Faster and Cheaper Serverless Computing on Harvested Resources

Yanqi Zhang
Cornell University
yz2297@cornell.edu

Íñigo Goiri
Microsoft Research
inigog@microsoft.com

Gohar Irfan Chaudhry
Microsoft Research
gochaudh@microsoft.com

Rodrigo Fonseca
Microsoft Research
Fonseca.Rodrigo@microsoft.com

Sameh Elnikety
Microsoft Research
samehe@microsoft.com

Christina Delimitrou
Cornell University
delimitrou@cornell.edu

Ricardo Bianchini
Microsoft Research
ricardob@microsoft.com

ABSTRACT

Serverless computing is becoming increasingly popular due to its ease of programming, fast elasticity, and fine-grained billing. However, the serverless provider still needs to provision, manage, and pay the IaaS provider for the virtual machines (VMs) hosting its platform. This ties the cost of the serverless platform to the cost of the underlying VMs. One way to significantly reduce cost is to use spare resources, which cloud providers rent at a massive discount. Harvest VMs offer such cheap resources: they grow and shrink to harvest all the unallocated CPU cores in their host servers, but may be evicted to make room for more expensive VMs. Thus, using Harvest VMs to run the serverless platform comes with two main challenges that must be carefully managed: *VM evictions* and *dynamically varying resources in each VM*.

In this work, we explore the challenges and benefits of hosting serverless (Function as a Service or simply FaaS) platforms on Harvest VMs. We characterize the serverless workloads and Harvest VMs of Microsoft Azure, and design a serverless load balancer that is aware of evictions and resource variations in Harvest VMs. We modify OpenWhisk, a widely-used open-source serverless platform, to monitor harvested resources and balance the load accordingly, and evaluate it experimentally. Our results show that adopting harvested resources improves efficiency and reduces cost. Under the same cost budget, running serverless platforms on harvested resources achieves 2.2× to 9.0× higher throughput compared to using dedicated resources. When using the same amount of resources, running serverless platforms on harvested resources achieves 48% to 89% cost savings with lower latency due to better load balancing.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Distributed computing methodologies**; **Planning and scheduling**.

KEYWORDS

Serverless computing, harvested resources

1 INTRODUCTION

Serverless computing is becoming an increasingly popular cloud programming paradigm, especially in the form of Functions as a Service (FaaS), with offerings from several commercial providers [3, 23, 39]. These FaaS platforms offer intuitive event-based interfaces for application development. The interface obviates the need for users to explicitly configure resources, such as the number and size of virtual machines (VMs) or containers to run the functions. FaaS is also cheaper for users, as they only pay for the exact amount of resources they use during function execution. This is in contrast to Infrastructure as a Service (IaaS), where users pay for long-term reserved resources in the form of VMs. FaaS is an ideal candidate for applications with high data-level parallelism and/or intermittent activity (*e.g.*, online sites that are driven by fluctuating user load). However, the *serverless provider* still needs

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483580

to provision, manage, and pay the *IaaS provider* for the VMs hosting its platform. This ties the cost of serverless to the cost of the underlying VMs. Worse, the serverless provider must pre-provision a large amount of VM capacity to provide fast elasticity and the illusion of infinite resources, while the FaaS users pay only for the resources their functions actually use.

Harvested resources. Fortunately, IaaS providers offer their surplus resources as VMs at a much lower price (and relaxed guarantees), such as Spot [6, 10] and Burstable VMs [7, 8]. Along similar lines, Harvest VMs [4] are an even cheaper and more efficient alternative. Each Harvest VM is evictable and has a minimum size, but it grows by harvesting unallocated CPU cores in its host server beyond this minimum. When a new "regular" (non-evictable) VM is placed on the server, the Harvest VM shrinks. The IaaS provider only evicts Harvest VMs when their minimum size is needed for a regular VM.

Serverless functions, which are mostly single-threaded and short-running [53], are a natural fit for running on harvested resources. Despite their low cost, Harvest VMs introduce two challenges: workloads can be evicted, and VMs have dynamic variations in terms of compute and/or memory resources. Not only do Harvest VMs have the potential to reduce the cost of hosting FaaS platforms, but they can also provide better performance at the same cost.

Our work. This paper tackles the challenges of running serverless platforms on Harvest VMs. To understand the impact of evictions and of the variability in harvested resources on a FaaS platform, we first characterize both a FaaS offering (Azure Functions) and the resources available to Harvest VMs using production traces from Azure. We contrast the duration of function executions with the lifetime of Harvest VMs and the durations over which resources are available for harvesting. Our characterization suggests a good match between FaaS platforms and Harvest VMs. Thus, we next study how to adapt a FaaS platform to run on harvested resources.

To address Harvest VM evictions, we explore the space of regular and Harvest VMs mixes, for short- and long-running functions, and quantify the trade-off between cost and reliability. Using detailed simulations combining FaaS and Harvest VM traces, we find that when running FaaS solely on Harvest VMs, evictions cause at most 0.0015% of invocations to fail.

To make this practical, we must address resource variations inherent to Harvest VMs. To this end, we design and implement a load balancer for FaaS platforms that places functions in VMs according to the availability of harvested resources. Our load balancer reduces resource contention while keeping the function cold start rate low.

Our implementation modifies OpenWhisk [45], a widely-used open-source FaaS platform, to monitor the availability of harvested resources and balance the load accordingly. Our experimental results demonstrate the performance improvement over the existing OpenWhisk load balancer and other widely used policies, achieving 22.6× throughput than vanilla OpenWhisk. We finally demonstrate the performance improvement and cost savings of serverless computing on Harvest VMs, compared to regular and Spot VMs. Under the same cost budget, serverless platforms hosted on Harvest VMs are able to achieve 2.2× to 9.0× throughput than regular VMs. When provisioned with the same amount of resources, serverless platforms hosted on Harvest VMs are 45% to 89% cheaper than regular VMs and 0% to 44% cheaper than Spot VMs.

2 BACKGROUND AND RELATED WORK

Serverless and FaaS. Serverless computing, especially Functions as a Service (FaaS), is gaining popularity as the way to deploy applications on the cloud [52]. The FaaS programming model offers simplicity of just uploading application code without having to manage resources or configurations. In the FaaS platform we study, functions are logically grouped to form applications and the application is the unit of scheduling and resource allocation. The platform provides elasticity by automatically scaling up resources with increasing load and scaling down to zero during idle periods. The user only gets billed for the resources consumed during function executions. All these properties make FaaS a compelling option for programming the cloud from the user's perspective.

The serverless provider faces the challenge of ensuring high performance while minimizing cost. To provide the illusion of always-on and infinitely scalable resources to the user, the provider needs to have the resources ready whenever a function is invoked. Shahrad *et al.* [53] show that 50% of functions execute for less than 1s and about 90% execute for less than 10s on average. A function can start quickly when the code is already in memory (*warm start*) and does not have to be brought in from persistent storage (*cold start*). Since these function executions are generally short lived, cold starts can dominate the overall execution time if the resources are not available at invocation time. To mitigate this, providers typically set a keep-alive threshold for which the function container is kept available after the invocation completes in anticipation of an upcoming invocation to the same function.

There has been a wealth of research on serverless computing, both to expand the set of applications that can use the model, and to improve the serverless infrastructure. Broadly, it spans: (a) scheduling policies for making serverless platforms cost-effective and performant [28, 53]; (b) performance-aware and cost-effective storage [33, 34, 43, 51]; (c) secure and lightweight container infrastructure [1, 2, 42, 44, 54, 56, 59]; (d) characterization of existing serverless workloads [53]; and (e) enabling applications to run in a serverless-native manner, including data processing and analytics [26, 49], video processing [20], ML training [14], DNA sequence visualization [36] and compilation [19]. We show that mindfully using cheaper resources without performance/reliability degradation is the right way to minimize the hosting cost of FaaS.

Harvest VMs. Harvest VMs were proposed in [4]. Users select and deploy them as they do any other VM. Each Harvest VM is defined by its minimum size (in terms of physical CPU cores, memory, disk space, and network bandwidth) and how many harvested physical cores they are capable of using. While the number of physical cores assigned to a Harvest VM may change dynamically, the other resources do not. The workload running on the Harvest VM can query the number of physical cores assigned to it in */proc* in Linux and the registry in Windows. The Harvest VM receives a 30-second notice before an eviction happens. These mechanisms allow the workload to take appropriate actions.

Users pay for the minimum size at a heavy discount, like those for Spot VMs, compared to regular VMs. For example, Spot VMs are 48% to 88% cheaper than the same size regular VMs in Azure [9]. The additional harvested cores are even cheaper because they vary over time. The total cost for the users is the sum of the minimum cost and the harvested one.

Despite offering a large amount of resources at low price, evictions and resource variation can impact the system reliability and performance [4]. This paper addresses those issues.

Cluster scheduling and load balancing. A large body of work [12, 16, 18, 21, 22, 25, 27, 30, 47, 48, 55] has focused on cluster scheduling frameworks, such as Kubernetes [35] and Apache YARN [57]. However, these works assumed that the underlying resources (VMs or bare-metal servers) are constant over time.

