Serverless Isn't Server-Less:

Measuring and Exploiting Resource Variability on Cloud FaaS Platforms

Samuel Ginzburg and Michael J. Freedman
Princeton University

Abstract

Serverless computing in the cloud, or functions as a service (FaaS), poses new and unique systems design challenges. Serverless offers improved programmability for customers, yet at the cost of increased design complexity for cloud providers. One such challenge is effective and consistent resource management for serverless platforms, the implications of which we explore in this paper.

In this paper, we conduct one of the first detailed *in situ* measurement studies of performance variability in AWS Lambda. We show that the observed variations in performance are not only significant, but stable enough to exploit.

We then design and evaluate an end-to-end system that takes advantage of this resource variability to exploit the FaaS consumption-based pricing model, in which functions are charged based on their fine-grain execution time rather than actual low-level resource consumption. By using both light-weight resource probing and function execution times to identify attractive servers in serverless platforms, customers of FaaS services can cause their functions to execute on better performing servers and realize a cost savings of up to 13% in the same AWS region.

1 Introduction

Functions as a service (FaaS) is the newest cloud computing paradigm, promising to abstract away the significant complexity of modern software architectures. Unlike deploying virtual machines or containers, developers who use FaaS products such as AWS Lambda only have to write small callbacks without worrying about fault tolerance, resource allocation, scheduling, concurrency, OS abstractions, infrastructure, and often state management.

Serverless platforms offer a highly fine-grained consumption-based billing model that bills customers only for the execution time of applications. Alternative cloud computing platforms such as Amazon EC2 bill customers using an *allocation-based billing model*. Allocation-based pricing schemes charge customers for the provisioned resources, as opposed to the magnitude of the workloads that the customers run using the provisioned resources. In contrast, AWS Lambda employs a *consumption-based billing model* [1]. Consumption-based pricing charges customers for the resources used during the execution of their workloads. Functions created on AWS Lambda still consume resources when not in use, but customers are only charged for the duration of the invoked functions rounded up to the nearest 100 millisecond increment.

This paper explores the key observation that, subtly counter to the notion of consumption-based pricing, users are not actually charged for the precise fine-grained resources they consume (e.g., CPU cycles), but rather just some narrow duration of time during which they execute. We show that not all time slices in serverless environments are equal as a result of performance variation, and that users can (and are incentivized to) exploit these differences for gain.

The challenge of exploiting performance variation emerges from the difference between allocation-based and consumption-based pricing. Under the latter model, because users are not charged for allocation, cloud providers cannot hard allocate resources to their customers. In addition, cloud providers cannot ensure minimal resource limits at allocation time when supporting resource bursting for work-conserving allocation. Rather, because functions are only billed when they execute and yet their execution can be stochastic, cloud providers are incentivized to over-subscribe functions to servers, in order to maximize the resource utilization of their physical machines. As more functions are scheduled onto the same physical machine, server utilization goes up, but so does the probability of resource contention.

Further, the unpredictable nature of invocation patterns in serverless environments implies that resource utilization and contention will differ across physical machines. As a result of increased resource contention, noisy neighbors in serverless environments also have the potential for increased impact on the performance of co-tenant functions. Both of these implications have significant impacts on the potential for significant performance isolation issues in public serverless environments.

Towards this end, this paper explores two main questions.

First, to what extent do performance variances exist across functions on AWS Lambda, and can they be exploited?

Second, can we build an end-to-end system, compatible with today's serverless environments without any platform modifications, that can optimize where to execute user functions to achieve *de facto* lower costs? We call this the *placement gaming* problem for serverless environments.

