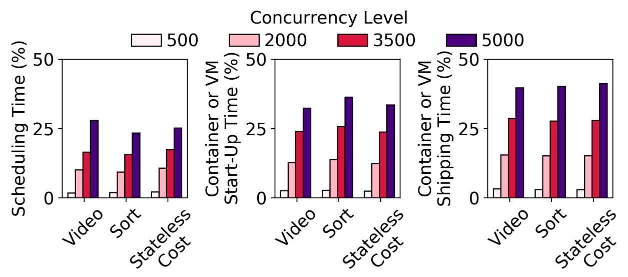

Figure 2: Scheduling Time, start-up time, and shipping time make up scaling time and all of them increase with concurrency (expressed as a % of scaling time at a concurrency of 5000).

instances has significant performance inefficiencies on various serverless cloud computing platforms. Essentially, when an application requests to spawn multiple function instances at the same time, the starting time of different function instances can vary significantly [10, 81, 87]. Note that each instance executes the same function code with different inputs inside a container. The function instance that starts the latest may get delayed by a significant amount. The time between the start of the first function instance and the start of the last instance (referred to as *scaling time*) can be prohibitively large and becomes worse with concurrency.

To provide quantitative evidence, we present results from three widely-used serverless benchmarks as an example: Thousand Island Scanner (Video) [60], Map Reduce Sort (Sort) [58], and Stateless Cost [87]). We note that these applications perform parallel data analytics and information processing tasks, which are some of the frequently used components of HPC workflows [18, 25, 76, 85]. For brevity, we are presenting results for these three applications for motivation (details in Sec. 3, ProPack is also demonstrated to be effective on parallel bioinformatics HPC applications, such as Smith-Waterman, which performs dynamic computation for comparing protein sequences [7, 19, 67, 68].

Fig. 1 shows that the scaling time can constitute more than 80% of the total service time of the applications on the Amazon Lambda platform (*total service time* refers to the time between the start of the first instance and the end of the application execution). We performed additional experiments on other serverless computing platforms and found that these scalability bottlenecks are not limited to the Amazon Lambda platform; applications also suffer from these performance inefficiencies on other platforms such as Google Cloud and Microsoft Azure. As our discussion in Sec. 2 shows, the internal scheduling algorithm is effective at providing performance isolation among different function instances. Thus, this scalability bottleneck is not a simple side-effect of the co-location of multiple function instances. Instead, spawning a large number of such function instances is not instantaneous and is being potentially affected by internal scheduling delays and containerization time or VM start-up time influences at the service provider side [4, 35, 81]. This scaling time (the time between the start of the first instance and the start of the last function instance plus the provisioning delay of the first instance) is a major scalability bottleneck that increases the total service time of serverless applications.

To understand the root cause of the scalability bottleneck, we need to understand what happens behind the scenes during the invocation of a serverless function. A serverless function is stored as an image containing the function source code, runtime environment, and information about its dependencies in a server of the cloud provider. Upon function invocation, a scheduling algorithm searches among the running servers of the datacenter to execute the function [78]. The time required to do this is called the *Scheduling Time*. The scheduling time increases with the invocation concurrency, as the scheduling algorithm needs to search and find more places for the execution of serverless functions. In the next step, the server containing the function image forms containers (or microVMs as in AWS Lambda) by downloading and installing the runtime environment and the dependencies, as specified in its image [33]. This step is bounded by the network bandwidth and the computing capacity of the server. Hence, with concurrency, as more containers are created, the time required to perform this step, or the *Container or VM Start-Up Time*, increases.

In the final step, the formed containers (or microVMs) are shipped to different servers of the datacenter as decided by the scheduling algorithm [84]. This step is again bounded by the network bandwidth of the server forming the containers and hence the time required to do this step (*Container or VM Shipping Time*) increases with concurrency. The scheduling time, start-up time and shipping time constitutes the scaling time. Since all of them scale up with concurrency (Fig. 2), the scaling time becomes the major bottleneck. Due to the architecture of serverless computing, the scaling time bottleneck is a fundamental issue and it cannot be avoided by simple manipulations by the cloud provider.

One intuitive solution to reduce the scaling time bottleneck is to group multiple functions together into smaller batches and spawn these batches serially. This solution decreases the level of concurrency (i.e., the number of function instances being invoked simultaneously), and hence, is promising to mitigate the scaling overhead. However, this approach is not suitable for multiple types of applications: (1) applications that require function instances to be executed simultaneously to take advantage of the parallelism of the serverless platform, and (2) applications (e.g., Sort) that have the turnaround time as a figure of merit (explicit serialization hurts the turnaround time). The serverless computing researchers have developed workload schedulers such as Pywren [34], which is widely used by the cloud and distributed computing community [9, 23, 38, 66, 72, 86]. Pywren performs optimizations like reusing serverless function instances (to avoid initial start-up latency or the cold start of serverless functions), optimizing the data movement pattern among serverless functions to reduce the scaling time and service time of serverless computing. This makes it useful at a low concurrency level, but it does not work equally effectively at a high concurrency level (Sec. 4).

ProPack: The Key Ideas. ProPack designs and implements a novel solution to tackle the scalability challenge on serverless platforms. The key idea behind ProPack is to "pack" multiple functions into one function instance and spawn all such instances concurrently. Packing multiple functions together does not decrease the parallelism that an application can exploit, but it does efficiently decrease the "effective" number of concurrent function instances spawned together, thereby, mitigating the scalability bottleneck.

However, because multiple functions are packed together in one function instance, it creates *two new non-trivial challenges*. The first challenge is directly related to the performance and the second challenge is related to the cost.

The Performance Challenge. Functions *packed* together in one instance interfere with each other's performances, as they run in parallel. Consequently, the execution time of each function instance increases; however, decreasing the "effective" number of concurrent function instances reduces the scalability overhead. In other words, ProPack's approach trades off the scaling time of running all instances with the execution time of individual instances. Therefore, ProPack needs to find a balance such that packing too many functions together to reduce the scalability overhead does not degrade the turnaround time due to the performance degradation of individual function instances.

The Cost Challenge. In the serverless computing model, users are only charged for the time they actually use the computing resources (i.e., execution time per function instance × memory consumption on the AWS Lambda platform). Recall that in the baseline case, although the total service time is severely affected by the scalability bottleneck, the end user does not pay the price for the queuing delay (i.e., the delayed start of function instances does not affect the billing). The price paid by users will be the same if there was no scalability bottleneck and all function instances were spawned instantaneously. However, with ProPack's approach, the execution time of each function instance increases, and hence, the cost can potentially increase. However, packing also reduces the number of concurrent function instances, thereby, decreasing the overall cost because the cost is proportional to the number of function instances. Therefore, ProPack needs to carefully determine how many functions to pack in one function instance.

Essentially, ProPack needs to effectively determine how many functions to pack together in one instance such that it is both performance and cost-aware. This is challenging because it depends on two factors: *(1) how the scaling time increases with the number of concurrent instances* (packing functions decrease the number of concurrent instances), and *(2) how co-located functions within a function instance affect each other's performance and the total cost.*

From our extensive experimentation on three serverless computing platforms (Amazon Lambda, Google Cloud Functions, and Microsoft Azure Functions), we observe that the scaling time increase is independent of the application being executed. That is, the scaling time depends on the number of concurrent function instances, but not on what code those instances are executing. *ProPack exploits this insight to build a simple-yet-effective analytical model that determines the "optimal packing degree" (i.e., the optimal number of functions packed together) for a given application by modeling the application-dependent interference among co-located functions and application-independent scaling behavior* to co-optimize for performance (total service time) and cost.

Packing affects both performance and cost. However, packing functions within one function instance affect performance (total service time) differently than how it affects cost – this is because the scaling time affects performance, but does not affect the cost (detailed discussion and modeling in Sec. 2). In fact, we show that the optimal packing degree can be different for different objectives (performance vs. cost) and for different applications.

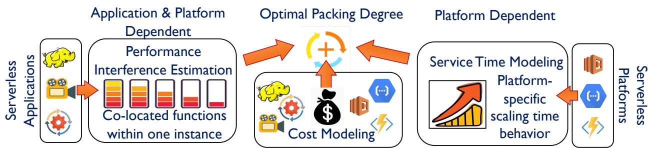
Application & Platform Dependent
Optimal Packing Degree
Platform Dependent
Serverless Applications
Performance Interference Estimation
Co-located functions within one instance
Cost Modeling
Service Time Modeling
Platform-specific scaling time behavior
Serverless Platforms

Figure 3: ProPack determines optimal packing degree.

ProPack: Evaluation. Our evaluation, using Amazon Lambda as the primary platform, demonstrates that ProPack's analytical model determines the optimal packing degree with more than 95% accuracy in most cases. Notably, ProPack becomes increasingly effective as the concurrency level increases, and reduces the overall service time by an average of 85% for a concurrency level of 5000. These improvements include the overhead incurred by ProPack to build its analytical model to capture application-specific performance behaviors. On average, ProPack reduces expenses incurred to the end user by 66% at a high concurrency of 5000.

