Making Serverless Pay-For-Use a Reality with Leopard

Tingjia Cao, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Tyler Caraza-Harter, *University of Wisconsin-Madison*

https://www.usenix.org/conference/nsdi25/presentation/cao

This paper is included in the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Making Serverless Pay-For-Use a Reality with Leopard

Tingjia Cao Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau Tyler Caraza-Harter
University of Wisconsin–Madison

Abstract

Serverless computing has gained traction due to its event-driven architecture and "pay for use" (PFU) billing model. However, our analysis reveals that current billing practices do not align with true resource consumption. This paper challenges the prevailing SLIM (static, linear, interactive-only model) assumptions that underpin existing billing models, demonstrating that current billing does not realize PFU for realistic workloads. We introduce the *Nearly Pay-for-Use (NPFU)* billing model, which accommodates varying CPU and memory demands, spot cores, and preemptible memory. We also introduce *Leopard*, an NPFU-based serverless platform that integrates billing awareness into several major subsystems: CPU scheduler, OOM killer, admission controller, and cluster scheduler. Experimental results indicate that Leopard benefits both providers and users, increasing throughput by more than 2x and enabling cost reductions.

1 Introduction

Serverless computing, also known as Function-as-a-Service, is increasingly popular in cloud computing environments [43, 51, 64]. This paradigm enables users to build event-driven applications through a collection of short-lived functions [6, 15, 18, 22, 39, 62]. It lets users define the events that trigger these functions; the platform provisions resources for function execution when said events occur [55, 64, 66]. Serverless computing is economically appealing as it has "pay-as-you-go" billing: developers are only charged when their function invocations are active [5, 13, 28]. Therefore, most cloud providers have a serverless offering, including AWS [13], Azure [5], and Google Cloud [28], and a majority of organizations running cloud workloads utilize serverless in some capacity [54, 63].

However, our examination of the details of serverless billing models (*i.e.*, how much a user is charged for an invocation, based on function configuration and resource usage), reveals more nuance to what providers mean when they advertise a "pay-per-use model" (Azure [5]) or claim that you only pay "for what you use" (GCP [28] and AWS [13]). To providers, these terms apparently mean something akin to "fine-grained billing" with respect to wall-clock time. An ideal pay-for-use platform for customers, in contrast, would charge based on actual resource consumption.

Characterizing these platforms as pay-for-use is justifiable if one makes four strong assumptions about function resource usage. The assumptions are: (1) resource usage is constant across time; (2) resource usage is similar across invocations; (3) CPU/memory usage is proportional; (4) all invocations are interactive. When these hold for a workload, most popular serverless platforms charge in proportion to resource usage; otherwise, invocations are charged for unused resources. We define a billing model, called SLIM (static, linear, interactive-only model) that captures these common billing practices.

Our first contribution is a detailed empirical evaluation of a suite of serverless functions that we have assembled (§2). The suite consists of a wide range of functions from different domains (developer, machine learning, and database workloads) and runs on varied input payloads. Our analysis shows that serverless workloads do not match SLIM assumptions. CPU and memory usage often fluctuate during a single invocation (it is not constant); resource consumption depends on input size, which varies across invocations; memory usage is rarely proportional to CPU usage; and most functions have non-interactive use cases. Put simply, "pay-for-use" models often charge for both used and unused resources.

Our second contribution is a new billing model, *Nearly Pay-for-Use (NPFU)* (§3), that can approximate pay-for-use efficiently. NPFU enables users to better express actual resource demands and eliminate SLIM assumptions. Specifically, NPFU separates memory and CPU demands and adds spot cores and preemptible memory for less urgent (non-interactive) work. Note that this approach does not depend on provider subsidization to reduce customer cost; NPFU is more flexible than current serverless approaches and can generate a win-win scenario for *both* providers and users. Users (nearly) pay for what they use and can better reach their objectives while providers can allocate their resources more effectively.