We explore these questions by conducting a measurement study examining the performance variation of applications in AWS Lambda. We conducted two experiments: one large scale evaluation taking place over the course of a week, and a second smaller evaluation within a duration of 48 hours. We observe that there is *temporal*, *spatial*, and *instantaneous*

Benchmark Name	Measured Resources
Cache Benchmark (cache)	CPU, CPU Cache
FFmpeg Encoding (video)	CPU, CPU Cache, Disk IO
S3 File Download (net)	Network IO
N-Queens (nqueens)	CPU

Table 1. Benchmarks and Measured Resources

performance variation within serverless environments. In particular, we observe that there is significant variation in performance for some compute-bound workloads, and that the performance of these functions is also stable enough to perform placement gaming.

Leveraging these insights, we explore the possibility of placement gaming—an optimization process by which users can reduce the end-to-end cost of running workloads by searching for better performing compute resources. We find that there exist workloads for which placement gaming results in consistent reductions in end-to-end costs of around 10%.

2 Measurements

Performance isolation problems in production cloud environments can be difficult to diagnose properly. The possible causes of performance isolation problems in multi-tenant cloud environments range from noisy neighbors to heterogeneous hardware setups. There are some past works that have observed some performance variation [6, 8], but neither work examined the magnitude or pattern of the observed performance variation.

In this study, we examine both the performance variability and performance stability of a set of benchmarks designed to measure a variety of shared resources. We specifically look for three types of performance variability: *instantaneous*, *temporal*, and *spatial*. We found that all three types of performance variability are present in AWS Lambda, in addition to finding that the end-to-end performance of CPU-bound workloads is stable enough to perform placement gaming.

2.1 Benchmarks

For our benchmarks, we selected a set of test functions, designed to approximate workloads that disproportionately use each measured resource. The benchmarks listed in Table 1 are designed to measure resources where performance isolation has historically been an issue in previous work, such as the CPU cache, CPU (processor frequency and processor pipeline), disk IO, and network IO [2–4, 7].

Cache Benchmark. This benchmark loops over the last level cache twice, writing a set of values and timing how long it takes to read them back after a short interval. It is designed to be representative of cache-heavy workloads.

Video Benchmark. The second benchmark is the ffmpeg video encoding benchmark, which takes in a short video in MP4 format from disk, loads it into memory, and re-encodes the same video file. Video encoding makes heavy use of the CPU and CPU cache in addition to performing some disk IO to read the file into memory.

N-Queens Benchmark. The nqueens problem is a chess problem where the goal is to place N queens on a chess board such that they cannot attack each other. The problem is classically formulated as a backtracking search problem, whose performance primarily varies with the CPU [3]. This benchmark stresses out the CPU without making heavy use of the CPU cache. For the implementation, we used the implementation found internally within the LLVM compiler [5].

Net Benchmark. The S3 file download (net) benchmark is designed to test network performance variance in AWS Lambda. The benchmark consists of a file download from Amazon S3, a persistent object store. We use a 40MB geo-replicated file to ensure locality within AWS regions. Additionally, we control for caching effects in Amazon S3 to ensure that we are only measuring the network performance of the local machine.

2.2 Performance Variance

We explored performance variability across three dimensions for the purpose of conducting placement gaming: *temporal*, *spatial*, and *instantaneous*.

To measure performance variance, we designed two experiments for each of our four benchmarks: first *within* the same AWS region, and second *across* separate AWS regions. Our first intra-regional experiment investigates the viability of performing instantaneous placement gaming as well as temporal placement gaming, while our inter-region experiment investigates the viability of performing spatial placement gaming.

2.2.1 Intra-Region Performance Variance. We first examine the performance variation that can occur in cloud platforms as a result of workload size varying over the course of a day. These workload patterns are also known as *diurnal patterns*, and are primarily caused by an increase in noisy neighbors correlating with when application workloads increase in size. We also explore the existence of instantaneous performance variation by computing the coefficient of variation for each benchmark. We find that significant performance variation exists for each resource both instantaneously, as well as over time.