In contrast, Harvest VMs may experience significant variation in their number of cores over their lifetime. Ambati *et al.* did adapt YARN to run on Harvest VMs [4]. However, the batch and Big Data analytics workloads common of YARN deployments are quite different than those of FaaS platforms [50, 53]. For example, function executions are typically substantially shorter than data analytics tasks, so FaaS workloads can more easily adjust to the frequent changes in the numbers of cores. On the other hand, each function typically consumes fewer resources (e.g., memory) than an analytics task, meaning that many of them can be packed on the same VM so an eviction may affect more computations.

3 CHARACTERIZATION

While previous work has studied some production characteristics of Harvest VMs [4] and FaaS workloads [53], in this section we take a closer look with the goal of understanding how they might interact. We are interested in the impact of Harvest VM evictions and core variations on function executions. In particular, we look at the distribution of Harvest VM lifetimes and the distribution of intervals between Harvest VM core changes. Before each eviction, the Harvest VM receives a 30-second grace period, which can be used to stop sending new invocations to the VM, and to finish ongoing function executions. Invocations that last longer than 30 seconds are at risk of being killed, and below we pay particular attention to these long invocations. Compared to the previous characterization of FaaS workloads [53], we feature a new analysis addressing the issues involved in hosting FaaS on Harvest VMs: the impact of VM evictions and the capacity needed to host the FaaS workloads.

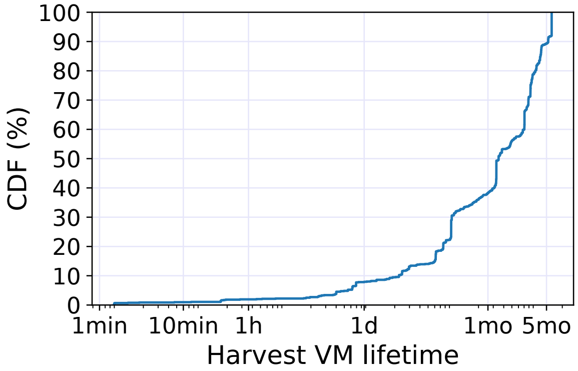


Figure 1: Distribution of the Harvest VM lifetime [4].

3.1 Harvest VMs

Evictions. To study Harvest VM evictions, we use a trace of the private cluster described in [4]. The trace includes 1075 Harvest VM instances deployed between October 8th 2019 and March 28th 2020. We include both evicted and not evicted Harvest VMs, and remove from the VM lifetime the 10 minutes required to install the FaaS platform and dependencies. Despite this overhead, 96.7% of all Harvest VMs are suitable for hosting FaaS. Figure 1 shows the lifetime distribution of these Harvest VMs. The average lifetime is 61.5 days, with more than 90% of Harvest VMs living longer than 1 day. More than 60% survive longer than 1 month.

Resource variability. To study the resource variation patterns of Harvest VM, we look at a smaller and more detailed trace of 37 Harvest VMs running in Azure production clusters between January 1st and February 24th 2021. To match the memory size of the smallest Harvest VM (*i.e.*, 16 GB), the maximum CPUs of each Harvest VM is limited to 32. Figure 2 shows the distribution of intervals between changes in Harvest VM CPUs. The expected interval is 17.8 hours, with around 70% of them being longer than 10 minutes, and around 35% longer than 1 hour. 62.2% of the studied Harvest VMs experienced at least one CPU shrinkage and 54.1% experienced at least one CPU expansion. 35.1% VMs never experienced any CPU changes.

Figure 3 shows a histogram of individual CPU changes for the studied Harvest VMs. Positive numbers represent expansions and negative numbers represent shrinkage. The

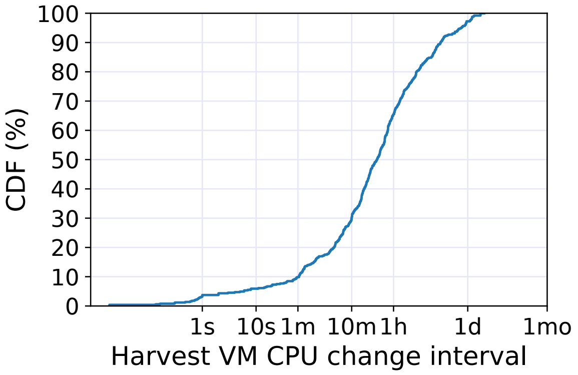


Figure 2: Intervals between Harvest VM CPU changes.

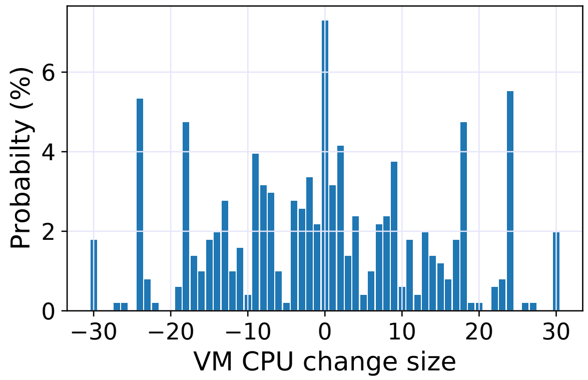


Figure 3: Distribution of Harvest VM CPU change sizes and correlation of change sizes and change interval.

Trace	F_{Large}	F_{Small}
Duration Data	Percentiles	Start/End Times
Granularity	Per App	Per Invocation
Dates	2021-01-31	2021-01-31 to 2021-02-13
#Apps	20,809	119
Invocations	910M	2.2M

Table 1: Details on the two FaaS traces used in the paper.

points at 0 represent the VMs that did not change during the period covered by the traces. The distribution tends to be symmetric with most of CPU changes falling within 20 CPUs. The average and maximum CPU change size are 12 and 30 for both shrinkage and expansion. Considering the maximum CPUs of the profiled Harvest VMs is 32, the size of the changes has a significant impact on instantaneous capacity of Harvest VMs. We did not find a significant correlation between the size of the change and the change interval.

3.2 Serverless Functions

We now study the duration of function invocations. We obtained two traces (Table 1) of invocations from Azure Functions: F_{Large} is a coarse 1-day trace with invocation duration percentiles for a subset of a cloud region, and F_{Small} is a detailed trace of a small cluster with precise invocation timings. We look at the overall trends with F_{Large}, and use F_{Small} for deeper analysis, including trace-driven simulations.

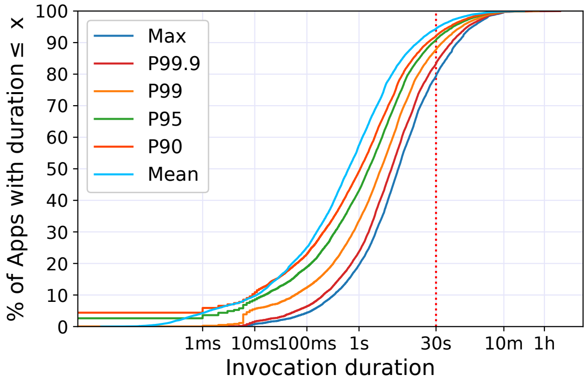


Figure 4: CDFs of the average and top percentiles of the invocation durations per application in the F_{Large} trace.

Duration per application. Figure 4 shows the distribution of maximum invocation durations per application from the F_{Large} trace, as well as those of the mean and other duration percentiles. The invocations are generally short. The graph shows the 30-second grace period of Harvest VM eviction. Invocations shorter than this are safe from evictions, while longer invocations could be terminated. 20.6% of the applications have at least one invocation (maximum) longer than 30 seconds. We refer to these applications as "long" applications. 16.7% and 12.3% of applications have 99.9^{th} and 99^{th} percentile durations longer than 30 seconds, respectively.

Figure 5 compares the same distributions between the two traces. The traces are similar with respect to the tails of the per-application invocation durations, with the applications in the F_{Small} trace having higher fractions of longer invocations. This is acceptable for our purposes, as it makes our analysis more pessimistic. We base our analysis in the remainder of the paper on the F_{Small} trace.

Durations per invocation. The F_{Small} trace allows us to look at the duration of every invocation. Figure 6 shows the latency distribution of all considered invocations. The vast majority are short, with more than 85% of invocations shorter than 1 second, and 96% of the invocations shorter than 30s. The longest recorded invocation is 578.6 seconds.

Long applications. In terms of sensitivity to Harvest VM evictions, only 4.1% of the invocations are 'long', but these long invocations take over 82.0% of the total execution time of all invocations. At the granularity of application, 58 applications (48.7% of all) are long applications. These long applications take up 67.5% of all invocations and 99.68% of the total invocation time. These long invocations (and applications) are vulnerable for evictions if placed on a Harvest VM. As we see in §4, naïvely allocating the long applications

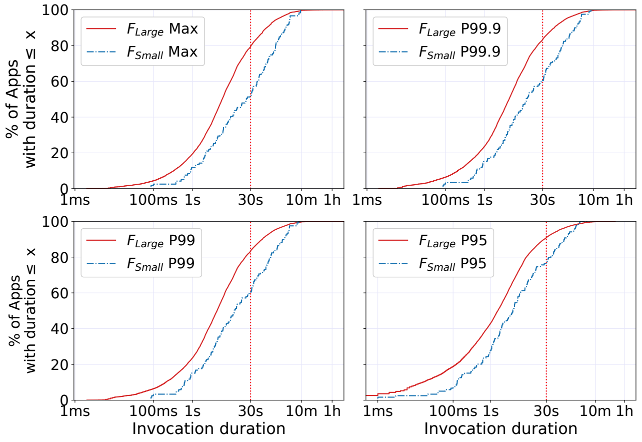


Figure 5: Invocation durations per app for F_{Large} and F_{Small}.