Our third contribution is an exploration of how billing models affect system design. We introduce *Leopard* (§4), a serverless system based on OpenLambda [34] that supports NPFU. Leopard builds billing awareness into several major subsystems. At the kernel level, we add support to the isolation API (cgroups), CPU scheduler (CFS), and preemption mechanism (OOM killer). These improvements enable efficient co-location of interactive and batch functions, which use reservation and spot billing, respectively. The new admission control logic leverages this support to aggressively run more functions concurrently, without sacrificing QoS. Finally, the load balancer creates more opportunities to resell unused resource reservations as spot cores and preemptible memory. Leopard demonstrates that billing must be considered as part of the system design process, rather than a late-stage addition.

Finally, we demonstrate the benefits of NPFU through rigorous experimentation (§5). We first illustrate a new methodology for comparing billing models. We show that Leopard's billing awareness improves worker throughput by 2.3x (on average). This efficiency may be delivered to customers in the form of price reductions (while keeping provider revenue per machine constant): 34% for interactive functions and 59% for batch functions for the average invocation.

2 Pay-for-Use Serverless Billing Models

When customers pay cloud providers for services, the *billing model* describes the cost of those services. Most services are highly configurable with various knobs (*e.g.*, the number of threads to use, or amount of memory), and those knobs may impact cost. Thus, a billing model consists of those knobs as well as a *billing function* that computes the cost based on those knobs and the actual usage of the service.

There are two families of billing models: pay-for-use (PFU), designed to charge based on actual resource consumption, and provisioned. Increasingly, modern services offer dual billing options: AWS EC2 allows customers to pay for instances on-demand (PFU), or commit to long-term reservations (provisioned); GCP BigQuery offers on-demand pricing (PFU) and capacity pricing (provisioned); AWS Lambda offers its original model, which charges for memory time used by invocations (PFU), as well as a newer provisioned concurrency option. Our focus is on PFU billing models for FaaS.

Pay-for-use vocabulary is frequently used to market popular serverless platforms, yet this term is subject to interpretation. We formalize a strict interpretation of PFU that we call strict PFU (SPFU) (§2.1). Real serverless billing models are usually an approximation of SPFU where the quality depends on the workload. We define another billing model in the PFU family called SLIM: a static, linear, interactive-only model (§2.2). SLIM resembles many billing models in use today, but it poorly approximates SPFU for realistic workloads (§2.3).

2.1 Idealized Billing: SPFU

We formalize a strict pay-for-use model (SPFU). In SPFU, the cost for each function invocation is computed as a constant factor multiplied by the used resources (*e.g.*, CPU time and bytes of memory) measured at arbitrarily fine granularity. The values for those constants are determined by *pricing*, a concept related to, but distinct from, *billing*. If the usage of each resource was plotted over time, the cost would be proportional to the area under each curve, where each resource is billed independently. For simplicity, we focus on CPU and memory as the billable resources, but the model is extensible to others (*e.g.*, network throughput, disk utilization). Given that a "CPU minute now" can be argued to be more valuable than "a CPU minute whenever it becomes available", SPFU may charge more for functions marked interactive (*i.e.*, urgent).

SPFU avoids billing for anything that is not a computing resource, such as function invocations, wall-clock time, or resource limits. Not charging for invocations implies that SPFU cannot have minimum charges per invocation. CPU time is a billed resource, but wall-clock time is not; a function that blocks mid-invocation can only be billed for memory during this idle time. A service offering SPFU can allow customers to configure resource limits, but these limits must be excluded from the billing function; customers might find resource caps useful to bound costs, but SPFU does not allow providers to charge more if resource caps exceed actual consumption.

SPFU billing has advantages for customers: not paying for unused resources can reduce costs, and decoupling billing from resource limits simplifies configuration. However, implementing SPFU presents challenges for providers; in particular, SPFU complicates admission control, as customers have less incentive to specify accurate resource limits. Lacking accurate limits, providers may concurrently execute too many functions, forcing functions to be terminated if a memory spike occurs; alternatively, providers may concurrently execute too few functions, leading to low utilization. A billing model that is bad for providers is usually bad for customers: providers are likely to compensate for inefficiencies with higher pricing.