Methodology. To observe diurnal patterns in serverless workloads, we allocated 50 functions in the `us-east-1` AWS region (Virginia) with 2048MB of memory for each benchmark in Table 1, with each of the 50 functions containing identical code. We know from prior work [8] that it is possible for AWS Lambda functions to end up co-located to each other; however, by allocating 50 function placements for each benchmark, we help ensure that a sufficient portion of them will represent unique function placements. In addition to measuring sufficiently many unique function placements, we

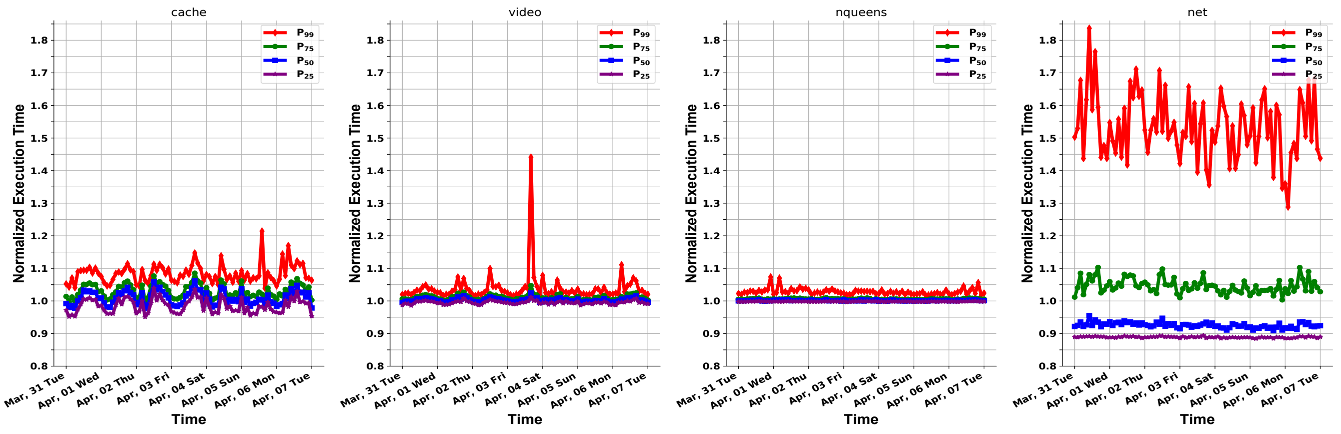

Figure 1. *End-to-End Performance.* The execution time of each benchmark is relative to the week average end-to-end execution time. Diurnal patterns can be seen in the cache and video benchmarks, both of which use the CPU cache heavily.

Benchmark Name	Coefficient of Variation
Cache Benchmark (cache)	5.35
FFmpeg Encoding (video)	2.12
S3 File Download (net)	14.79
N-Queens (nqueens)	1.629

Table 2. Instantaneous Coefficients of Variation

also ensure that each subsequent request to a placed function is executed on the same physical machine (specifically, by validating that the performance statistics reported by *procfs* are monotonically increasing). Each function was executed five times sequentially, for a total of 1000 total function executions every two hours over the course of a week.

Diurnal Patterns. Figure 1 shows strong diurnal patterns for the cache and video benchmarks for all percentiles, while not showing any pattern for the net and nqueens benchmarks. The net benchmark does displays diurnal patterns, however these patterns can not be seen in the figure, as they are significantly smaller (1-2% in magnitude) and only visible when examining the average execution time.

We also observed the magnitude of outlier results (which tended to occur at peak load), finding that significant shifts in performance occur throughout the week occurred for them. The speedup from the worst to best performing function invocations over the course of a week was 29% for the nqueens benchmark, 80% for the cache benchmark, and 75% for the video benchmark. For the net benchmark, the speed between the worst and best performing invocations was 136%.