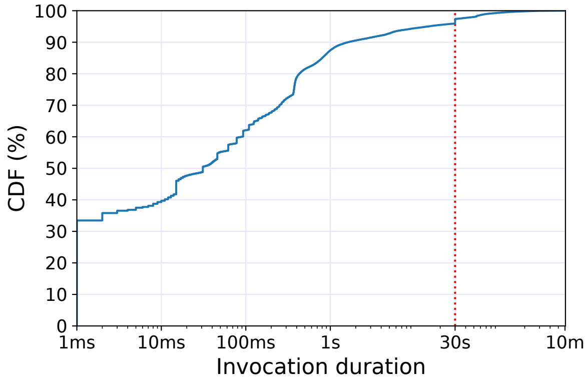


Figure 6: Durations of all invocations in the F_{Small} trace.

to regular VMs, and running the others on Harvest VMs may be too conservative a strategy, with very modest gains.

Looking closer, Figure 7 shows the duration distribution of the long applications, where each point on x-axis corresponds to one application, and the error bar shows the standard deviation of the durations. There are big gaps between the max and mean duration of long applications, especially for applications with max duration longer than 100 seconds, indicating that long applications fall under this category mainly due to a small fraction of invocations in the tail of their duration distribution. We use this to our advantage in the next section.

3.3 Implications

Combining the characteristics of Harvest VMs (Section 3.1) and serverless workloads (Section 3.2) shows that, intuitively, FaaS workloads are a good fit for Harvest VMs. The short duration of the majority of the invocations (only 4.1% are longer than 30 seconds) and the relatively much longer Harvest VM lifetime (more than 90% of Harvest VMs live longer than 1 day) make serverless workloads unlikely to be affected by Harvest VM evictions. Based on this intuition, in Section 4 we use trace-drive simulations to more precisely characterize the reliability of serverless compute on Harvest VMs.

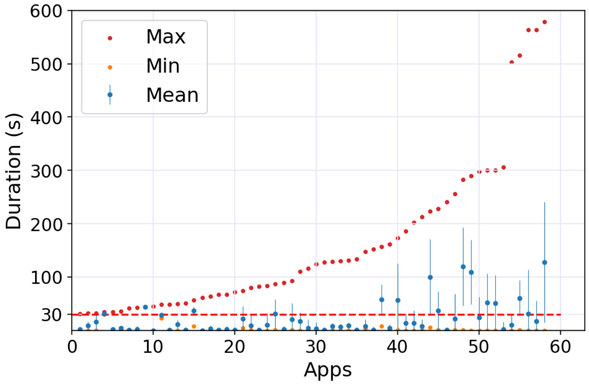


Figure 7: Durations of long applications invocations.

Resource variation on Harvest VMs is much more common than evictions, but compared to the short duration of most invocations, the number of CPUs of Harvest VMs can be considered relatively stable: 70% of CPU change intervals are longer than the longest invocation in the studied serverless workload trace (578.6 seconds). However, because of the frequency and magnitude of resource changes (Figure 3), Harvest VM-aware load balancing is essential to guarantee system performance. In addition to this, even if mostly stable, Harvest VMs tend to be more heterogeneous than regular VMs, reinforcing the importance of proper load balancing.

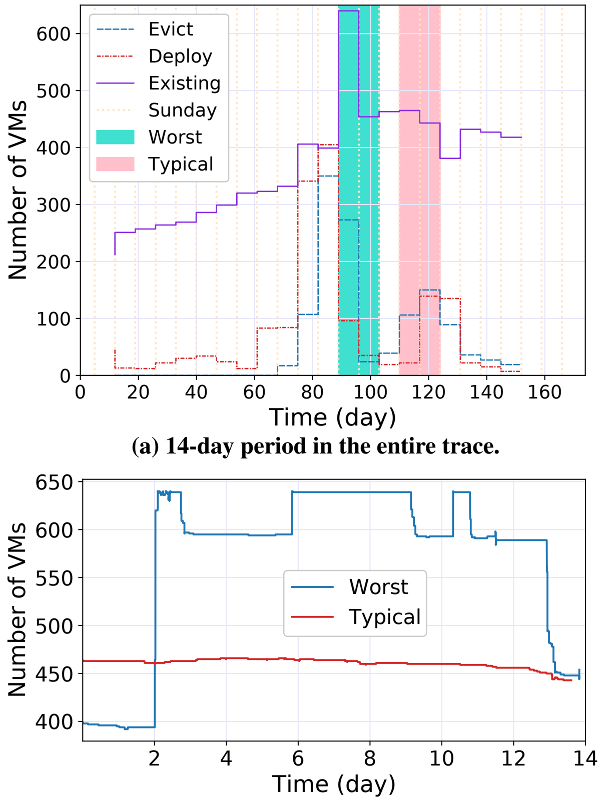
4 HANDLING EVICTIONS

In this section, we study the impact of Harvest VM evictions when running serverless workloads. When an eviction occurs, any function running at the time fails. *What is the best strategy to eliminate or minimize these failures?*

4.1 Methodology

While the comparison of the distributions in the previous section provides bounds on the failure rates, the interaction of evictions and long executions is not trivial, and we resort to trace-driven simulations to answer this question.

We used the Harvest VM trace from Figure 1 and the F_{Small} functions trace (§3). Since the Harvest VM trace (173 days) is longer than the serverless workload trace (14 days), we select a 14-day period from the Harvest VM trace which aligns with the serverless workload trace. Figure 8a shows, for the 14-day period starting at each Sunday (dotted vertical lines), the total number of VMs, and the number of VM creations and evictions. We use the Harvest VM eviction rate defined as number of VM evictions over the number of existing VMs, as the metric to categorize the Harvest VM trace periods. The average eviction rate of all 14-day periods is 13.1%. We select two periods: (1) one with the max VM eviction (86.4%), as



(a) 14-day period in the entire trace.

(b) Selected 14-day periods for simulation.

Figure 8: Harvest VM creations and eviction patterns.

worst case, and (2) one with an eviction rate close to average (8.4%), as the typical case. Starting days of the worst and typical cases are marked as *Worst* and *Typical* in Figure 8a.

We simulate the serverless framework as a global pool of containers; an invocation is able to use any existing container of the same application and randomly chooses one if there are multiple candidates. Invocations have a keep-alive time set to 10 minutes (the default in OpenWhisk [45]). A container is removed if it does not execute any invocations for the entire keep-alive period. Each container is randomly allocated to VM that has not been warned of eviction. The number of concurrent invocations that each container can host is set to 1. Since our traces do not record CPU usage, we assume that the CPU usage of all applications is identical.

For Harvest VMs, when we receive the 30-second eviction warning for a VM, the load balancer stops sending new invocations to it. Pending invocations continue to execute on the VM and fail if they do not complete before the VM eviction. In the event that resources start to decline below a pre-configured threshold, it spins up additional VMs.

For each 14-day Harvest VM trace snippet, we run the simulation 1000 times and show the aggregated results. Our simulation models the key components of serverless frameworks, including container pool and keep-alive. It can model potential future workload changes by simply acquiring new traces, assuming no changes to the serverless framework.

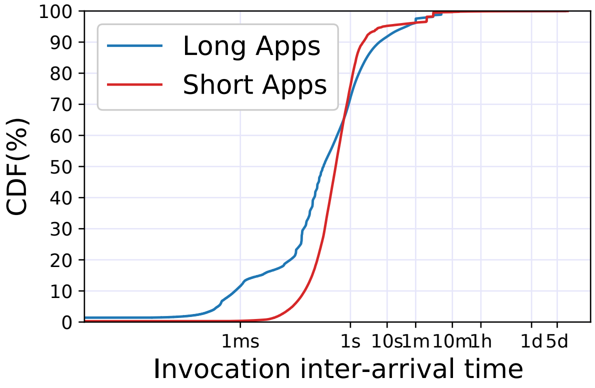


Figure 9: Inter-arrival times for short vs. long apps.

4.2 Combining Regular and Harvest VMs

Strategy 1: No failures. We start with the most conservative provisioning where all long applications (*i.e.*, those with at least one invocation longer than 30 seconds) are allocated in regular VMs and the rest in Harvest VMs. This guarantees that no invocation longer than 30s will run on Harvest VMs, but is the least efficient provisioning strategy. Section 3.2 showed that long applications take up to 67.5% of all invocations but 99.7% of the invocation time. However, we also need to account for the keep-alive period to prevent cold starts. We ran a simpler version of our simulation here to estimate the computation capacity taken by the two application types, while accounting for their arrival times and keep-alive behavior.