Compared to the other *state-of-the-art* serverless workload manager, Pywren [34], ProPack reduces service time by 52% and expense by 78%, on average. We found that on-premise serverless execution platform for parallel applications, *FuncX* [11], performs better than cloud-based serverless platforms in terms of scaling issues. Our evaluation demonstrates that ProPack is also effective in mitigating the scalability bottleneck of the FuncX framework. Finally, we show that ProPack is portable and effective on multiple cloud-based serverless platforms.

ProPack's implementation is open-sourced for community adoption at https://github.com/rohanbasuroy/ProPack.

2 PROPACK: SOLUTION DESIGN AND IMPLEMENTATION

ProPack breaks down the task of determining optimal packing degree into three parts: (1) Performance interference estimation: ProPack estimates the performance degradation due to packing multiple functions in one function instance. (2) Modeling total service time: ProPack models the overall execution time at different concurrency levels by considering both the performance interference due to packing and scaling time overhead at a given concurrency level. (3) Modeling total cost (incurred expense to the user): ProPack models the effect of different packing degrees and corresponding concurrency levels on the total cost. The models developed by ProPack are driven by the experimental data and function-fitting from different production serverless platforms (Amazon Lambda, Google Functions, and Azure Functions) and validated against them, but should not be assumed to represent a physical system governed by a set of equations. The overall workflow of ProPack is illustrated in Fig. 3.

2.1 Performance Interference Estimation

In this section, we describe how ProPack estimates the performance interference among co-located functions within a single function instance. The purpose of this phase is to analytically model the effect of packing multiple functions together in one instance on the execution time of the function. As expected, packing more functions together increases the execution time of the function, and the effect is application-specific (as shown in Fig. 4).

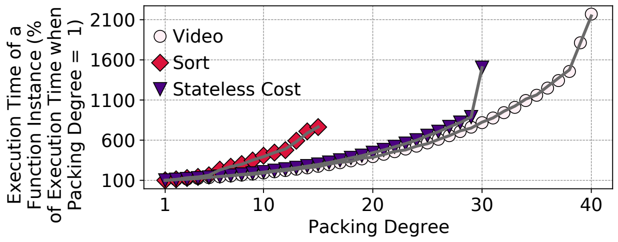

Figure 4: The execution time of a function instance increases with an increase in the packing degree. (The markers represent the observed data and the solid lines represent ProPack's analytical models).

Table 1: List of the variables used in ProPack formulation.

Variable	Description
P_i^{deg}	Packing degree i
M_{platform}	Maximum memory size allowed by the platform
M_{func}	Memory consumed by a single function
$ET_{P_i^{\text{deg}}}$	Execution time of a function instance at the packing degree P_i^{deg}
C	Concurrency level (number of concurrent function instances)
C_{eff}	Effective concurrency level at the packing degree P_i^{deg} ($C_{\text{eff}} = \frac{C}{P_i^{\text{deg}}}$)
$P_{\text{opt\_s}}^{\text{deg}}$	Optimal packing degree to minimize service time
$P_{\text{opt\_e}}^{\text{deg}}$	Optimal packing degree to minimize expense
$P_{\text{opt}}^{\text{deg}}$	Optimal packing degree to minimize service time and expense
$S(\mathbf{P^{deg}})$	Service time as a function of the packing degree
$E(\mathbf{P^{deg}})$	Expense as a function of the packing degree

To capture this relationship for a given application, we run a function instance multiple times with a varying "packing degree" (i.e., number of functions packed together). However, such a run (i.e., when multiple functions are packed in one instance) is ultimately limited by the memory requirement of the function and the maximum memory size imposed by the serverless platform; all functions combined in one instance cannot consume more than the maximum memory size (e.g., 10 GB on Amazon Lambda). One challenge is that capturing this relationship requires multiple runs, each with a different "packing degree". Fortunately, the function that captures this relationship is monotonic. Consequently, ProPack can approximate the curve (e.g., Fig. 4) by skipping alternate points and limiting the number of sample points that need to be evaluated. For example, ProPack needs to evaluate 20, 8, and 15 sample points for Video, Sort, and Stateless Cost respectively. Formally, we can express our model as follows (all relevant variables are also summarized in Table 1).

Let $P_{\text{i}}^{\text{deg}}$ denote the packing degree i. The value of i equal to 1 corresponds to the traditional case where only one function runs inside a function instance. $P_{\text{max}}^{\text{deg}}$ denotes the maximum number of functions that can be packed inside a function instance. $P_{\text{max}}^{\text{deg}}$ can be evaluated as $\frac{M_{\text{platform}}}{M_{\text{func}}}$ where M_{func} is the maximum memory consumed by a single function during its execution (known apriori, by running a single function once), and M_{platform} is the maximum memory size allowed by the platform. The execution time of a

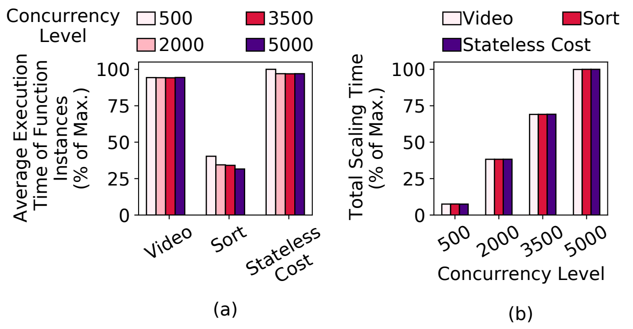

Figure 5: (a) The execution time of a function instance is largely unaffected by an increase in the concurrency level. (b) The scaling time overhead is independent of the application characteristics.

function instance at packing degree P_i^{deg} can be modeled as:

$$ET_{P_i^{\text{deg}}} = e^{M_{\text{func}} * \alpha * P_i^{\text{deg}}} \quad (1)$$

Here, $ET_{P_i^{\text{deg}}}$ is the execution time of a function instance at packing degree i, and α is a linear constant of proportionality. Note that the $ET_{P_i^{\text{deg}}}$ formulation includes the dependence on M_{func} as observed empirically (Fig. 4). $P_{\text{max}}^{\text{deg}}$ can also be configured to be constrained at a degree lower than $\frac{M_{\text{platform}}}{M_{\text{func}}}$, depending upon the maximum allowable latency of a function instance, which is determined by $ET_{P_{max}^{\text{deg}}}$ (e.g., meeting different quality of service (QoS) targets). Fig. 4 shows that ProPack's analytical model formulation of execution time fits the observed experimental data.

The overhead of estimating the performance interference. As noted earlier, ProPack needs to perform multiple runs, each with a different packing degree to determine the parameters of Eq. 1. This overhead is minimal because ProPack does not need to evaluate all the packing degrees and multiple evaluations can be performed in parallel – without incurring the concurrency bottleneck since the concurrency level is small (less than 100 function instance execution in parallel). Although the overhead of all consumed resources is minimal ($< 1\%$), our performance and cost results include all the overhead of building this analytical model.

Is the execution time of a function instance affected by the concurrency level, i.e., the number of concurrent function instances? Previously, we captured how packing multiple functions in one function instance causes interference. However, to fully understand the effect of packing on end-to-end performance, one needs to capture the effect of concurrency on the execution time of a single function instance. For example, if multiple function instances are spawned concurrently, does it affect the execution time of each function instance compared to their isolated execution? Our experimental evidence suggests that fortunately, concurrency level does not affect the execution time of function instances. The variation in execution time is less than 5% in most cases as the concurrency level increases from 500 to 5000. This is because cloud providers take measures for resource isolation among multiple co-running function instances [4]. Fig. 5(a) shows that on AWS Lambda, increasing the concurrency level does not increase the average execution time

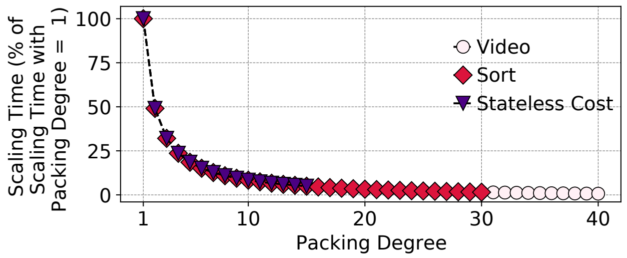

Figure 6: For a fixed level of concurrency (5000), scaling time decreases with an increase in packing degree.

of a single function instance (similar results were obtained for other platforms). This result also further affirms that a drastic increase in service time is not due to poor co-location of function instances at high concurrency levels, but it is instead due to varying lags in the start times of different function instances. A related implication of this result is that ProPack needs to model the effect of increasing concurrency on the scaling time.