Instantaneous Performance Variation. We used the coefficient of variation to measure the instantaneous end-to-end performance variation for each benchmark over a one week period. We found that each of the CPU-bound benchmarks maintained a similar coefficient of variation, with values averaging between 1-6% for the CPU-bound benchmarks, similar to past work done on placement gaming for EC2 [3]. However, in Table 2 above, we can see that the net benchmark displayed significantly larger instantaneous variation in end-to-end performance than the CPU-bound benchmarks.

In order to take advantage of instantaneous placement gaming, we also examined the performance stability of each benchmark. Performance stability is necessary to achieve any meaningful benefit from placement gaming, as otherwise changing performance conditions can make placement gaming untenable.

We defined performance stability as the correlation between past and future end-to-end execution time, and determined the correlation between the first and subsequent four invocations of each of our benchmarks. The R^2 between the first and following four function invocations was between 0.8 and 1 for the CPU-bound benchmarks, and 0 for the net benchmark, indicating that CPU-bound workloads have high performance stability, while network-bound workloads display low performance stability. The disparity in performance stability likely comes from the resource sharing mechanisms used. Network contention results in jitter, which is random. Contention for the CPU results in the flushing of microarchitectural state, which is a function of neighboring functions as opposed to being random.

Implications. The results above suggest that temporal placement gaming is not only viable for some workloads, but can result in significant cost reductions for customers who can time-shift their workloads. The existence of instantaneous performance variation as well as performance stability for CPU-bound workloads shows that placement gaming is viable for these workloads, while likely not viable for network-bound workloads.

2.2.2 Inter-Region Performance Variance. We have observed in Section 2.2.1 that there is substantial performance variation within an AWS region at any given time, and that the

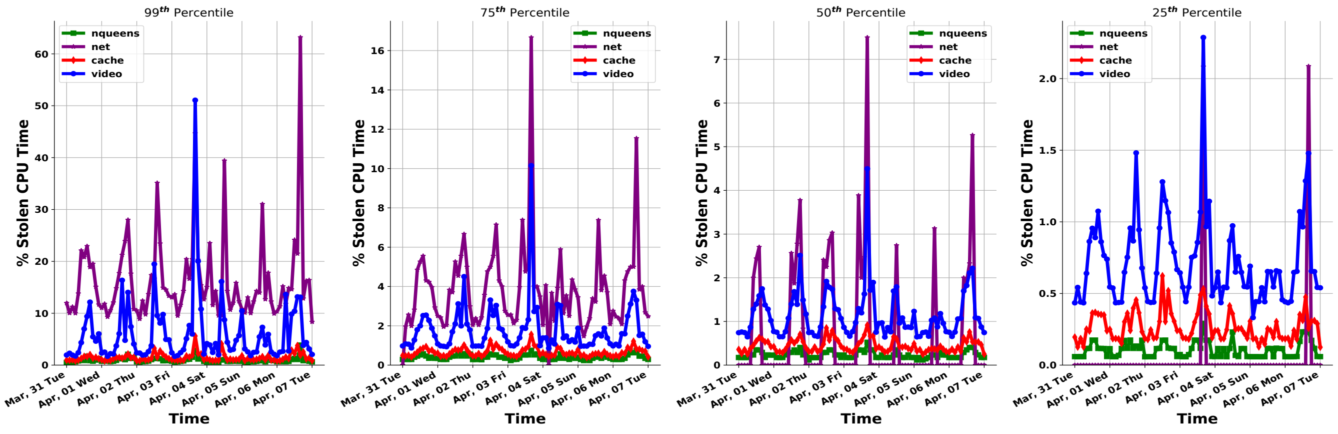

Figure 2. *Percentage of Stolen CPU Ticks Over 1 Week.* We see a visible diurnal pattern in the amount of stolen CPU cycles for each benchmark during the week. The net and video benchmarks report significantly higher fractions of their CPU time as stolen when compared to the other two benchmarks.

performance of each observed resource did change throughout the week, following a diurnal pattern. Since we observed clear diurnal patterns in our end-to-end execution time, we designed a second experiment to investigate *disjoint diurnal patterns*, which are sets of two diurnal patterns where the daily shifts in performance do not overlap. The existence of these patterns can be used to determine the viability of *spatial* placement gaming.