For 10-minute keep-alive, the simulation shows that only 12.0% of computation capacity can be hosted by low-cost Harvest VMs. While this is much higher than the fraction of execution time for short applications (0.32%), it is significantly lower than the fraction of invocations that corresponds to short applications (32.5%). This is due to their shorter invocation times on average, and to their inter-arrival times. Figure 9 shows that a larger fraction of the inter-arrival times for short applications is below 10s, and multiple close invocations reduce the wasted idle time due to keep-alive. We verified that these results do not change significantly for different keep-alive periods ranging from 1 minute to 24 hours. Ultimately, this strategy is too conservative, and 94% of the invocations that run on the regular VMs are still short.

Strategy 2: Bounded failures. Given the high operational cost of Strategy 1, we study a relaxation of the bound on failures caused by eviction. If we are willing to tolerate a small

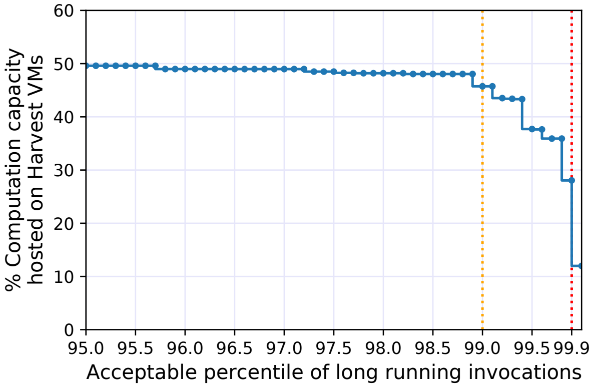


Figure 10: Fraction of Harvest VM capacity versus acceptable percentile of per-app long invocations.

fraction of eviction failures, we can allocate more applications to Harvest VMs, and trade reliability for efficiency.

We can provide an upper bound $(100-x)\%$ (say, 1%) on the per-application eviction failure rate by allocating to regular VMs applications with the x^{th} (say, 99^{th}) percentile duration longer than 30s, instead of the maximum. In effect, some long applications from Strategy 1 are allocated to Harvest VMs in this strategy, but only those where $(100-x)\%$ of the invocations are longer than 30s.

To characterize the trade-off between reliability and efficiency of the policy, we perform the same trace-driven simulation as in §4.2, and sweep the percentile x from 95 to 99.9, with increments of 0.1. Figure 10 shows the results, with the decision percentile in the x-axis, and the resulting fraction of computing capacity used by Harvest VMs.

In summary, bounding the failure rate to less than 0.1% allows 28% of computation to be hosted by Harvest VMs. A rate lower than 1% allows 45.7% of computation to be hosted by regular VMs. Although efficiency improves compared to Strategy 1, it is still pessimistic, as most invocations that run in regular VMs are still short, and even the long invocations would only fail if they run in a Harvest VM and start less than 30s before an eviction.

4.3 Running on Harvest VMs

Strategy 3: Live and Let Die. We next examine running a full serverless workload solely on Harvest VMs. We ran the full simulation described in §4.1. For the *Worst* period in the Harvest VM trace (*i.e.*, max VM eviction rate), the average invocation failure rate is 0.0015% (99.9985% success rate). The *Typical* period has a failure rate of 3.68×10^{-8} (*i.e.*, "7 nines" of reliability).

Intuitively, failures caused by VM evictions are rare because they require two low-probability events to happen simultaneously: a Harvest VM gets evicted *while* it is running a long invocation. VM evictions are also correlated and frequently happen in bursts, with a large number of VMs evicted within a few seconds, as shown in Figure 8b.

Cold starts are also minimal when the workload runs on Harvest VMs. The average simulated cold rate is 1.1967% in the Typical period, and 1.1981% in the Worst period, increasing by 0.0084% and 0.1254% compared to regular VMs.

4.4 VM Migration/Snapshotting

An alternative, or even complementary approach, to increase the reliability of the serverless framework hosted on Harvest VMs is to use VM live migration [15] or snapshot/restore [17, 56]. The idea is to run serverless applications in nested VMs hosted by Harvest VMs, and migrate the nested VMs that correspond to long invocations when the Harvest VM is warned of eviction. The main metric, however, is not the downtime of the application, but the total time for which the source VM must be available. Because of the low invocation failure rate from Strategy 3, we leave using VM migration to improve system reliability as future work.

4.5 Conclusion

When running solely on Harvest VMs, the failures caused by VM evictions are rare while fully utilizing the cheap harvested resources. This is caused by the low joint probability of a rare long-running execution during a Harvest VM eviction. As a result, in the rest of the paper, we assume all applications are hosted on Harvest VMs.

5 HANDLING RESOURCE VARIABILITY

In this section, we develop a resource variation-aware load balancing policy for serverless frameworks on harvested resources. We start with the well-known algorithm *join-the-shortest-queue (JSQ)* [24], which aims to minimize the resource contention caused by CPU variation. Based on that, we then present our *min-worker-set (MWS)* algorithm, which aims to reduce the container cold start rate for serverless workloads while reducing resource contention.

5.1 Join-the-Shortest-Queue (JSQ)

JSQ is a CPU-aware load balancing algorithm. The load balancer monitors the compute load of each backend VM and allocates an invocation to the VM that has the least amount of pending work. This effectively reduces queueing time and resource contention, leading to shorter end-to-end latencies.

Since the ground truth of pending compute work is unknown in advance, we approximate it with a weighted sum of CPU and memory utilization $w_c \frac{cpu_{used}}{cpu_{avail}} + w_m \frac{mem_{used}}{mem_{avail}}$, with

$w_c > w_m$ to reflect the scarcity of allocated CPUs. We show that the weighted utilization of CPU and memory is a better usage metric than the number of pending invocations (queue length) at an invoker, or the sum of expected resource usage of pending invocations (weighted queue length). This is because queue length does not account for varying function resource needs, and weighted queue length can deviate from the ground truth, due to insufficient samples and different function inputs. The utilization metric also captures the variation of allocated CPUs of Harvest VMs, and avoids starvation by stopping assigning invocations to VMs that suffer from excessive resource shrinkage. In terms of overhead, the complexity of each scheduling operation is $O(N)$, where N is the number of backend VMs in the system. The scheduling overhead can be reduced by randomly sampling a subset of d backend VMs and choosing the least loaded one [13, 47, 58], although at the expense of scheduling quality.

5.2 Min-Worker-Set (MWS)

In serverless computing, the end-to-end latency of an invocation includes cold start time, queueing time and execution time. Although JSQ can reduce queueing time by preventing long queues and execution time by alleviating resource contention, it can potentially increase the cold start rate and harm the end-to-end latency. Assuming that a function has a Poisson arrival process with arrival rate λ, and the serverless platform has N backend VMs, JSQ will distribute the invocations across all N backend VMs. The resulting invocation arrival rate on each backend VM will be $\frac{\lambda}{N}$. In a large system (*i.e.*, large N), the expected inter-arrival time $\frac{N}{\lambda}$ is more likely to be larger than the container keep-alive time of serverless platform, increasing the chance of cold starts.

We design the MWS algorithm to jointly reduce queueing time, execution time, and cold starts. This is inspired by the intuition that, in the common case, where the compute resources of the system are not overloaded, slight imbalance of invocation assignment among invokers is unlikely to cause resource contention and queueing leading to increased latency. MWS consolidates each function to a minimal set of k backend VMs that have adequate resources to accommodate all invocations of the function. The invocation arrival rate on individual backend VMs becomes $\frac{\lambda}{k}$. With $k \ll N$, the invocation inter-arrival time in MWS is much shorter than JSQ. Thus, it is very likely to be shorter than the container keep-alive time, enabling warm starts. The sketch of MWS is shown in Algorithm 1. For each function f, the load balancer assigns it a home VM as the beginning of the search process, and estimates its resource usage u_f as the product of requests per second (RPS), expected resource usage, and expected duration. The load balancer keeps adding new VMs to the worker set s until the total usable resources r of all VMs in the set s exceeds the estimated usage of the function u_f. Finally, the load balancer picks the least loaded VM in the worker set s to execute the invocation, where the load is defined as the weighted sum of CPU and memory utilization as in JSQ.

Algorithm 1: Min-worker-set (MWS) algorithm

```
Input: Function $f$;
Function: Expectation $E$; Consistent hashing $CH$;
Variable: Requests per second $RPS_f$;
Variable: CPU usage $CPU_f$;
Variable: Invocation latency $lat_f$;
$u_f = RPS_f \cdot E(CPU_f) \cdot E(lat_f)$;
$r = 0, s = \emptyset$;
$VM = CH(f)$;
while $r < u_f$ do
    $r = r +$ usable_resources($VM$);
    $s = s \cup VM$;
    $VM =$ next($VM$);
end
return $argmin_{VM}\{\text{load}(VM) \mid VM \in s\}$
```

In the common case that the system is not overloaded, MWS is more scalable than JSQ, with minimum scheduling overhead. For each invocation, the controller only needs to search for the least loaded invoker in the worker set of the function (*i.e.*, usually a small number) rather than searching among all invokers. In the worst case that the system is running at full utilization, the scheduling overhead increases with the load of the system and converges to JSQ as MWS spans all backend VMs. Compared to JSQ, MWS can also reduce the number of functions allocated to each VM, thus reducing the storage space occupied by function images.