2.2 Service Time Modeling

We demonstrated how to capture the effect of the packing degree on the execution time of a single function instance. Next, we discuss how to combine this knowledge with scaling time overhead to derive an expression for the total service time as this is the performance metric that the end users care about.

The key to modeling the service time is determining how the scaling time changes with the concurrency level (i.e., the number of concurrent function instances). The specific challenges are to determine (1) if the scaling time is affected by application characteristics, and (2) how the scaling time increases with the change in the concurrency level.

Our experiments (Fig. 5(b)) reveal that the scaling time increase is completely independent of the application characteristics, and a strong polynomial function of the concurrency level (i.e., the number of concurrent function instances). Recall that packing functions together effectively decrease the concurrency level, as of result of which the scaling time decreases (Fig. 6). Formally, if an application requested a concurrency level of C (i.e., C concurrent function instances, each with a packing degree of one in the baseline case), then the effective concurrency level under ProPack is $C_{\text{eff}} = \frac{C}{P_i^{\text{deg}}}$ for packing degree P_i^{deg}. ProPack models the scaling time as:

$$\text{Scaling Time} = \beta_1 \times C_{\text{eff}}^2 + \beta_2 \times C_{\text{eff}} - \beta_3 \qquad (2)$$

Here, $\beta_1, \beta_2, and \beta_3$ are linear coefficients that can be determined through polynomial regression. Note that in the traditional case, the packing degree is equal to one (i.e., $C_{\text{eff}} = C$). Second-order polynomial works reasonably well across different serverless computing platforms. The coefficients remain independent of the application but change across platforms. The useful implication is that ProPack can estimate the scaling time for different platforms in an application-independent manner via spawning different numbers of concurrent function instances. This scaling time estimation is then used by ProPack to model the total service time. After attempting several models like linear, quadratic, cubic, exponential, logarithmic, logistic, normal, and sinusoidal, we chose an exponential model and linear model for representing execution time (Eq. 1) and scaling time (Eq. 2), respectively, as they proved to be the best fit for the experimental data.

ProPack observes that although the execution of multiple function instances may overlap, the total service time is determined by the longest chain: the start of the last function instance and the time it takes to execute the function instance. Formally, the total service time at P_i^{deg} is essentially the scaling time ($C_{\text{eff}} = \frac{C}{P_i^{\text{deg}}}$) plus the execution time of a function instance ($e^{M_{\text{func}} * \alpha * P_i^{\text{deg}}}$). Therefore, ProPack can elegantly estimate the optimal packing degree for service time ($P_{\text{opt\_s}}^{\text{deg}}$) by estimating the total service time for all packing degrees and taking the minimum out of them (without needing to run the application at every packing degree or at high concurrency levels).

$$P_{\text{opt\_s}}^{\text{deg}} = \underset{1 \le P_i^{\text{deg}} \le P_{max}^{\text{deg}}}{\text{argmin}} \; [e^{M_{\text{func}} * \alpha * P_i^{\text{deg}}} + \beta_1 (\frac{C}{P_i^{\text{deg}}})^2 + \beta_2 (\frac{C}{P_i^{\text{deg}}}) - \beta_3] \qquad (3)$$

The overhead of modeling the service time. The only additional overhead is the estimation of scaling time, which requires fitting a polynomial regression model (Eq. 2). Fortunately, this fitting is inexpensive since evaluating a sample does not require the execution of any actual function code (scaling time behavior is independent of the application code) and can be well approximated by ten or fewer samples. This overhead is less than 1% of performance interference estimation. This model needs to be developed only once and can be used across all applications on a given platform. Our performance and cost results include the overhead of building this analytical model, even though it is only amortized over the few applications we evaluate; in practice, this overhead will be much lower due to amortization over thousands of applications and runs.

2.3 Cost Modeling

Recall that packing functions in one instance affect the total service time and user expense differently. This is because scaling time overhead affects the total service time directly, but queue wait time does not count toward the user bill.

To further demonstrate the competing trade-offs in terms of cost, we present experimental evidence to show that the cost may not decrease monotonically with an increase in the packing degree (Fig. 7). This is because increasing the packing degree decreases the number of serverless function instances (i.e., decreases the user expense). But at the same time, increasing the packing degree increases the execution time of a function instance. ProPack needs to model the cost by taking three factors into consideration: the execution time of a function instance (a function of the packing degree), the number of function instances (a function of the packing degree), and the expense rate for using the platform per unit time.

The incurred expense can be mathematically expressed as the multiplication between the execution time of a function instance at a given packing degree, the number of function instances to be spawned, and the expense rate of running a function instance per unit time (R). Thus, the optimal packing degree for minimizing the

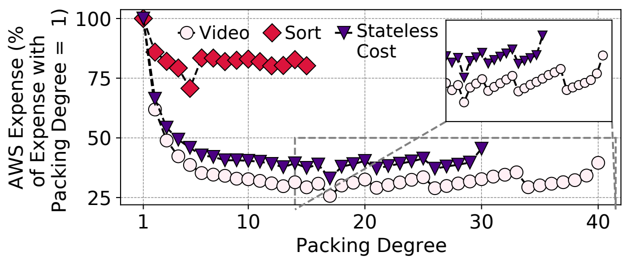

Figure 7: Expense incurred is not a monotonic function of packing degree (results shown for the concurrency level of 1000).

expense ($P^{\text{deg}}_{\text{opt\_e}}$) can be expressed as:

$$P^{\text{deg}}_{\text{opt\_e}} = \underset{1 \le P^{\text{deg}}_i \le P^{\text{deg}}_{max}}{\text{argmin}} \left[e^{M_{\text{func}} * \alpha * P^{\text{deg}}_i} \times R \times \left(\frac{C}{P^{\text{deg}}_i}\right)\right] \tag{4}$$

Note that the exponential part in $P^{\text{deg}}_{\text{opt\_e}}$ increases with higher packing degree, while $\frac{C}{P^{\text{deg}}_i}$ decreases with it. We note that the above expression (Eq. 4) is not the same as the optimal degree expression for service time (Eq. 3). As expected, our evaluation also shows the optimal degree for service time and cost are different. Therefore, we need to analytically determine the optimal packing degree that jointly optimizes both objectives.

The overhead of modeling the cost. We note that this step does not require running any additional experiments. The optimal degree for cost optimization can be analytically determined using the knowledge obtained in the previous two steps (modeling the performance interference and the service time).

2.4 Validation of ProPack's Analytical Models

We validate the effectiveness of ProPack's analytical models by performing the Pearson χ^2 Goodness-of-Fit test, which is a well-known statistical test for model fit validation [27]. In this test, the χ^2 value is calculated as the sum of the squared difference between the observed value and expected value (according to the analytical model) as a fraction of the expected value across different packing degrees. This value is then compared against the χ^2 distribution. In our case, the degree of freedom of this distribution is $15 - 1 = 14$, where 15 is the maximum packing degree of the Sort application (this is the lowest maximum packing degree across all applications). Using a high p-value of 0.995 (we use a confidence of 99.5%, which is higher than the typically used 95% value), the χ^2 value with 14 degrees of freedom is 4.075. If our χ^2 test statistic is lower than this value, we accept the null hypothesis that the observed and expected values are from the same distribution with a 99.5% confidence. We observe that this is indeed the case. Across all applications and concurrency values, our maximum statistical test value is 3.81 for service time and 0.055 for the expense. Thus, we can conclude that ProPack's analytical models are representative of the observed service time and expense characteristics with high confidence.

This is important because ProPack's accuracy in estimating service time and expense needs to be high enough to obtain the most optimal packing degree without performing the exhaustive brute force search, which would have been necessary otherwise. ProPack is effective because the validated analytical-model-driven predictor is accurate at determining the optimal packing degrees for both scaling time and overall expense. Next, we discuss how we can combine the two models to obtain a packing degree that balances service time and expense.

2.5 Putting It All Together: The Optimal Packing Degree

Recall that $P^{\text{deg}}_{\text{opt\_s}}$ and $P^{\text{deg}}_{\text{opt\_e}}$ are different due to the fact that scaling time plays a direct role in determining $P^{\text{deg}}_{\text{opt\_s}}$, while it does not affect the incurred expense. On the other hand, the number of function instances spawned plays a direct role in determining $P^{\text{deg}}_{\text{opt\_e}}$. Finally, in this section, we describe how ProPack determines the optimal packing degree ($P^{\text{deg}}_{\text{opt}}$) that gives equal importance toward optimizing both service time and expense, although it can be easily configured to place the desired unequal weights on both the objectives.

At any given level of concurrency (C), the service time is expressed as the function $S(P^{\mathbf{deg}})$. This is the same as the summation of the execution time of a function instance and the scaling time, expressed as the argument in Eq. 3. The expense can be expressed as the function $E(P^{\mathbf{deg}})$, which is same as the argument to determine $P^{\text{deg}}_{\text{opt\_e}}$ in Eq. 4.