Methodology. For the inter-regional measurements, we invoked a single function instance of the cache and net benchmarks over the course of 48 hours across two geographically distant regions: `ap-northeast-2` (Seoul, South Korea) and the `us-east-1` (Virginia).

Disjoint Diurnal Patterns. Similarly to our intra-region measurements in Section 2.2.1, we were able to see a clear diurnal pattern in the cache benchmark for Virginia, with a comparatively weaker diurnal pattern for Seoul. We observed an average 11% difference in end-to-end performance between the two regions for the cache benchmark, with peaks in performance occurring approximately 12 hours apart. Further, we also observed that Virginia displayed consistently better performance for the net-benchmark.

Implications. The observed differences in function performance across timezones show that it is possible to perform spatial placement gaming for serverless functions in AWS Lambda. To perform such placement gaming, customers should simply run their workload in a region that is currently in a period of local inactivity.

3 Exploiting Performance Variation

In Section 2.2, we found that serverless applications running in AWS Lambda display significant performance variation, and that non network-bound workloads demonstrate predictable performance in the short term.

As a proof of concept, we have designed an end-to-end system to exploit performance variation in AWS Lambda. Temporal, spatial, and instantaneous performance variation all showed potential as avenues for performing placement gaming. While performing temporal and spatial is straightforward and can attain meaningful performance improvements (namely, 80% and 11% respectively), these two approaches introduce significant downsides as well: inter-region data transfer costs and time-sensitive workloads are both factors to be considered when applying these techniques to real applications. Instead we focus on the challenges of performing instantaneous placement gaming, which is the most applicable form of placement gaming for a majority of workloads.

To conduct instantaneous placement gaming, we explore two strategies: *up-front replacement* and *opportunistic replacement*. We base these strategies on prior work in placement gaming for virtual machines [3], and show that with minor modifications, they can be effective in serverless environments as well.

3.1 Up-Front Replacement

Up-front replacement is a straightforward technique: a large set of N functions are allocated up front and invoked a single time to determine the performance of each placed function. The resulting workload is then shifted to the top performing placements and executed until the job is complete.

We implemented two variants of up-front replacement: a black-box technique that works by using function execution time as a proxy for performance, and a grey-box technique that utilizes insights gained from our measurement study. The grey-box variant uses CPU steal time to approximate server load, which is accurate when machines are oversubscribed. We show in Figure 2 that, similarly to end-to-end execution time, CPU steal time follows a diurnal pattern, showing that