2.2 Practical Billing: SLIM

We now define SLIM billing (static, linear, interactive-only model), another billing model in the pay-for-use family. SLIM is a generic billing model resembling real billing models used by popular platforms and can be viewed as an approximation of SPFU. With SLIM, users can specify a memory limit for each function, where the limit applies to all its invocations; a CPU size is calculated that is proportional to this memory limit (*i.e.*, the memory and CPU caps cannot be adjusted independently). The memory and CPU sizes are guaranteed to be available during execution. The cost of an invocation is proportional to the execution time multiplied by the memory limit, regardless of how much memory is actually used.

SLIM may still be considered a pay-for-use billing model given a weak definition of "use", *i.e.*, execution time of a function multiplied by an upper bound on resource consumption. SLIM's main advantage is that it may be better for admission control than SPFU, as customers have a strong incentive to set resource limits as low as possible. We suspect SLIM is common because it lends itself to FaaS implementations with greater profit potential than SPFU.

2.3 Does SLIM Approximate SPFU?

We now consider to what extent SLIM approximates SPFU. *For realistic workloads, are "pay-for-use" serverless platforms strictly pay-for-use?* For some workloads, SLIM and SPFU may indeed produce similar billing outcomes. We enumerate four workload characteristics that must apply for SLIM and SPFU to produce similar bills (§2.3.1), collect a suite of realistic serverless functions (§2.3.2), and evaluate whether the SLIM assumptions hold for our workload (§2.3.3).

2.3.1 Workload Assumptions

Given the following four assumptions about the workload, we can expect SLIM to approximate SPFU:

Assumption SI (Static Internal): Resource usage is constant throughout the invocation of a function. SLIM invocation cost is proportional to a resource cap for an entire invocation. If usage is close to the cap, the cap properly approximates usage; however, if resource limits are selected to provision for short bursts of peak usage, tenants will pay for unused resources.

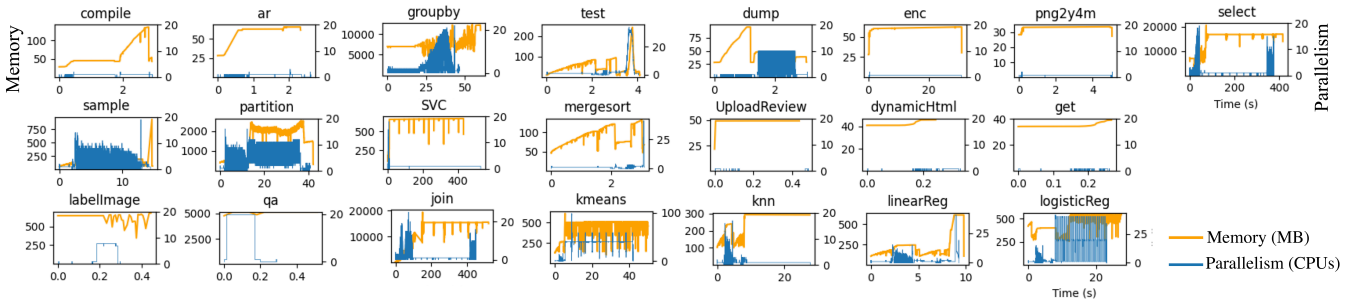

Figure 1: The diversity of parallelism behavior and memory usage over time within an invocation with medium payload.

Assumption SE (Static External): Resource usage is similar across invocations. SLIM uses same-size instances for all invocations of a given function. Resource limits are presumably chosen to be sufficient for the largest expected invocations so that those invocations will not fail. If many invocations have computational needs significantly lower than the large invocations, they will incur excess charges for unused resources.

Assumption L (Linearity): CPU and memory usage are linearly proportional. SLIM services offer instances with fixed compute-to-memory ratios (*e.g.*, 1 vCPUs to 1,769 MB of memory in AWS Lambda). As a result, CPU-dominant functions pay for unused memory resources (and vice versa). In contrast, functions that are relatively balanced across CPU and memory pay approximately for what they use.

Assumption I (Interactive): The QoS for all invocations are interactive. When invocations are triggered, SLIM-based platforms immediately provision resources. Although some triggers enable batching (*e.g.*, event hubs in Azure), when a batch arrives, the platform still immediately provisions instances for the batch. To support interactivity, providers maintain idle instances [51], although they could save memory by shutting down non-interactive instances. Under SLIM, all functions pay for low latency, even when it is not needed.