Striking the balance between $P^{\text{deg}}_{\text{opt\_s}}$ and $P^{\text{deg}}_{\text{opt\_e}}$, ProPack determines the optimal packing degree ($P^{\text{deg}}_{\text{opt}}$). which jointly minimizes both service time and expense. ProPack measures the fractional change of service time from the most optimal service time $S(P_{\text{opt\_s}})$, for different packing degrees as:

$$\Delta S_{P^{\text{deg}}_i} = \frac{S(P^{\text{deg}}_i) - S(P^{\text{deg}}_{\text{opt\_s}})}{S(P^{\text{deg}}_{\text{opt\_s}})} \tag{5}$$

Similarly, ProPack evaluates the fractional change of expense with respect to the most optimal expense $E(P^{\text{deg}}_{\text{opt\_e}})$, for different packing degrees as:

$$\Delta E_{P^{\text{deg}}_i} = \frac{E(P^{\text{deg}}_i) - E(P^{\text{deg}}_{\text{opt\_e}})}{E(P^{\text{deg}}_{\text{opt\_e}})} \tag{6}$$

ProPack then sets the weights to the respective objectives of reducing service time and expense, W_S and W_E, to 0.5, to give equal importance to both objectives. These can be modified (but they must sum up to 1) to give more importance to one of the objectives over the other. ProPack's optimal packing degree ($P^{\text{deg}}_{\text{opt}}$) is the one that reduces the combined relative change in both expense and service time from their most optimal value.

$$P^{\text{deg}}_{\text{opt}} = \underset{1 \le P^{\text{deg}}_i \le P^{\text{deg}}_{max}}{\text{argmin}} \left[W_S \times \Delta S_{P^{\text{deg}}_i} + W_E \times \Delta E_{P^{\text{deg}}_i}\right] \tag{7}$$

Packing multiple functions into one serverless function instance is a solution to both reducing the service time and the expense incurred to the user. ProPack leverages this to find the optimal solution, which jointly optimizes both objectives through its mathematical formulations. For a fixed level of concurrency, packing

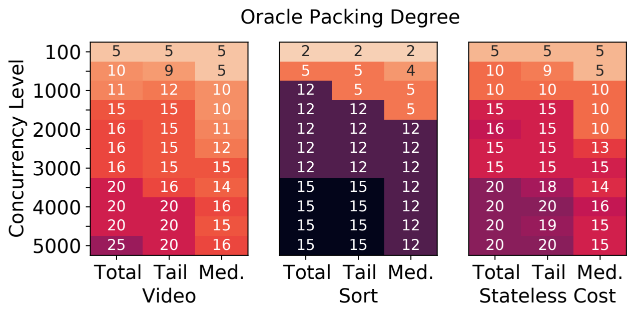

Figure 8: Variation of Oracle packing degree (maximum packing degree of Video, Sort, and Stateless Cost is 40, 15, and 30, respectively). ProPack determines all the packing degrees correctly, except in two cases – for Sort, at a concurrency of 2000, ProPack determines a total packing degree of 14 (Oracle packing degree is 12), and for Stateless Cost, at a concurrency of 1000, ProPack determines a tail packing degree of 9 (Oracle packing degree is 10).

decreases the number of function instances to spawn, as a result of which scaling time decreases. However, packing increases the execution time of individual instances. ProPack's optimization objective is to find the optimal packing degree, ensuring that the magnitude of the increase in execution time of individual instances is much lesser than the decrease in scaling time – this results in an improvement in service time. To gain further insight into the effectiveness of this approach, Fig. 8 shows the optimal (Oracle) packing degree for different currency levels. We make two important observations: (1) As the concurrency level increases the packing degree also increases. This is because scaling time increases with a higher gradient than the packing cost (i.e., an increase in the function execution time) and hence, it is beneficial to pack more functions together. (2) However, determining how many functions to pack together is non-trivial and depends on application properties, the concurrency level, and the objective. ProPack's analytical modeling captures this interaction without incurring an extensive cost.

2.6 QoS Aware Optimal Packing Degree

Serverless computing is often used for running latency-critical applications with strict Quality of Service (QoS) bounds on tail latency of service time. This is because these applications are short running, stateless, and have varying invocation counts. A packing degree of $P^{\text{deg}}_{\text{opt}}$ (Eq. 7) jointly optimizes for both service time and user expense. However, with the default value of 0.5 for both W_S and W_E, it does not ensure that the service time of the functions will be within the QoS tail latency bound. If having equal weights on optimizing for both objectives violates the QoS guarantee, the weight on optimizing service time needs to be increased so that the tail latency falls within the QoS bounds. The tail service time at a packing degree which jointly optimizes both the objectives can be expressed as:

$$TS_{W_S,W_E} = Tail(S(\underset{1 \leq P_i^{\text{deg}} \leq P_{max}^{\text{deg}}}{\text{argmin}} [W_S \times \Delta S_{P_i^{\text{deg}}} + W_E \times \Delta E_{P_i^{\text{deg}}}])) \quad (8)$$

Here, the weights W_S and W_E ($= 1 - W_S$) are to be chosen in a way such that the estimated TS_{W_S,W_E} is minimized and it is lower than the QoS bound.

$$W_S = \underset{0 \leq W_S \leq 1}{\text{argmin}} [TS_{W_S,1-W_S} \mid TS_{W_S,1-W_S} \leq QoS] \quad (9)$$

The obtained values of W_S and W_E from Eq. 9 can be used in Eq. 7 to obtain the optimal packing degree ($P^{\text{deg}}_{\text{opt}}$) that optimizes for both service time and user expense, while maintaining the tail latency of service time within the required QoS bound.

Practical realization of function packing. Packing multiple functions inside one serverless function instance is implemented by spawning each of the functions separately as individual software threads using the *thread* library of Python3.6. Thus multiple functions share 10 GB of memory and 6 CPU cores in a function instance (function instances of Amazon Lambda). Note that, the traditional implementation of Python is constrained by the global interpreter lock (GIL), which is known to have concurrency limitations because it can serialize the control of the Python interpreter. Hence, by normally using software threads, it is challenging to scale up the functions in the multiple cores of a serverless function instance. To mitigate this challenge, ProPack uses a widely-used CPython extension that performs multi-threading without GIL [1]. It has several alternative mechanisms for thread safety, with minimal impact on single-threaded code performance. Also, if the optimal packing degree determined by ProPack is larger than the memory limit enforced by the cloud provider for a single function, then ProPack's packing degree can be modified to ensure that it does not violate the memory limit enforced by the cloud provider - treating that as a constraint. From the user's point of view, ProPack only requires the application executable with the dependencies. It performs packing and determining the optimal packing degree.

3 EXPERIMENTAL METHODOLOGY

Serverless Platforms. We use AWS Lambda, one of the most widely used commercial serverless providers [53, 81], as the primary testbed for our evaluation. AWS Lambdas run on microVMs scheduled on AWS Elastic Compute Cloud (EC2) servers for their execution [4, 22, 81]. To invoke Lambdas concurrently, we use the "Step Functions" framework as it provides dynamic parallelism [20, 24]. We use Lambdas with the maximum memory size (10 GB) to achieve a considerable maximum packing degree. For storing results and intermediate application data, we use the AWS S3, and its cost is included in the overall expense analysis. We also evaluate the effectiveness of ProPack on two other widely used platforms, Google Cloud Functions and Microsoft Azure Functions [54, 70].

To take advantage of the massive concurrency available in serverless, the HPC community has developed serverless workload managers like FuncX [11]. It is designed to run embarrassingly parallel scientific computations on clusters. It spawns the processes of parallel applications as separate serverless instances via Docker containers. We evaluate ProPack using FuncX as a workload manager on a 100 node cluster of *r5.2xlarge and r5.4xlarge* VMs of Amazon EC2, having a total of 1000 cores and 20,608 GB of memory, by running high performance data parallel applications.

Benchmarks. For evaluation, we use the following benchmarks that are widely used for serverless benchmarking and represent diverse real-world serverless applications. These benchmarks perform massively parallel data analytics and information processing tasks, which are some of the very frequently used components of HPC workflows [18, 25, 76, 85].

Thousand Island Scanner Video Processing (Video) [60] performs distributed video processing using serverless functions (similar to Netflix applications to encode media files). 5.2 MB of video input from TV News Scanner's database is provided to the serverless functions for encoding and classification by MXNET DNN. Processing multiple chunks of the videos can be performed in parallel.

Map Reduce Sort (Sort) [39, 58] is a Hadoop-based implementation of a sorting algorithm. It has a mapper class that takes the input data and divides it into multiple arrays, each to be sorted by separate serverless functions. The work done by all the functions is finally written to a shared file in S3.