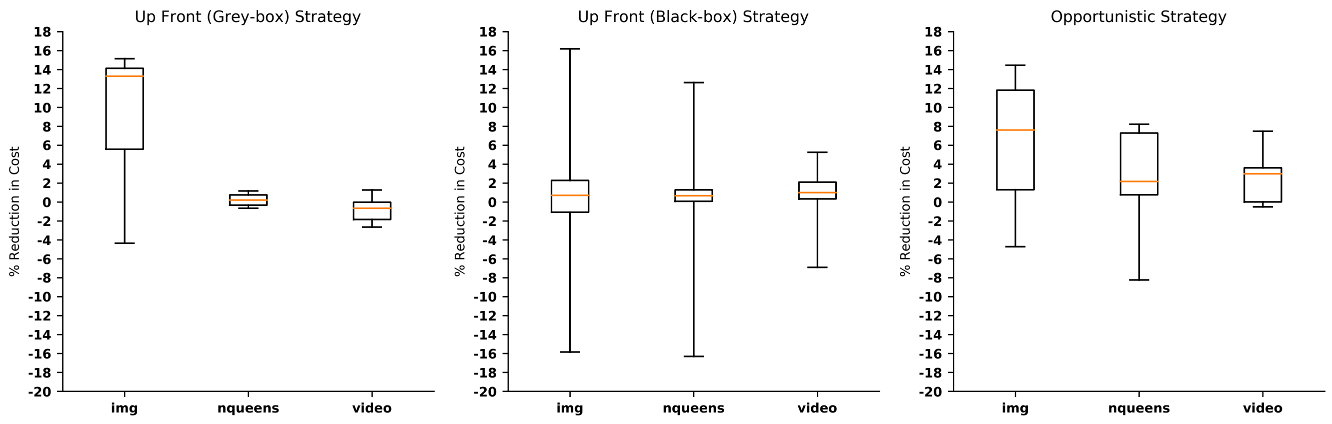

Figure 3. *Placement Gaming Benchmark Results.* Our placement gaming results show that the up-front strategy was only able to reduce costs significantly for a single benchmark, while the opportunistic replacement strategy was able to reduce end-to-end cost for all three of our benchmarks. The box plot for each strategy shows the 25th, 50th, and 75th percentiles of the data, with the whiskers representing the minimum and maximum reduction in cost.

there is a correlation between server load and function performance (with a corresponding R^2 of 0.4 to 0.6, varying diurnally—plot omitted due to space constraints). Using this metric we construct a light-weight probing technique that allows us to sample function placements without paying for the full end-to-end execution time of each benchmark, which also has the added benefit of being accurate independent of inputs to the function.

3.2 Opportunistic Replacement

Unlike up-front replacement, opportunistic replacement takes a more dynamic approach by continuously searching for and replacing functions with subpar performance. The key parameter in opportunistic replacement is the threshold for function replacement, which is the performance cutoff at which a function placement is discarded and a new one allocated. We track the end-to-end performance of all prior function invocations, and replace functions using a percentile cutoff. For our benchmarks, we experimentally determined that the 60th percentile resulted in good performance for this strategy. Replacing functions that drift in performance results in a strategy that is significantly less sensitive to variations in performance over time, making it a better strategy for longer running workloads.

4 System Evaluation

To evaluate our proposed end-to-end system, we decided to target a subset of serverless workloads that are amenable to placement gaming. We found that only a subset of serverless workloads are ideal targets for performing placement gaming.

We found that batch workloads without low-latency requirements worked best, since function placement search costs can increase function latency. In addition, we observed earlier that network-bound workloads (net) do not display significant performance stability, making these workloads a poor target as well. Function chaining (calls to serverless functions from inside a serverless function) can also complicate placement gaming, since chaining functions together results in additional function placements.

With this in mind, we chose to examine three functions: the video benchmark, nqueens benchmark, and a new "img" benchmark, which resizes a JPG image file and then applies a blurring effect to the image. The img and video benchmarks, which both use compute resources that we observed large variation in, were chosen to represent realistic serverless workloads. The difference between these two benchmarks is that the img benchmark has a shorter end-to-end execution time, which can affect placement gaming results. The nqueens benchmark showed little variation, with less than a 10% decrease in performance from the 50th percentile to the 99th percentile. This benchmark was chosen to represent a lower-bound in terms of what results we can get from functions that are entirely compute-bound. All three benchmarks were written in C or Rust to avoid significant runtime overhead.

To evaluate our placement gaming algorithms, we compare our results to a control: a single 2048MB function is allocated, on which we sequentially invoke our workload. Following this, we then sequentially execute the same workload using our selected placement gaming strategy. We use a fixed-size workload of 100 total function invocations. The difference in cost between the control and our placement algorithm is then computed. We then repeat this process 10 times, for a total of 1000 function invocations across three benchmarks. The 10 samples are evenly spread out over the course of a day to account for the possible effects of diurnal patterns.