2.3.2 A Serverless Function Suite

Based on realistic workloads from a range of serverless research papers [21,23,27,39,42,47,48,70,72], we collected 22 serverless functions in 7 domains, representing applications in scientific, multimedia, engineering, and database usage scenarios, as listed in Table 1. For each function, we provide small, medium, and large input sizes. When possible, we try to anchor the sizes to actual inputs to make the workload more realistic (*e.g.*, the large input of the Compile function compiles the largest C file in the Linux kernel).

The function suite enables an analysis of SLIM assumptions across a variety of resource usage patterns. The average CPU and memory usage for each invocation varies significantly, from 0.04 CPU to 35 CPUs and from 28 MB memory to 28 GB memory, respectively. Additionally, the functions stress possible QoS: those that must complete urgently ("interactive") or those that prioritize cost and throughput ("batch"). Both types are desirable for the elasticity and pay-for-use characteristics of serverless computing [40–42, 62, 70, 73].

Application	Function	Possible QoS	
		Interactive	Batch
Software Compilation [21]	Compile	✓	
	Archive	✓	
	Testing	✓	
Video Analysis [23]	xc_dump	✓	✓
	xc_enc	✓	✓
	png2y4m	✓	✓
Batch Analysis [62]	Sample		✓
	Partition		✓
	Mergesort		✓
DB Client [39]	Upload	✓	
	DynamicHtml	✓	
	Get	✓	
DB server operations [62,74]	Groupby	✓	✓
	Join	✓	✓
	query	✓	✓
ML Inference [21,36]	ImageLabel	✓	
	Q and A	✓	
ML Training [40]	Kmeans		✓
	KNN		✓
	LinearReg		✓
	LogisticReg		✓
	SVC		✓

Table 1: A serverless function suite.

2.3.3 Workload Analysis

We now revisit those assumptions and evaluate how well they hold for realistic workloads.

Assumption SI: To evaluate whether resource usage is constant throughout a single invocation, we measure the parallelism of each function and resident memory size over time; for parallelism we trace the number of runnable tasks throughout an invocation by recording when tasks join a CPU's runqueue within Linux. As shown in Figure 1 for medium-sized functions, CPU and memory consumption varies dramatically during the execution of a single invocation.

Assumption SE: Figure 2 plots the memory and parallelism under different payloads for three representative functions. We observe that resource usage can be quite different across invocations: larger input usually corresponds to greater resource consumption. On average, CPU usage for large sizes is 1.5x higher than medium and 3x higher than small sizes, while memory usage is 8x and 16x greater, respectively.

Assumption L: Figure 3 plots the memory-to-CPU ratio for average and max usage for all functions; the line shows the memory-to-CPU ratio of AWS Lambda and extrapolated past the max support size (Google Cloud Functions and Azure Functions use similar ratios). We observe a departure from the fixed AWS Lambda ratio for most functions; 91% could use more CPU than memory relative to the AWS Lambda ratio, suggesting that CPU may be more scarce than memory.

Platform	Knobs	Instance Size r: memory to cpu ratio	Concurrent Invocations per instance	Bill per Execution T: Execution time CP: CPU unit price *MP*: memory unit price	Resource Limitation
AWS Lambda	Memory (M)	Memory: M, CPU: M*r	1	M * T * MP + M * r * T * CP	10GB Mem, 6vcpu
GCP	Memory (M)	Memory: M, CPU: M*r	1	M * T * MP + M * r * T * CP	32GB Mem, 8vcpu
Azure (premium)	Memory (M), Target	Memory: M, CPU: M*r	< target	M * T * MP + M * r * T * CP	14GB Mem, 4vcpu
Azure (consumption)	Target	Memory: 1.5GB, CPU: 1vcpu	< target	AvgMemory * T * MP	1.5GB Mem, 1vcpu
CloudFlare	Memory (M)	Memory: 128MB CPU: No dedicate CPU	1	AvgCPU * T * MP	128MB Mem, 10ms cputime

Table 2: The resource allocation policy and billing model of mainstream commercial serverless platforms.