Dealing with VM evictions. Harvest VM evictions can be detrimental to the performance of the MWS algorithm, because VM failure and redeployment lead to variation in the number of VMs in the system, and thus reshuffling of home VMs for all functions, making cold starts dominant. To minimize the number of functions that need to be reshuffled and thus minimize cold starts, we use consistent hashing. Thus, whenever the number of VMs changes, home VMs are only reshuffled for a minimal number of functions.

In consistent hashing [31], all VMs in the system are assigned a hash ID within $[0, I]$ where I is much larger than the number of VMs in the system, so that VMs are uniformly distributed in the ID space. Conceptually, all VMs in the system are organized into a ring with VM IDs increasing clockwise, except VM I, whose next VM in the ring is VM 0. Functions are mapped to and uniformly distributed in the same ID space $[0, I]$ and are assigned next VM in clockwise direction (to the ID of the function) as home VM. As the VM IDs are uniformly distributed, the expected number of functions assigned

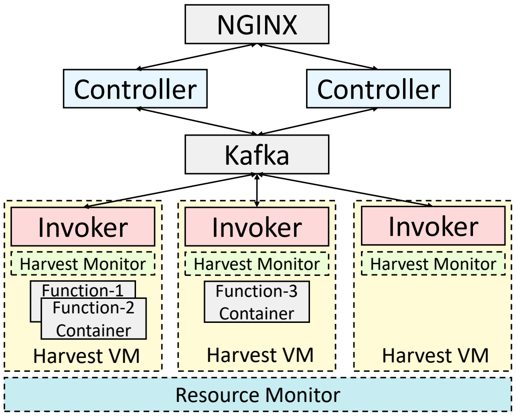


Figure 11: Architecture of our resource-variation-aware load balancing solution on OpenWhisk. The dotted lines show our modifications and components not present in vanilla OpenWhisk.

to each VM are identical. When an existing VM crashes or a new VM joins the system, only functions originally assigned to the crashed VM or the new VM are reshuffled.

6 IMPLEMENTATION

We implement our proposed resource-variation-aware load balancing scheme on OpenWhisk [45], a popular open source serverless platform developed by IBM. In this section we first describe the architecture of OpenWhisk and then the changes we made for Harvest VM-aware load balancing.

6.1 OpenWhisk Architecture

Figure 11 shows the architecture of OpenWhisk including the modifications we have made represented by in dotted lines. NGINX acts as a reverse proxy of the system and exposes a public HTTP endpoint to clients and forwards user requests to Controllers. The Controller performs load balancing and selects an Invoker instance to execute the function invocation. OpenWhisk by default implements memory bin packing: the Controller keeps track of memory usage of all pending invocations that are issued and iteratively directs all incoming invocations to one Invoker until the memory quota of that Invoker is exhausted. Controllers do not communicate with each other, and each Controller has access to all Invokers. The message delivery system between Controllers and Invokers is implemented using Kafka [29]. Invokers are usually deployed per VM and each manages a pool of containers, which are Docker containers by default. Depending on whether a suitable container exists, a function invocation is assigned to an existing container (warm start), or a newly created one (cold start). Existing containers are removed after a fixed keep-alive period (10 minutes by default) and when usable memory is inadequate to allocate a new container. Invocation results are stored in CouchDB for later retrieval.

6.2 Harvest VM-Aware Load Balancing

We modify both the Invoker and the Controller to implement the resource variation-aware MWS load balancing algorithm.
Invoker. We modify the Invoker so it can efficiently use the dynamically changing number of available CPUs. We introduce a module called *Harvest Monitor* in each Invoker that is responsible for periodically gathering: (a) the latest number of CPUs allocated to the Harvest VM using Hyper-V Data Exchange Service [11]; (b) the cumulative CPU time using *cpuacct.usage* interface from *cgroups* [37]; and (c) any scheduled deallocation event for the VM using Azure Metadata Service [40]. This information is embedded into the health pings that the Invoker sends to the Controller every second.

All function containers for the same user run in the same cgroup so that we can gather CPU utilization statistics. For each function invocation, the Invoker collects its (a) execution duration and (b) CPU usage by querying the cgroup for its container. The Invoker embeds the information in the invocation response message back to the Controller. In addition, the Invoker performs admission control by computing the current utilization as $(\frac{cpu_{usage}}{cpu_{avail}})$; if this is higher than a predefined threshold, new function invocations are delayed.
Controller. We modify the Controller to receive the additional information collected by the Harvest Monitors through the Invoker health pings. The Controller updates its local data structures with this information (off the critical path, using Scala Actors). It maintains (a) CPU usage, (b) available CPUs, and (c) eviction notifications events for each Invoker. If an Invoker has an eviction notification, the Controller stops sending new invocations to it. The Controller maintains local per-function histograms of the observed execution times and CPU usage. Each Controller independently constructs these histograms which eventually converge to similar values as more samples are collected. The Controller also maintains a per-function invocation arrival rate which is periodically updated. We multiply the arrival rate observed locally with the number of Controllers in the system (available at startup) to get an estimated total invocation arrival rate.

We use the expected values computed from the execution time and CPU usage histograms along with the estimated total invocation arrival rate as inputs to execute the MWS algorithm for the function. To mitigate the potential user load oscillation and smooth the worker set size changes, we set a minimal interval of 30 seconds between worker reductions.

Finally, the Controller also maintains the mapping of functions to their hash ID (used in assigning home VM based on consistent hashing) and hash IDs to list of functions (used for function to home VM assignment update in the face of Invoker arrival/departure) as described in Section 5.2.

Resource Monitor. We introduce a separate module per deployment, called *Resource Monitor*, to track the resource variation in our system. It periodically queries for the total available resources (*e.g.* CPUs) and spins up new VMs to maintain a minimum pool of available resources, if they fall below a pre-configured threshold. As mentioned in Section 4.1, this is important because reduction in the CPUs or eviction of Harvest VMs reduces the available resources and can adversely impact service quality.

7 EVALUATION

We first demonstrate the benefits of the MWS algorithm compared to JSQ and the default load balancing algorithm of OpenWhisk. Then we demonstrate the benefits and cost savings of Harvest VMs for hosting serverless workloads.

7.1 Experiment Setup

For this evaluation, we deploy OpenWhisk (PR#4611) [46] on Azure with Ansible [5]. We use one controller VM and a variable number of invokers with their own VMs. The controller VM contains core OpenWhisk components, including two controllers, NGINX and CouchDB. In this section, we use cluster size to refer to the number of invoker VMs.

We port multiple Python serverless functions from FunctionBench [32] to OpenWhisk as the benchmark (Table 2). We create a Docker image for each function for a total of 401 functions. We use Locust [38] to generate the workload with a Poisson arrival process, and MinIO [41] as the object store to serve the input data. We use the 99^{th} percentile latency denoted P99 as the SLO metric and set it to 50 seconds, which clearly indicates saturation for our benchmarks.

The experiments use actual Harvest VMs (where we cannot control how their resources vary) and Harvest VM traces. When using traces, each trace corresponds to a Harvest VM. To emulate the CPU changes from the trace on a regular VM, we use cgroups to set the CPU limit of the parent Docker group of all user invocations. Each experiment runs for 20 minutes unless otherwise stated.

7.2 Impact of Load Balancing

First, we compare three load balancing algorithms: min-worker-set (MWS), join-the-shortest-queue (JSQ), and vanilla OpenWhisk ($Vanilla$). We deploy OpenWhisk with 10 invokers, each hosted by a regular VM with 32 CPUs and 128 GB of memory. The CPUs of the invokers are asymmetric, with the maximum of 28 and the minimum of 5, to mimic the resource heterogeneity in Harvest VM clusters. Figure 12 depicts the $P99$ latency of the three algorithms.

Functions	Description
Floatop	Sine, cosine & square root
Matmult	Square matrix multiplication
Linpack	Linear equation solver
Chameleon	HTML table rendering
Pyaes	AES encryption & decryption
Image processing	Flip, rotate, resize, filter & grayscale images
Video processing	Grayscale video
Image classification	MobileNet inference
Text classification	Logistic regression

Table 2: The examined serverless functions from FunctionBench [32] and their description.

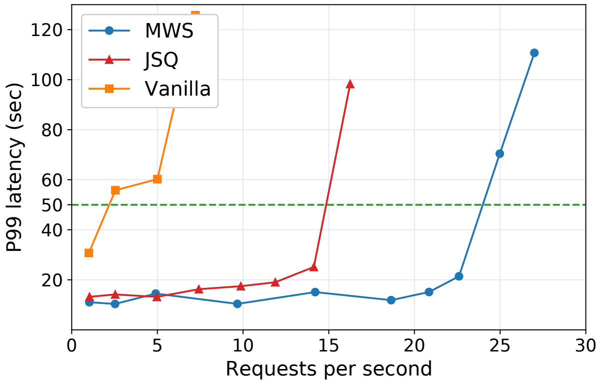


Figure 12: P99 latency across load balancing algorithms.