Stateless Cost performs image resizing and is included in the open-source serverless benchmark suite [87]. This application is suitable for serverless computing as it has relatively low execution time with multiple stateless requests that can be executed in parallel. Serverless image handler by AWS performs similar tasks [3].

Smith-Waterman is a massively parallel HPC application that performs dynamic computation for comparing protein sequences [7, 19, 67, 68]. It is a core bioinformatics [17] application that is shown to benefit from serverless computing [2, 13, 56].

Xapian is a search engine, which is a typical latency-critical, compute-intensive workload with a strict QoS bound on tail (95^{th} percentile) latency [32, 36]. It searches queries on Wikipedia pages.

Evaluation Metrics and Competing Techniques. For a level of concurrency, ProPack predicts different packing degrees that jointly minimize total, tail, and median service times, scaling times, as well as expenses. Here total, tail, and median service times refer to the time required till the end of execution of all, first 95% and first 50% concurrent function instances, respectively.

We perform an exhaustive brute force search to determine the optimal packing degree (Oracle packing degree). For benchmarks such as Sort, which have multiple concurrent function instances that work together to perform a single application, minimizing total service time is a suitable figure of merit. However, for benchmarks such as Video and Stateless Cost that serve requests individually, minimizing median or tail service time can be a more suitable figure of merit for performance or meeting QoS targets. We also evaluate the performance of ProPack with the goal of minimizing only the service time (*ProPack (Service Time)*). This is useful when there is a requirement for workloads to run under time constraints to meet specific deadlines. Likewise, we also evaluate ProPack with minimizing expense as the only goal (*ProPack (Expense)*). This is useful for workloads that need to run under budget constraints.

We report ProPack's percentage improvement in service time, scaling time, or expense over spawning serverless instances in the traditional way, with no packing (packing degree = 1).

4 PROPACK: EVALUATION AND ANALYSIS

Is ProPack effective in improving the scaling time and the total service time of applications? Fig. 9 shows the total service

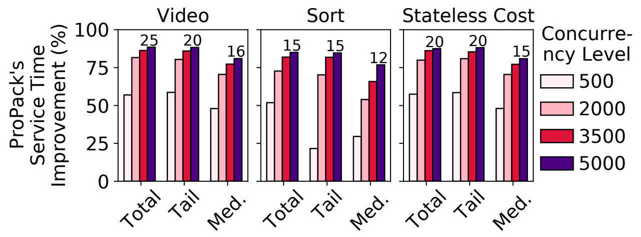

Figure 9: ProPack improves service time (% improvement over no packing, packing degree =1) by more than 85% for a concurrency level of 5000 (results for AWS Lambda). The numbers over the bars denote the packing degree for a concurrency level of 5000.

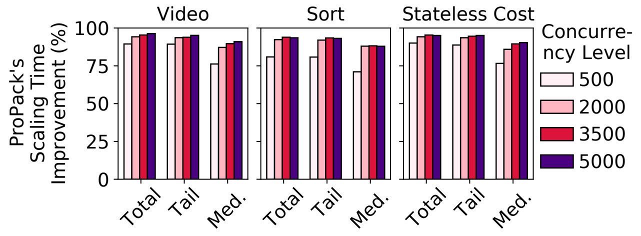

Figure 10: ProPack's improvement in scaling time (% improvement over no packing, packing degree =1) increases with the level of concurrency (results for AWS Lambda).

time when using ProPack, compared to the baseline case when multiple functions are not packed together in one function instance ($P^{deg} = 1$). We observe that ProPack reduces the total service time for all applications and at all concurrency levels, by more than 50% in most cases. ProPack yields higher benefits as the concurrency level increases. For example, at a 5000 concurrency level, ProPack reduces the total service time by 85% on average compared to the baseline. This is because at higher concurrency levels, the scaling time becomes more prominent and ProPack reduces that scaling time by packing multiple functions in one function instance and hence decreasing the effective concurrency level itself.

This is substantiated in Fig. 10, which shows the scaling time reduction for all applications at different concurrency levels. As expected, the amount of reduction is larger at higher concurrency levels. For example, at a concurrency level of 5000, the reduction in scaling time is often more than 90%. We note that this is higher than the reduction obtained in the total service time. This is because packing multiple functions in one function instance increases the execution time of each function instance and that eats up some of the benefits of reducing the effective concurrency level.

We also observe that ProPack achieves faster service and scaling time for all figures of merit (total, tail, and median service times) and therefore, is useful for different types of applications that may prefer different latency metrics (e.g., Sort v/s. Video).

Finally, we note that all ProPack's results include the exploration overhead when it estimates the effect of packing degree on the function execution time. This overhead is specific to ProPack only and is not included in the baseline. ProPack yields significant benefits despite this overhead because in practice the exploration phase requires pre-running a function only a few times and it is

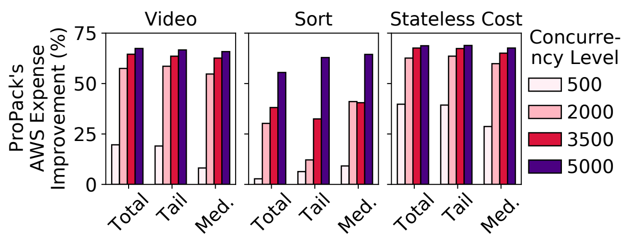

Figure 11: ProPack can reduce AWS expense (% improvement over no packing, packing degree =1) by an average of 66% at high concurrency (results for AWS Lambda).

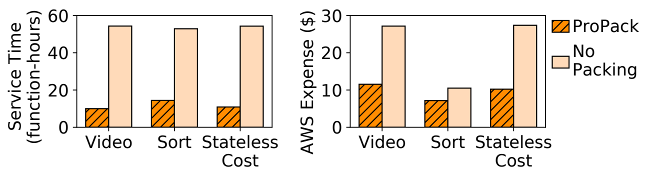

Figure 12: The reduction in the absolute value of service time and expense by ProPack is significant (concurrency level of 2000).

independent of the concurrency level of the application. Therefore, this overhead does not scale with the concurrency level.

Can ProPack reduce the overall expense incurred for an end user? We saw in Sec. 2.3 that packing multiple serverless functions inside one function instance has the potential to reduce the end-user expense as the number of spawned function instances decreases. ProPack finds the optimal packing degree such that a reduction in expense is observed for all the applications at all levels of concurrency (Fig. 11). On average, a 66% reduction in expense is observed for a concurrency level of 5000. Note that expense is not affected by the scaling time as the users of serverless computing are only billed for the time when they are actually executing functions in the function instances. However, still with an increase in concurrency, ProPack results in more reduction in expense. This is because the scope for packing multiple functions in one function instance increases with the level of concurrency and also the execution time of each function instance increases in a sub-linear manner with an increase in packing degree (Eq. 1). Across all figures of merit (total, tail, and median), the reduction in expense remains similar, in most cases.

The reported performance and cost numbers needed to be normalized as the absolute values vary widely with different levels of concurrency (i.e, different bars in Fig. 9 and Fig. 11) and hence, the trends in performance benefits are not visible unless they are normalized. Serverless platforms limit the maximum execution time (e.g., 15 minutes). Due to scaling time bottlenecks, choosing a long execution time per function instance, timeouts the baseline case (no packing) at high concurrency. Also, for statistical significance, one needs to repeat each experiment multiple times - even setting the execution time in order of minutes can cost thousands of dollars.

For reference, we have included Fig. 12 with absolute values (both for service time and expense) at a concurrency level of 2000. Note that, similar to HPC node-hour resource accounting, we are reporting execution time as function hours (sum of execution time

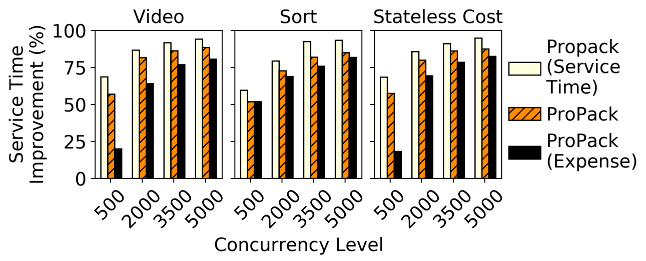

Figure 13: Under time constraint, ProPack (service time as objective) can improve total service time by an average of 7.5% over ProPack (both service time and expense as objectives, % improvement over no packing, packing degree =1, results for AWS Lambda).