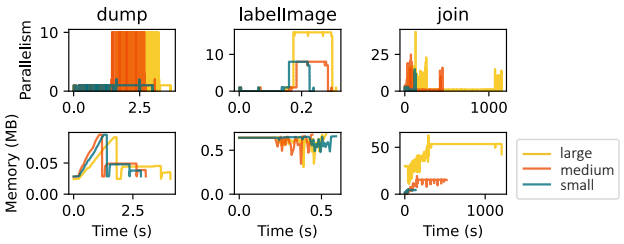

Figure 2: The resource demands with different payloads.

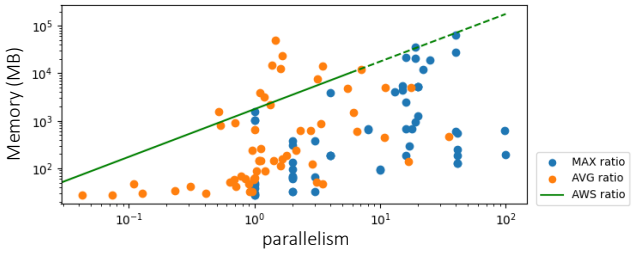

Figure 3: The ratio of cpu/memory demands.

Assumption I: In Table 1 the "possible QoS" column indicates whether a function must be urgently completed upon invocation, *i.e.*, is it latency critical (interactive) or not (batch). For example, model training is typically done offline; video analytics are used in both real-time stream processing systems and as background jobs. Thus, not all functions require interactivity all of the time.

Conclusion: Our analysis shows that serverless workloads challenge all four SLIM assumptions. CPU and memory usage fluctuate during a single invocation (assumption SI); resource consumption depends on input size, which varies across invocations (SE); memory usage is rarely proportional to CPU usage (L), and most functions have some non-interactive use cases (assumption I). Therefore, SLIM is a poor approximation of SPFU.

2.4 Serverless Billing in Practice

We now summarize the billing models in practice, as shown in Table 2. AWS Lambda, GCP, and Azure Functions (Premium Plan) closely resemble SLIM, the weakest interpretation of PFU billing. While Azure Functions (Consumption Plan) and Cloudflare align with SPFU, they enforce small, untunable resource limits (*e.g.*, Azure allocates 1.5 GB of memory and 1 vCPU and Cloudflare Workers are limited to 128 MB of memory and 10 ms of CPU time); these limits prevent customers from over-provisioning resources, but restrict the platforms' generality, prohibiting functions with larger resource requirements. Existing billing models either resemble SLIM, or SPFU with restrictive resource limit options.

3 NPFU: Nearly Pay-for-Use Billing

SPFU billing offers the most natural interpretation of "pay-for-use", but is rarely used in practice because providers must provision for resources that are unused (and unbilled). Instead, most providers adopt a billing model that resembles SLIM. We propose an alternative billing model that more closely approximates SPFU while exceeding SLIM's support for efficient and profitable FaaS implementations. We call this new model NPFU, Nearly Pay-for-Use. NPFU has two goals:

Profitable: The billing model should lend itself to efficient FaaS implementations. The drawback of SPFU is that it generally requires implementations to provision resources that may go unbilled. An ideal billing model will support greater utilization, leading to greater profits for providers (and perhaps lower costs for consumers).

Closely Approximates SPFU: SPFU offers the strongest interpretation of "pay-for-use" terminology, while SLIM only approximates SPFU for unrealistic workload assumptions. A good PFU billing model should better approximate SPFU for a wide range of workloads. Therefore, it requires adding proper knobs that enable users to describe resource requirements more accurately for workloads that breaks SLIM assumptions.

We introduce NPFU, defining the knobs for billing (§3.1) and the billing function that calculates cost per invocation (§3.2); we then compare NPFU to other billing models (§3.3).

3.1 NPFU Knobs

NPFU provides four knobs for customers to describe their resource demands: *cpu-cap*, *spot-cores*, *mem-cap*, and *preemptible-mem*. These options allow users to independently specify their CPU and memory demands instead of using the strict memory-to-CPU ratio of SLIM.