Throughput. We evaluate the throughput without breaking the SLO of each policy. MWS achieves a throughput 22.6× higher than the vanilla OpenWhisk load balancing. *Vanilla* has the worst throughput because it only considers memory and keeps allocating invocations to an invoker until the memory capacity of the invoker is exhausted. However, because of the scarcity of CPUs on some Harvest VMs, CPU tends to saturate at much lower load than memory. The CPU saturation caused by *vanilla* is also exacerbated by the heterogeneity of VM CPUs in the test cluster because even if there are additional CPU resources in the cluster, the invoker with the least CPUs will always saturate at low loads. MWS has a throughput 1.6× higher than JSQ because it improves locality.

Cold starts. Better locality reduces the cold start rate. Figure 13 compares the cold start rate of MWS and JSQ at their non-saturating loads. At the same input loads, MWS reduces cold starts between 56.0% and 75.9%. Figure 14 compares the latency of both policies. It shows that reducing cold starts, we also reduce the latency. With our strategy, we can provide the same latency with fewer VMs.

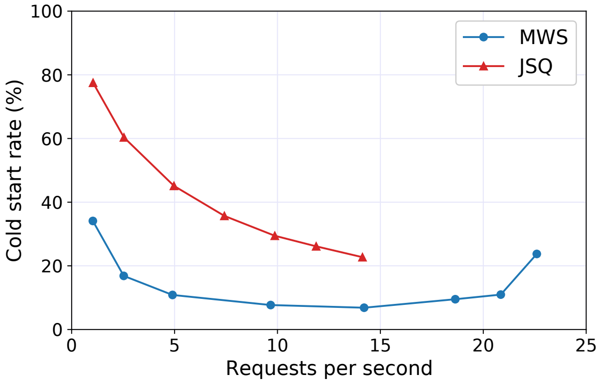


Figure 13: Cold start rate of MWS vs. JSQ.

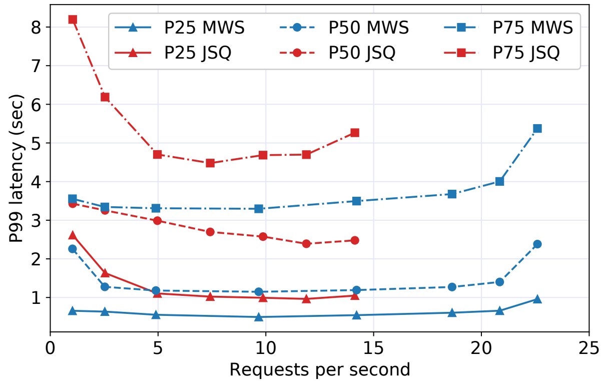


Figure 14: Low percentile latency of MWS vs. JSQ.

7.3 Impact of Resource Variability

We use Harvest VM traces that have frequent CPU changes with large change sizes to show the worst-case performance. Although CPUs of Harvest VMs are relatively stable in the normal case, they can also experience frequent and significant resource changes as depicted in Figure 3. Frequent CPU changes are challenging to handle since they require the load balancer to detect the changes and adjust task assignment promptly. Significant CPU shrinkage has a direct impact on system performance since pending activations on the Invoker that experiences significant shrinkage are prone to severe resource contention, especially at high loads.

To study the worst-case performance of Harvest VMs, we select a set of Harvest VM traces that have both extremely frequent and significant CPU changes. Specifically, we choose 8 real Harvest VM traces among the traces with the highest change frequency and large change size, and also synthesized 2 traces to control the total capacity of the cluster. The average CPU change interval in the set of 10 traces is 3.6 minutes, orders of magnitude shorter the expected CPU change interval for the common case as discussed in Section 3.1. The traces also include significant CPU shrinkage, with the maximum shrinkage size being 26 CPUs, meaning that 81.3% of all CPUs are suddenly taken away from an Invoker.

We compare the performance of this actively changing Harvest VM cluster (“Active”) to two clusters: “Normal”, a Harvest VM cluster with normal variations, and “Dedicated”, a cluster with dedicated resources using regular VMs. All three clusters have 180 CPUs total. The “Normal” harvest cluster has stable per-VM CPUs, but the size of each Harvest VM varies, with the largest VM having 28 CPUs and the smallest VM having 5 CPUs. The “Dedicated” cluster has both stable and homogeneous per-VM CPUs.

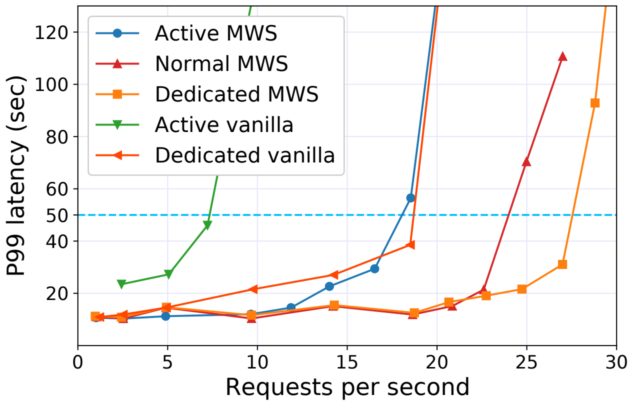


Figure 15: Performance of harvest clusters in normal case (“Normal”), under frequent and significant CPU changes (“Active”), and of the “Dedicated” cluster.

We compare the performance of these three clusters in Figure 15. “Active” achieves 73.1% throughput of the “Normal” harvest cluster and 61.2% of the “Dedicated” cluster. The frequent and significant CPU changes result in a higher cold start rate for the “Active” harvest cluster compared to “Normal” harvest cluster at similar loads as shown on the left side of Figure 16. The “Dedicated” cluster achieves 19% higher throughput than the “Normal” cluster because the small VMs in “Normal” are more prone to saturation at high loads. We also experiment deploying vanilla OpenWhisk on the “Active” and “Dedicated” clusters. Vanilla OpenWhisk only achieves 39.0% throughput on “Active” compared to “Dedicated” cluster. This demonstrates that MWS can better handle active resource variations. Even with the 26.9% performance loss for the worst case, we show in the next section that running serverless computing workloads on Harvest VMs significantly outperforms running them on regular VMs under the same cost budget.

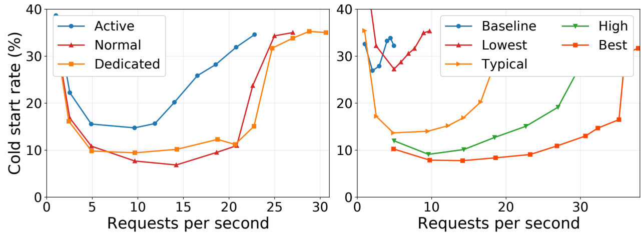


Figure 16: Cold start rate against load for fixed budget.

Discount	d_{evict}(%)	d_{harv}(%)	#VMs
Baseline (dedicated)	0	0	2
Lowest	48	48	6
Typical	70	80	12
High	80	90	18
Best	88	90	21

Table 3: Number of Harvest VMs with the same budget, based on the discount level.

7.4 Cost vs Performance

Cost. To evaluate the benefits of using harvested resources, we set a fixed budget and compare how many Harvest VMs we can provision and the load we can serve. As the budget baseline, we use two regular VMs with 16 CPUs and 64 GB of memory. We use the cost model introduced in Section 2 where the minimum resources and the harvested cores have a discount of d_{evict} and d_{harv} respectively. Table 3 shows the impact of the discounts. With the most pessimistic discount, we obtain 6 Harvest VMs, and up to 21 with an optimistic discount configuration.

Performance. Figure 17 compares the performance of each of these cluster configurations. These harvest clusters have 1.9×, 4.6×, 7.8× and 9.7× more CPUs than the baseline with 2 regular VMs. The throughput is 2.2×, 4.6×, 7.7× and 9.0× better as a result of cheaper harvested CPUs.

Cold start rate. The improvement in throughput is also reflected with lower cold start rates for each load value as depicted in Figure 16 (right side). Notice that at very low loads, all clusters have high cold start rates since the function invocations are spread across many VMs but without significant impact on latency. As the load increases, the cold start rate initially decreases in all configurations, and then increases as load approaches system capacity (around 25% at saturation).

7.5 Harvest VMs vs Spot VMs

Harvest VMs and Spot VMs are both evictable VMs that leverage surplus resources. In this section, we compare hosting serverless workload on Harvest VMs and Spot VMs via simulation, and focus on reliability and cost.