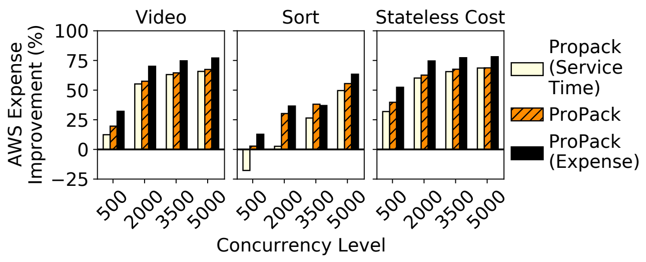

Figure 14: Under expense constraint, ProPack (expense as objective) can improve total AWS expense by an average of 9.3% over ProPack (both service time and expense as objective, % improvement over no packing, packing degree =1, results for AWS Lambda).

across all function instances; each function instance executed for approximately 100 seconds). ProPack reduces the total service function hours from more than 50 hours to less than 14 hours and expenses from more than $25 to less than $12. Similarly, at a 5000 concurrency level, the cost is reduced from $75 to $33. Note that, these benefits accumulate over multiple runs of an application, especially noting a cost benefit of more than double.

Does ProPack perform effectively when the objective is different than placing equal weights on service time and expense? From Fig. 13 we note that ProPack's effectiveness in reducing the service time is 7.5% higher when ProPack determines the packing degree with only service time reduction as the sole objective, compared to the case when ProPack tries to reduce both service time and expense with equal priority. This can provide additional benefit when ProPack needs to serve serverless functions which have a strict time budget. Note that, in this case, due to packing the expense also reduces. However, the observed reduction in expense is 7.6% lower than when ProPack has both service time as well as expense reduction as the objective.

Likewise, when ProPack determines the packing degree with the reduction in expense as the only objective, it reduces expense by more than 9.3% compared to the case when ProPack reduces the expense and scaling time with equal emphasis (Fig. 14). However, service time increases by 12.3%.

Fig. 15 shows the most optimal (Oracle) packing degree when only service time reduction or expense reduction is the sole objective. We observe that the Oracle packing degree increases when the

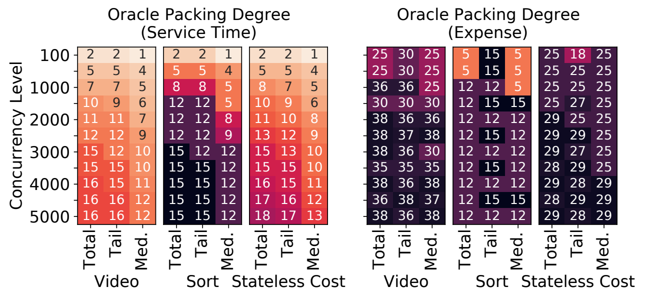

Figure 15: Oracle packing degree increases when expense minimization is given higher importance than service time minimization. ProPack finds all the packing degrees correctly except in one case while determining Oracle packing degree (expense) – for Sort, at a concurrency of 1500, ProPack determines a median packing degree of 13 (Oracle packing degree is 15).

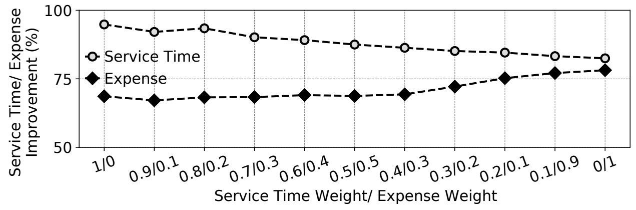

Figure 16: ProPack is effective for different values of service time weight and expense weight (results shown for Stateless Cost for a concurrency level of 5000, % improvement over no packing).

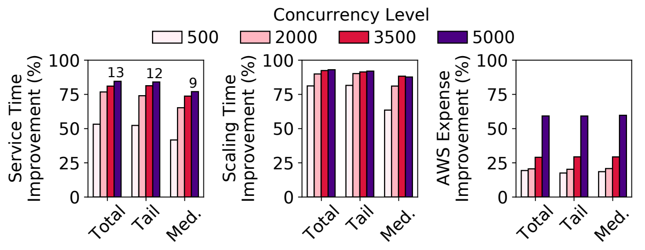

Figure 17: ProPack improves service time, scaling time, and expense (% improvement over no packing, packing degree =1) of Smith-Waterman (results for AWS Lambda).

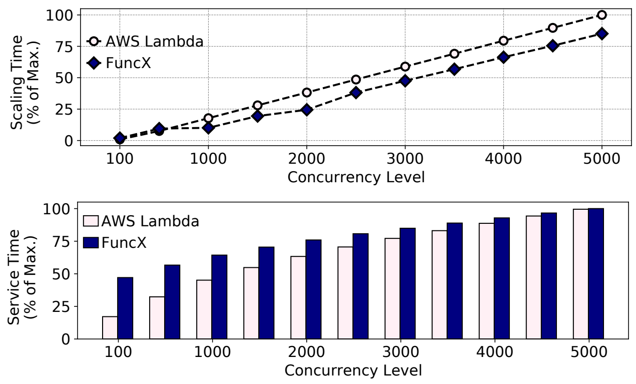

Figure 18: FuncX results in lower scaling time of serverless workers compared to AWS Lambda. On average, ProPack results in 12% lower total service time in AWS Lambda than in FuncX.

objective is to reduce the expense only. Again, when the objective is to reduce both service time and expense, the Oracle packing degree falls in between these two (Fig. 8). As the objective leans toward reducing expense more than service time, the major benefit comes by reducing the effective concurrency ($C_{\text{eff}} = \frac{C}{P_i^{\text{deg}}}$). This is because, the expense experiences a multiplicative increase (argument of Eq. 4) with an increase in the effective concurrency, while the service time only experiences an additive increase (argument of Eq. 3). This establishes the fact that ProPack's analytical model is correctly able to represent the true nature of service time and expense of serverless function instances.

From Fig. 16 we observe that ProPack is effective in reducing service time and expense for different values of the service time weights (W_S) and expense weights (W_E) of Eq. 7. As expected, as the expense weight increases, ProPack obtains more improvement in expense, and as the service time weight increases, ProPack results in more reduction in service time. A experimental variation manifests as a minor dip at $\frac{0.9}{0.1}$, but does not alter the overall trend.

Is ProPack effective in improving the performance of a widely-used, parallel bioinformatics application? Like other benchmarks, ProPack also improves the service time, scaling time, and expense of an HPC application, Smith-Waterman. This application is used to perform a large number of parallel and independent computations, and hence serverless computing is a promising computational model for it [2]. However, increased scaling time due to huge concurrency can pose a challenge in using serverless [13, 56]. Through optimal packing of multiple functions, ProPack solves this problem, improving the service time and expense by 81% and 59% respectively, for a concurrency level of 5000 (Fig. 17). Though the maximum packing degree of Smith-Waterman is 35, the Oracle packing degree is much lower (shown by the numbers over the bars in Fig. 17). Packing a large number of functions is inefficient for this application as its functions are compute-intensive.

How effective is ProPack with serverless HTC/HPC deployments like FuncX? FuncX is an HTC/HPC-focused serverless scheduler, which is used to spawn and manage resource-intensive, embarrassingly parallel, and loosely-coupled HPC workloads using serverless function instances on any cluster [11, 12, 37, 51]. A serverless deployment of HTC/HPC workloads makes the spawning process easier for scientific researchers as now they do not have to configure nodes and wait to receive the allocations in a cluster. HTC/HPC workloads run hundreds of processes in parallel, and hence, it is particularly important for these workloads to elastically scale with efficiency. We evaluated the performance of ProPack on a cluster (described in Sec. 3) which spawns serverless function instances using FuncX. Serverless function instances spawned with FuncX scale faster than AWS Lambdas, especially at high concurrency. At a concurrency of 5000, FuncX scales 15% faster than AWS Lambda (Fig. 18). This improvement in scaling time over AWS Lambda happens because FuncX spawns worker processes in Kubernetes pods, which have less start-up time than

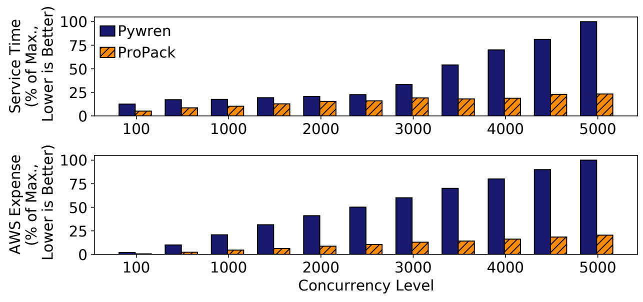

Figure 19: ProPack reduces service time and expense compared to the *state-of-the-art* serverless workload manager, Pywren.