Overall, for the benchmarks that we evaluated, we found that opportunistic replacement performed slightly better. With opportunistic replacement we were able to reduce costs by between 2% and 8% on average, while up-front replacement was only able to improve costs for the img benchmark. Up-front replacement had the largest reduction in costs overall for the img benchmark, with an average reduction in costs of about 13.5%. We find that strategy choice is workload dependent, varying based on both resource usage and end-to-end execution time.

4.1 Up-Front Strategy Evaluation

In our up-front strategy, we found that both our black-box and grey-box strategies were mostly ineffective. We can see from Figure 3 that the naive black-box strategy performed poorly on all benchmarks, with an average reduction in costs of just slightly above 0%. In contrast, the grey-box variant attained the largest reduction in end-to-end costs of both strategies.

The resources that each benchmark used played a significant role in the efficiency of both placement gaming strategies. The most notable discrepancy was the difference between the video and img benchmarks, which use both the CPU and the CPU cache. The img benchmark was both shorter, and also displayed a larger gap in performance (of around 25%) between the best and worst performing invocations.

4.2 Opportunistic Strategy Evaluation

The opportunistic strategy explored was able to obtain a consistent decrease in costs for all benchmarks. Similarly to the grey-box up-front strategy, the img benchmark had the largest reduction in cost, but the opportunistic strategy was able to do slightly better for the longer running benchmarks. This is because opportunistic replacement constantly searches for new placements, dealing with changes in the performance of function placements better.

4.3 Implications of Instantaneous Placement Gaming

The placement gaming strategies that we developed both require over-allocating functions in order to search for an optimal function placement. There is an inherent trade-off here. Exploiting AWS Lambda's billing model enables us to search for optimal function placements with a minimal cost; however, this results in lower resource utilization and in turn an increased amount of cold starts (increased function startup time) for serverless providers if sufficiently many customers employ placement gaming strategies.

5 Conclusion

AWS Lambda and other FaaS providers have exploded in popularity in recent years, due to their simplified programming and pricing models.

At the core of these platforms, however, is a complex resource allocation problem. Figuring out how to maximize server utilization is important to such platforms. However, the balance between server utilization and strong performance isolation is also important, since the impacts of weak performance isolation are explicitly visible to customers—as opposed to server utilization. In this paper, we show that the lack of performance isolation between tenants in AWS Lambda enables customers to exploit the system.

We investigated a number of performance characteristics of a production serverless platform. For compute-bound workloads, we demonstrated significant *spatial*, *temporal*, and *instantaneous* performance variation. We then investigated the performance stability of each benchmark, and found that non-network resources demonstrated significant performance stability. We further briefly explored the underlying cause of performance variation in AWS Lambda, and found a strong diurnal pattern in CPU steal time that correlated with end-to-end function performance.

We further investigated a client strategy to improve the performance of their use of AWS Lambda, by performing placement gaming on the platform. We demonstrated two effective strategies that do not require in-depth knowledge of the functions being executed, as well as one that utilizes CPU cycle stealing. Using such, we were able to obtain a decrease in end-to-end costs by up to 13% for selected workloads, matching placement gaming results on other platforms, using a fully exogenous approach (i.e., without any support by, modifications to, or inside knowledge of the serverless platform). Following this, we briefly speculate about the possible impacts of placement gaming on resource utilization. The problem of balancing strong performance isolation with the maximization of resource utilization in serverless platforms remains an important open problem.

References

[1] AWS Lambda pricing. https://aws.amazon.com/lambda/pricing/, 2020.

[2] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proc. First Annual ACM SIGMM Conference on Multimedia Systems*, pages 35–46, 2010.

[3] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proc. Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[4] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, and C. Kim. EyeQ: Practical network performance isolation for the multi-tenant cloud. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[5] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 75–86, 2004.

[6] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proc. of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 131–141, 2020.

[7] M. Shahrad, J. Balkind, and D. Wentzlaff. Architectural implications of function-as-a-service computing. In *Proc. 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075, 2019.

[8] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proc. USENIX Annual Technical Conference*, pages 133–146, 2018.