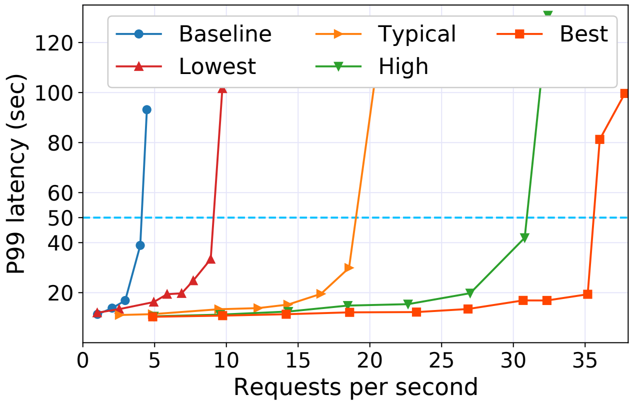


Figure 17: Regular vs Harvest VMs with same budget.

Experiment setup. For a fair comparison, we create synthetic Spot VM and Harvest VM traces with the idle resources of the same physical cluster (described in the characterization of resource variability in Section 3.1). For Harvest VMs, we place one VM on each node as long as the node can accommodate its base size, and the VM can harvest all idle resources on the node. For Spot VMs, we place as many as VMs as will fit on each node. Both Harvest VM and Spot VM are given a 30-second grace period before eviction. We use the same serverless workload trace as in Section 3.2, and pick the 5-day snapshot with aligning weekdays as the VM traces. We also extend the simulation framework in Section 4.1 to incorporate CPU usage: each invocation consumes one CPU and an invocation is buffered when the cluster runs out of CPUs. New containers are created on the VM with the least CPU utilization.

Sensitivity analysis. We analyze Harvest VMs with base size of 2, 4 and 8 CPUs (referred to as H2 to H8), and Spot VMs with size of 2, 4, 8, 16, 32 and 48 CPUs (referred to as S2 to S48), and the results are shown in Figure 18. $CPUs \times time$ is normalized against the idle $CPUs \times time$ of the physical cluster, and price is normalized against regular CPUs under the *Typical* configuration in Table 3.

Reliability. H2 achieves the lowest invocation failure with 4.31×10^{-6} (*i.e.*, "5 nines" of reliability). For Harvest VMs, the invocation failure rate increases with base size, reaching 3.54×10^{-5} at H8. For Spot VMs, invocation failure rate reaches its minimum of 1.00×10^{-4} at S2, but is significantly higher than Harvest VMs, being at least 23.2× higher than H2. The invocation failure rate on Spot VMs reaches the maximum at S16 and decreases with VM size afterwards. This is because the fragmentation caused by large VMs creates a larger buffer of unused resources that prevents VM eviction upon shrinkage of idle resources. Cold start rates show similar trends for the same reason.

Cost. To calculate the price, we incorporate the additional per-VM cost incurred by the framework installation as in [4]. Assuming an installation time of 10 minutes as in Section 3.1, we use the following equation:

$$\frac{base\ core\ time \times d_{evict} + harvest\ core\ time \times d_{harv}}{base\ core\ time + harvest\ core\ time - install\ core\ time}$$

With the same idle resources, Harvest VMs also provide more effective compute power ($CPUs \times time$) than Spot VMs at cheaper prices. H2 can utilize 99.62% of the total idle compute power, and S2 can only utilize 91.67%. H2 offers an amortized per-CPU price of $0.211\$/hour$, while the lowest per-CPU price of Spot VM is $0.313\$/hour$ (offered by S48). Harvest VMs are cheaper for two reasons: d_{harv} being smaller than d_{evict}, and less installation overhead as a result of less VM evictions. For Spot VMs, the effective compute power decreases with VM size as a result of fragmentation.

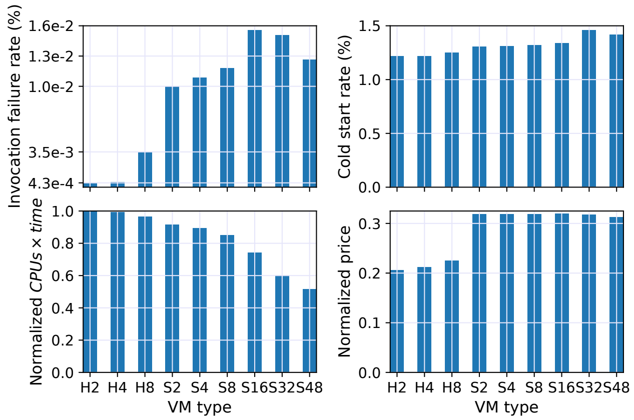


Figure 18: Harvest VMs vs Spot VMs. Hx refers to Harvest VMs with base size of x CPUs, and Sx refers to Spot VMs with x CPUs.

7.6 Running on Real Harvest VMs

We now demonstrate executing snapshots of the function traces on real Harvest VMs. For this experiment, we cannot control the number of available CPUs and just report the organic numbers.

Experiment setup. To reproduce the invocations from the function trace, we use CPU-intensive loops with the same duration. Because the maximum number of concurrent running invocations in the function trace is too high to fit in the size of our cluster, we combine multiple 2-hour snapshots with fewer concurrent running invocations, making it feasible to replay the function trace.

Figure 19 reports the number of concurrent running invocations (the peak is 120 invocations), and we provision a cluster with 150 CPUs so that its CPU utilization is below 80%.

VM type	Base CPUs	Max CPUs	Memory
Harvest	2	6	16GB
Regular	8	8	32GB
Spot-4	4	4	16GB
Spot-48	48	48	192GB

Table 4: Characteristics of the Harvest VMs, regular VMs, and Spot VMs used in the experiment in §7.6.

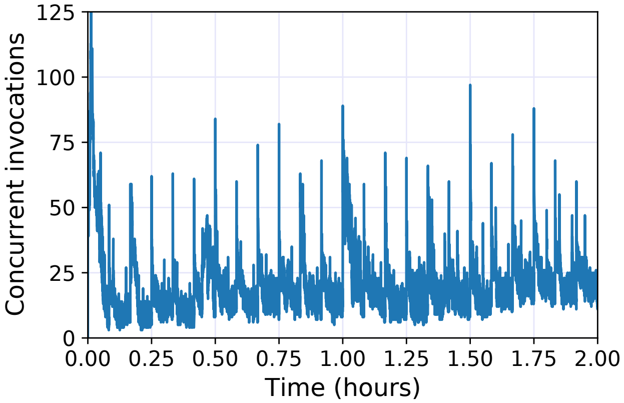


Figure 19: Invocations in the combined function trace.

Percentile	Harvest	Spot-4	Spot-48
25^{th}	56%	53%	53%
50^{th}	47%	43%	52%
75^{th}	32%	4%	38%
90^{th}	41%	15%	55%
95^{th}	74%	35%	83%
99^{th}	62%	16%	81%

Table 5: Latency reduction at multiple percentiles of Harvest and Spot VM clusters over regular VM clusters.

We test four clusters, consisting of Harvest VMs, baseline regular VMs, Spot-4 VMs and Spot-48 VMs (Table 4). We deploy MWS OpenWhisk on the Harvest VM and Spot VM cluster and the vanilla OpenWhisk on the regular VM cluster.

Utilization and performance. Figure 20 shows the total number of CPUs and the utilization for the Harvest, regular and Spot clusters. All clusters show similar CPU utilization patterns and the Harvest and Spot-48 clusters run all the functions with no failure. Figure 21 shows the invocation latency distribution of the tested clusters. Table 5 lists the latency reduction of Harvest and Spot VM clusters over the regular VM cluster at different percentiles. Harvest VMs outperform other alternatives except Spot-48 VM, because large VMs are less likely to be saturated, both in terms of CPUs and memory. However, using large Spot VMs leads to lower resource utilization and higher failure rate, as discussed in Section 7.5.

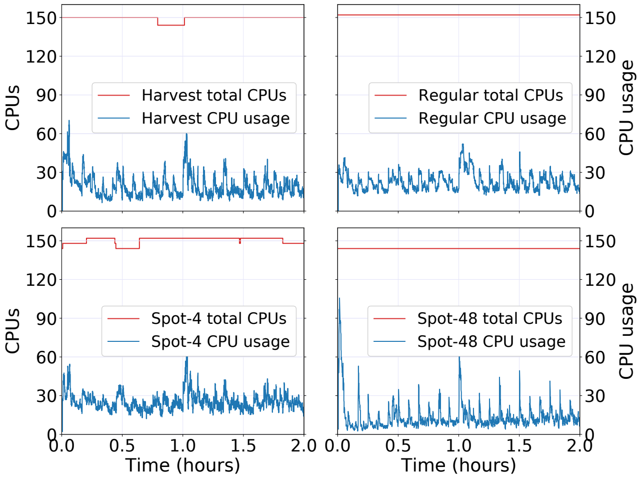


Figure 20: CPU number and cluster CPU utilization for Harvest VMs (upper left), regular VMs (upper right), and Spot VMs with 4 CPUs (lower left) and 48 CPUs (lower right).

Cost. We now compare the cost of Harvest VMs against regular VMs and Spot VMs, and we assume that the Spot VM cluster has the same configuration as the regular VM cluster. We analyze the four configurations of d_{evict} and d_{harv} from Table 3. Compared to regular VMs, Harvest VMs are 49%, 77%, 83% and 89% cheaper, respectively. Compared to their Spot-4 VMs counterparts, they are 0%, 22%, 45% and 11% cheaper, respectively. The worst case achieves no savings compared to Spot VMs because it pessimistically assumes harvested CPUs have the same price as evictable CPUs. This pricing is unlikely to happen in practice.