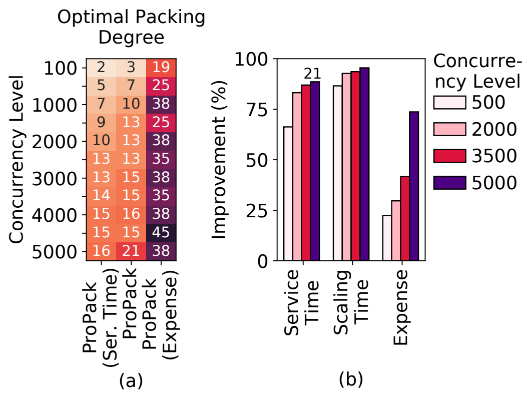

Figure 20: (a) For Xapian, the packing degree for minimizing tail service time increases as more importance is given to minimizing expense. (b) ProPack jointly minimizes tail service time and expense while maintaining QoS bound of Xapian.

microVMs (AWS Lambda spawns Firecracker microVMs). This is because FuncX co-locates multiple workers inside one pod, but, AWS Lambdas may not co-locate function instances. Also, FuncX utilizes Kubernetes' in-built container caching, which helps it to scale faster. However, with packing, ProPack can reduce service time more by 12% in AWS Lambda than on FuncX (Fig. 18). Firecracker microVMs are optimized for sharing network, computing, and storage resources. With concurrency, multiple serverless function instances run on a single node. With the efficient resource isolation of Firecracker microVMs, serverless function instances execute faster in AWS Lambdas, than on FuncX.

How does ProPack perform in comparison to the state-of-the-art serverless workload manager, Pywren?. We observe that ProPack significantly improves service time as well as expense over the state-of-the-art serverless workload manager, Pywren (Fig. 19). Pywren tries to execute serverless function instances faster by carrying out optimizations to reduce the cold start of function instances, reusing instances at high concurrency so that the software dependencies need not be loaded for each of them separately, and optimizing the data movement pattern among concurrent instances via common storage. While these optimizations are beneficial, they do not directly aim to solve the main source of inefficiency – the increased scaling time at high concurrency.

Pywren's optimizations can be workload-specific and benefit workloads that are friendly to function reuse. Pywren's reuse optimization, while useful, can waste resources if the subsequent invocations do not use the same degree of resources and requires knowing which function instances can be used by which instances. ProPack does not rely on function reuse. Alternative to packing, we also attempted other latency-hiding techniques such as staggering instances, but such techniques result in severe service degradation due to inserted delays and are unsuitable for workloads that need synchronous progress. Our evaluation shows that ProPack is invariant of the workload type and characteristics, and on average reduces the service time by 52% and expense by 78% over Pywren.

Can ProPack reduce service time and expense while maintaining the QoS? Latency critical applications, like Xapian, are frequently executed on serverless computing platforms as it is stateless and requires elastic scaling. However, it has strict QoS bounds on tail latency. ProPack finds the optimal packing degree $P_{\text{opt}}^{\text{deg}}$ by determining the weights on optimizing service time (W_S) and expense (W_E) to maintain the QoS as per Eq. 9. From Fig. 20(a), we see that the optimal packing degree when ProPack optimizes for both service time and expense satisfying the QoS (middle column), falls in between the optimal packing degree of ProPack (Service Time) and ProPack (Expense). This is because packing more functions inside one function instance is beneficial for reducing cost as less number of function instances are spawned, but it increases the service time because of an overall increase in execution time due to packing. For Xapian, by giving 65% weight to optimizing service time, and 35% weight to optimizing expense, ProPack maintains the QoS while optimizing service time and expense (Fig. 20(b)). ProPack achieves more than 80% improvement in service time and 65% improvement in expense for a concurrency of 5000.

Does ProPack provide benefits across different serverless offerings like Google Cloud and Microsoft Azure Functions? Our results (Fig. 21) indicate that ProPack is cost-effective and reduces the service time across multiple serverless computing platforms. Fig. 21 shows that the expense improvement in AWS is lower than with other platforms. This is because Google and Azure charge a networking fee (per GB transfer) for serverless functions, which is significantly reduced due to the co-location of functions done by ProPack. This fee is not charged by AWS.

5 RELATED WORK AND DISCUSSION

Several works in the past have focused on designing fast serverless schedulers [61, 78, 79] and managers of computing resources among serverless functions to make them meet the required QoS [26, 39, 64, 65, 77]. Some of the works focused on the usage of specialized hardware and storage mediums to improve the throughput of serverless applications [5, 40, 69, 88]. To further benefit serverless performance, the community has developed specialized lightweight virtualization and containerization mechanisms with low set-up times [4, 50, 57, 63, 73]. Benchmarking of serverless platforms has also been performed to characterize the problem of large scaling overhead at high concurrency [41–43, 45, 59, 62, 75, 81, 82]. This overhead affects the response time of

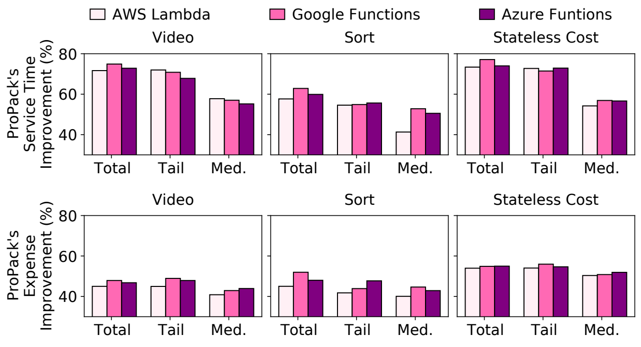

Figure 21: ProPack improves service time and expense (% improvement over no packing, packing degree =1) across multiple cloud platforms (concurrency level of 1000).

several serverless applications [29, 30, 55], like the ones used in the evaluation of ProPack. Some of the recent works faced this problem while running machine learning and data analytics applications on serverless platforms [6, 71]. Some of the works tried to workaround this overhead by performing several VM and container-based optimizations [61, 74], but none of them mitigates scalability at high concurrency levels as "high concurrency in a serverless platform makes it harder to achieve low-latency startup"[87]. Also, previous works have shown that serverless function instances are not always well utilized and can increase end-user expense unexpectedly, especially at high concurrency [16, 44, 49, 52, 54, 83]. This can happen due to several reasons like increase in function instance execution time, network bandwidth contention, or overwhelming the servers where function instances are executing [15, 31, 39, 40, 47, 72, 89]. However, these works do not explicitly provide strategies which decrease these high concurrency serverless computing issues.

Next, we discuss recent works that address scalability bottlenecks. Wukong [10] is a serverless DAG scheduler that improves the performance, scalability, data locality, and execution cost of DAGs. Wukong provides decentralized scheduling using a combination of static and dynamic scheduling approaches. It uses several optimizations like utilizing multiple storage clusters, delayed I/O, and task clustering for its effectiveness. Wukong's key insight is that scheduler decentralization improves serverless performance.

FaaSNet [80] is a middleware system designed to address the challenges of rapidly provisioning custom-container-based FaaS infrastructure, which is necessary for serverless computing to achieve high elasticity. It enables decentralization of container provisioning processes across host VMs organized in function-based tree structures, providing high scalability and adaptivity. It also employs a tree-balancing algorithm that dynamically adapts the function tree topology to accommodate the joining and leaving of VMs.

Owl [21] proposes a new system for large content distribution that satisfies the requirements of speed, efficiency, and reliability. It has a split design with a decentralized data plane and a centralized control plane. The decentralized data plane streams data from sources to clients via a distribution tree, while the centralized control plane makes detailed policy decisions about content distribution. Owl strikes the right balance between decentralization and centralization, and it is flexible enough to meet the requirements of many different types of services that require content distribution.

The commonality in prior approaches like Wukong [10], FaaS-Net [80], and Owl [21] is to strike a balance between centralization and decentralization. We recognize that centralized solutions, like ProPack, can further benefit from decentralization approaches. However, these approaches are not necessarily competitive. One important point to note is that decentralization is not free, may continue to be prone to scalability bottlenecks at high concurrency, and may not always be under the end-user's control depending upon the implementation. We envision ProPack-like ideas to work in conjunction with decentralized scheduling methods – they can be designed to be complementary and not necessarily competitive, especially at very high concurrency. In fact, excessive decentralization may induce high synchronization and communication overhead (potentially causing straggler behavior) – therefore, user-side mitigation such as packing can be complementary in nature.

Interaction with the cloud provider side. We recognize that cloud providers are continuously making improvements in their back end to mitigate scalability bottlenecks for spawning a large number of concurrent functions. The key ideas presented in ProPack are primarily on the user side and can work in conjunction with cloud provider-side mitigation. One important implication is that if the cloud provider side mitigation is effective, the optimal packing degree for ProPack is likely to decrease – which is desirable for functions with large memory footprints. In fact, function packing may also be indirectly beneficial to cloud providers in certain cases, as function packing improves resource utilization.

Interaction with security concerns. We recognize that there is a potential interplay between security aspects of ProPack's packing. ProPack is primarily intended for a single user, in its current form, to avoid potential security concerns. Technically, it is possible to extend ProPack to multiple users, but it may have multiple security caveats. Also, packing functions of different characteristics present new modeling challenges – ProPack can be extended to account for those, but it does not do so currently because it is not aware of what different kinds of users can collaborate to pack their functions.