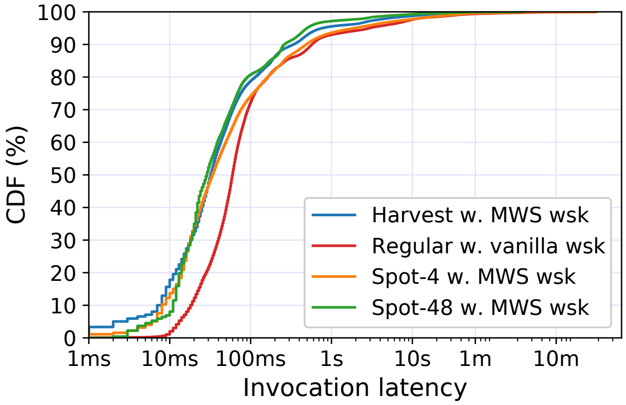


Figure 21: Response latency comparing MWS on harvested resources to vanilla OpenWhisk running on dedicated resources.

7.7 Summary

We demonstrate the performance benefit of MWS load balancing. It achieves 22.6× higher throughput than vanilla OpenWhisk, as it addresses resource variations. It also improves locality, resulting in lower cold start rates. With MWS, we realize the benefits of running serverless platforms on harvested resources, achieving lower cost and better performance: Under the same cost budget, running serverless platforms on harvested resources achieves 2.2× to 9.0× higher throughput compared to using dedicated resources; and with the same amount of provisioned resources, running serverless platforms on harvested resources achieves 48% to 89% cost savings, with lower latency due to better load balancing.

8 CONCLUSION

In this paper, we propose to host serverless platforms on harvested resources. We quantify the challenges of using harvested resources for serverless invocations, including Harvest VM evictions and resource variation by characterizing the serverless workloads and Harvest VMs of Microsoft Azure. We demonstrate the reliability of hosting serverless workloads on harvested resources with trace-driven simulation. We also design and implement a harvesting-aware serverless load balancer on OpenWhisk, with which we demonstrate the performance and economic benefits of hosting serverless platforms on harvested resources.

ACKNOWLEDGMENTS

We would like to sincerely thank James Mickens for his feedback and guidance while shepherding our paper. We also thank the anonymous reviewers for their extensive feedback on earlier versions of this manuscript. This work was partially supported by a Microsoft Research Faculty Fellowship and NSF grants NeTS CSR-1704742 and CCF-1846046.

REFERENCES

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI* (2020).

[2] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. SAND: Towards High-Performance Serverless Computing. In *USENIX ATC* (2018).

[3] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2021.

[4] Ambati, P., Goiri, Í., Frujeri, F., Gun, A., Wang, K., Dolan, B., Corell, B., Pasupuleti, S., Moscibroda, T., Elnikety, S., and Bianchini, R. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *OSDI* (2020).

[5] Ansible. Ansible is Simple IT Automation. https://www.ansible.com/, 2021.

[6] AWS. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot, 2021.

[7] AWS. AWS Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html, 2021.

[8] AZURE. Azure Burstable VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable, 2021.

[9] AZURE. Pricing - Linux Virtual Machines | Microsoft Azure . https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/, 2021.

[10] AZURE. Use Azure Spot Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/spot-vms, 2021.

[11] AZURE, M. Hyper-V Integration Services. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/integration-services.

[12] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI* (2014).

[13] BRAMSON, M., LU, Y., AND PRABHAKAR, B. Randomized Load Balancing with General Service Time Distributions.

[14] CARREIRA, J., FONSECA, P., TUMANOV, A., ZHANG, A., AND KATZ, R. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS* (2018).

[15] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *NSDI* (2005).

[16] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid datacenter scheduling. In *USENIX ATC* (2015).

[17] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS* (2020).

[18] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys* (2012).

[19] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *USENIX ATC* (2019).

[20] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, Fast and Slow: Low-latency video processing using thousands of tiny threads. In *NSDI* (2017).

[21] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI* (2011).

[22] GOG, I., SCHWARZKOPF, M., GLEAVE, A., WATSON, R. N., AND HAND, S. Firmament: Fast, centralized cluster scheduling at scale. In *OSDI* (2016).

[23] GOOGLE. Google cloud functions. https://google.com/functions/, 2021.

[24] GUPTA, V., BALTER, M. H., SIGMAN, K., AND WHITT, W. Analysis of Join-the-Shortest-Queue Routing for Web Server Farms. *Performance Evaluation 64*, 9-12 (2007), 1062–1081.

[25] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *SOSP* (2009).

[26] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the Cloud: Distributed Computing for the 99%. In *SoCC* (2017).

[27] JYOTHI, S. A., CURINO, C., MENACHE, I., NARAYANAMURTHY, S. M., TUMANOV, A., YANIV, J., MAVLYUTOV, R., GOIRI, I., KRISHNAN, S., KULKARNI, J., ET AL. Morpheus: Towards automated slos for enterprise clusters. In *SOSP* (2016).

[28] KAFFES, K., YADWADKAR, N. J., AND KOZYRAKIS, C. Centralized Core-Granular Scheduling for Serverless Functions. In *SoCC* (2019).

[29] KAFKA. Apache Kafka: A distributed streaming platform. https://kafka.apache.org/, 2021.

[30] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX ATC* (2015).

[31] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC* (1997).

[32] KIM, J., AND LEE, K. Functionbench: A suite of workloads for serverless cloud function service. In *CLOUD* (2019).

[33] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. Understanding ephemeral storage for serverless analytics. In *USENIX ATC* (2018).

[34] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI* (2018).

[35] KUBERNETES. Kubernetes Production-Grade Container Orchestration. https://kubernetes.io/, 2021.

[36] LEE, B. D., TIMONY, M. A., AND RUIZ, P. DNAvisualization.org: A Serverless Web Tool for DNA Sequence Visualization. *Nucleic acids research 47*, W1 (2019), W20–W25.

[37] LINUX. Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2021.

[38] LOCUST. Locust: A modern load testing framework. https://locust.io/, 2021.

[39] MICROSOFT AZURE. Azure functions. https://microsoft.com/en-us/services/functions/, 2021.

[40] MICROSOFT AZURE. Azure metadata service: Scheduled events for linux vms. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/scheduled-events, 2021.

[41] MINIO. Minio - high performance, kubernetes native object storage. https://min.io/, 2021.

[42] MOHAN, A., SANE, H., DOSHI, K., EDUPUGANTI, S., NAYAK, N., AND SUKHOMLINOV, V. Agile Cold Starts for Scalable Serverless. In *HotCloud* (2019).

[43] MVONDO, D., BACOU, M., NGUETCHOUANG, K., NGALE, L., POUGET, S., KOUAM, J., LACHAIZE, R., HWANG, J., WOOD, T., HAGIMONT, D., DE PALMA, N., BATCHAKUI, B., AND TCHANA, A. OFC: An Opportunistic Caching System for FaaS Platforms. In *EuroSys*.

[44] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC* (2018).

[45] OPENWHISK. Apache OpenWhisk Open Source Serverless Cloud Platform. https://openwhisk.apache.org/, 2021.

[46] OPENWHISK. OpenWhisk Pull Request 4611. https://github.com/apache/openwhisk/pull/4611, 2021.

[47] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, Low Latency Scheduling. In *SOSP* (2013).

[48] PARK, J. W., TUMANOV, A., JIANG, A., KOZUCH, M. A., AND GANGER, G. R. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *EuroSys* (2018).

[49] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI* (2019).

[50] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC* (2012).

[51] ROMERO, F., CHAUDHRY, G. I., GOIRI, Í., GOPA, P., BATUM, P., YADWADKAR, N. J., FONSECA, R., KOZYRAKIS, C., AND BIANCHINI, R. Faa$T: A Transparent Auto-Scaling Cache for Serverless

Applications. *arXiv preprint arXiv:2104.13869* (2021).

[52] SCHLEIER-SMITH, J., SREEKANTI, V., KHANDELWAL, A., CARREIRA, J., YADWADKAR, N. J., POPA, R. A., GONZALEZ, J. E., STOICA, I., AND PATTERSON, D. A. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Communication of the ACM* (2021).

[53] SHAHRAD, M., FONSECA, R., GOIRI, Í., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX ATC* (2020).

[54] SHILLAKER, S., AND PIETZUCH, P. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *USENIX ATC* (2020).

[55] TUMANOV, A., ZHU, T., PARK, J. W., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys* (2016).

[56] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*. 2021.

[57] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC* (2013).

[58] VVEDENSKAYA, N. D., DOBRUSHIN, R. L., AND KARPELEVICH, F. I. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii 32*, 1 (1996), 20–34.

[59] WANG, L., LI, M., ZHANG, Y., RISTENPART, T., AND SWIFT, M. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC* (2018).