6 CONCLUSION

ProPack proposes a solution to the scalability challenge in executing concurrent function instances faster and cheaper. It requires optimal determination of packing to mitigate serverless-specific challenges (packing affects performance differently than how it affects cost). Using this solution, end users can improve the total service time of their applications by an average of 85% at high concurrency and reduce cloud computing cost by 66% on multiple serverless computing platforms including Amazon, Microsoft, and Google. ProPack is especially useful for parallel HPC applications, which require a high concurrency of serverless function instances.

ACKNOWLEDGMENTS

We thank our anonymous shepherd and reviewers for their valuable feedback. This work is supported by NSF Award 1910601, 2124897, and Northeastern University. This work was also supported in part by Argonne National Laboratory under U.S. Department of Energy under Contract DE-AC02-06CH11357 and used resources of the Argonne Leadership Computing Facility.

REFERENCES

[1] Python Multithreading without GIL, url: https://github.com/colesbury/nogil.

[2] Building high-throughput genomics batch workflows on aws: Introduction| aws compute blog. https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-introduction-part-1-of-4/, 2017.

[3] *Serverless Image Handler Implementation Guide.* Amazon Web Services, 2020.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 419–434, 2020.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.

[6] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 972–986. IEEE Computer Society, 2020.

[7] Jeff Allred, Jack Coyne, William Lynch, Vincent Natoli, Joseph Grecco, and Joel Morrissette. Smith-waterman implementation on a fsb-fpga module using the intel accelerator abstraction layer. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–4. IEEE, 2009.

[8] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[9] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.

[10] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 1–15, 2020.

[11] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 65–76, 2020.

[12] Ryan Chard, Tyler J Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. Serverless supercomputing: High performance function as a service for science. *arXiv preprint arXiv:1908.04907*, 2019.

[13] Rodrigo Crespo-Cepeda, Giuseppe Agapito, Jose Luis Vazquez-Poletti, and Mario Cannataro. Challenges and opportunities of amazon serverless lambda services in bioinformatics. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 663–668, 2019.

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

[15] Adam Eivy and Joe Weinman. Be wary of the economics of" serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[16] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[17] Anton J Enright, Ioannis Iliopoulos, Nikos C Kyrpides, and Christos A Ouzounis. Protein interaction maps for complete genomes based on gene fusion events. *Nature*, 402(6757):86–90, 1999.

[18] Mazen Farid, Rohaya Latip, Masnida Hussin, and Nor Asilah Wati Abdul Hamid. Scheduling scientific workflow using multi-objective algorithm with fuzzy resource utilization in multi-cloud environment. *IEEE Access*, 8:24309–24322, 2020.

[19] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[20] Rustem Feyzkhanov. *Hands-On Serverless Deep Learning with TensorFlow and AWS Lambda: Training serverless deep learning models using the AWS infrastructure.* Packt Publishing Ltd, 2019.

[21] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and flexibility in distribution of hot content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, 2022.

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

[23] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. Outsourcing everyday jobs to thousands of transient functional containers. 2019.

[24] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[25] Daniel James Goodman. Introduction and evaluation of martlet: A scientific workflow language for abstracted parallelisation. In *Proceedings of the 16th international conference on World Wide Web*, pages 983–992, 2007.

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

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

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

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

[30] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262. IEEE, 2018.

[31] David Jackson and Gary Clynch. An investigation of the impact of language runtime on the performance and cost of serverless functions. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 154–160. IEEE, 2018.

[32] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. *ACM SIGARCH Computer Architecture News*, 38(3):314–325, 2010.

[33] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

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

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

[36] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[37] Joanna Kijak, Piotr Martyna, Maciej Pawlik, Bartosz Balis, and Maciej Malawski. Challenges for scheduling scientific workflows on cloud functions. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 460–467. IEEE, 2018.

[38] Youngbin Kim and Jimmy Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455. IEEE, 2018.

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

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

[41] Jörn Kuhlenkamp and Sebastian Werner. Benchmarking faas platforms: Call for community participation. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 189–194. IEEE, 2018.

[42] Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Karim El Tal, and Stefan Tai. An evaluation of faas platforms as a foundation for serverless big data processing. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 1–9, 2019.

[43] Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Dominik Ernst, and Daniel Wenzel. Benchmarking elasticity of faas platforms as a foundation for objective-driven design of serverless applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1576–1585, 2020.

[44] Jörn Kuhlenkamp, Sebastian Werner, and Stefan Tai. The ifs and buts of less is more: a serverless computing reality check. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 154–161. IEEE, 2020.

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

[46] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud*

Computing (CLOUD), pages 442–450. IEEE, 2018.

[47] Valentina Lenarduzzi, Jeremy Daly, Antonio Martini, Sebastiano Panichella, and Damian Andrew Tamburri. Toward a technical debt conceptualization for serverless computing. *IEEE Software*, 38(1):40–47, 2020.

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

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

[50] Ju Long, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. A lightweight design for serverless function as a service. *IEEE Software*, 38(1):75–80, 2020.

[51] Andre Luckow and Shantenu Jha. Performance characterization and modeling of serverless and hpc streaming applications. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5688–5696. IEEE, 2019.

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

[53] Horácio Martins, Filipe Araujo, and Paulo Rupino da Cunha. Benchmarking serverless computing platforms. *Journal of Grid Computing*, pages 1–19, 2020.

[54] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[55] Lisa Muller, Christos Chrysoulas, Nikolaos Pitropakis, and Peter J Barclay. A traffic analysis on serverless computing based on the example of a file upload stream on aws lambda. *Big Data and Cognitive Computing*, 4(4):38, 2020.

[56] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Leveraging serverless computing to improve performance for sequence comparison. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 683–687, 2019.

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

[58] Owen O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008.

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

[60] LI QIAN and HONG JAMES. Dd thousand island scanner (this): Scaling video analysis on aws lambda, 2018.

[61] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021.

[62] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Characterizing and mitigating the i/o scalability challenges for serverless applications. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 74–86. IEEE, 2021.

[63] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Daydream: executing dynamic scientific workflows on serverless platforms with hot starts. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 291–308. IEEE Computer Society, 2022.

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

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

[66] Josep Sampé, Pedro Garcia-Lopez, Marc Sánchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. Toward multicloud access transparency in serverless computing. *IEEE Software*, 38(1):68–74, 2020.

[67] Edans Flavius de O Sandes and Alba Cristina MA de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 1199–1211. IEEE, 2011.

[68] Edans Flavius de O Sandes and Alba Cristina MA de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2012.

[69] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M Hellerstein. A faas file system for serverless computing. *arXiv preprint arXiv:2009.09845*, 2020.

[70] Mohit Sewak and Sachchidanand Singh. Winning in the era of serverless computing and function as a service. In *2018 3rd International Conference for Convergence in Technology (I2CT)*, pages 1–5. IEEE, 2018.

[71] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 281–295, 2020.

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

[73] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. Babelfish: Fusing address translations for containers. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 501–514. IEEE, 2020.

[74] Gaetano Somma, Constantine Ayimba, Paolo Casari, Simon Pietro Romano, and Vincenzo Mancuso. When less is more: Core-restricted container provisioning for serverless computing. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1153–1159. IEEE, 2020.

[75] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Panopticon: A comprehensive benchmarking tool for serverless applications. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 144–151. IEEE, 2020.

[76] Wei Song, Fangfei Chen, Hans-Arno Jacobsen, Xiaoxu Xia, Chunyang Ye, and Xiaoxing Ma. Scientific workflow mining in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2979–2992, 2017.

[77] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10. IEEE, 2020.

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

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

[80] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

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

[82] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Yun Ma, Haoyu Wang, and Xuanzhe Liu. Understanding characteristics of commodity serverless computing platforms. *arXiv preprint arXiv:2012.00992*, 2020.

[83] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. Serverless big data processing using matrix multiplication as example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365. IEEE, 2018.

[84] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Transactional causal consistency for serverless computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 83–97, 2020.

[85] Xiaolong Xu, Wanchun Dou, Xuyun Zhang, and Jinjun Chen. Enreal: An energy-aware resource allocation method for scientific workflow executions in cloud environment. *IEEE transactions on cloud computing*, 4(2):166–179, 2015.

[86] Ryan Yang, Nathan Pemberton, Jichan Chung, Randy H Katz, and Joseph Gonzalez. Pyplover: A system for gpu-enabled serverless instances. Technical report, Technical report, University of California, Berkeley, 2020.

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

[88] Michael Zhang, Chandra Krintz, and Rich Wolski. Stoic: Serverless teleoperable hybrid cloud for machine learning applications on edge device. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2020.

[89] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

[90] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: a programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 328–343, 2020.