CPU: Cap and Spot. NPFU requires serverless users to specify *cpu-cap*, which is the maximum number of CPUs it can use; this bounds their cost. Given that many serverless functions are not interactive, NPFU introduces *spot-cores*: a subset of *cpu-cap* that a function does not need immediately, but could use if available. NPFU guarantees that *cpu-cap* minus *spot-cores* is available; this number of *reserved-cores* number is crucial for ensuring the function meets its QoS. Spot cores improve CPU usage: if instances underutilize their reserved cores, other instances can use them as spot cores.

Figure 4 shows an example of a server with 4 physical CPUs, where at t_0, the server hosts function invocations F1 and F2. F1 sets *cpu-cap* to 4 and *spot-cores* to 1 (*reserved-cores* is 3); F2 sets its *cpu-cap* as 3 and *spot-cores* as 2

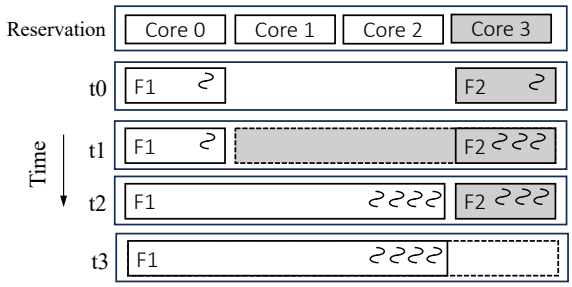

Figure 4: NPFU stresses allocated but unused cputime.

(*reserved-cores* is 1). The system ensures that each invocation can expand to its full set of reserved cores. At t_1, F2 forks two children; since there are idle CPUs from F1's reserved cores, F2 expands to use 3 CPUs. However, at t_2 when F1 forks 3 children and needs more cores, F2 immediately shrink to its reserved allocation. By t_3, F2 terminates, at which point F1, which has pending tasks, expands to all cores.

Decoupling *cpu-cap* and *spot-cores* helps batch functions describe low CPU urgency. It also benefits some interactive functions: although these functions have strict deadlines, their *reserved-cores* do not need to match maximum parallelism if deadlines can be met with fewer cores. Adding *spot-cores* creates opportunities to further reduce latency and costs.

Memory: Cap and Preemptible. Like the CPU knob, NPFU requires users to specify *mem-cap*, which is its maximum amount of memory; however, NPFU does not expose knobs for spot memory because reclaiming memory that has been borrowed by other functions can impose unacceptable overhead for interactive functions. Instead, NPFU allows users to simply specify if instances can be preempted (*preemptible-mem* as true). Preemptible instances can use the allocated-but-idle memory of non-preemptible instances; if memory usage exceeds physical memory, the preemptible instances are killed, allowing non-preemptible instances to access the reserved memory with minimal overhead. NPFU ensures that the total mem-cap of non-preemptible invocations does not exceed the available memory.

3.2 NPFU Billing Function

NPFU provides a flexible billing model that generates a win-win scenario for providers and users. NPFU leverages a used/lent model that enables users to buy/sell CPU and memory capacity from/to each other.

To calculate CPU cost, we define three kinds of cputime: *reserved-cputime*, the total reserved cputime (*execution_time* $\times$*reserved-cores*); *lent-cputime*, the idle reserved cputime used by other functions; *borrowed-cputime*, the used cputime of spot CPUs. The CPU cost, where C_r is the unit price for *reserved-cores*, and C_s for *spot-cores*, with $C_r > C_s$, is:

$$\begin{aligned} & C_r \times \textit{reserved-cputime} \\ + & \ C_s \times \textit{borrowed-cputime} \\ - & \ C_s \times \textit{lent-cputime} \end{aligned} \tag{1}$$

To calculate memory cost, non-preemptible memory acts as the lender, while preemptible instances are borrowers. For non-preemptible instances, the memory cost is:

$$\begin{aligned} & M_r \times \textit{mem-cap} \times \textit{execution\_time} \\ - & \ M_p \times \textit{avg-lentMem} \times \textit{execution\_time} \end{aligned} \tag{2}$$

$M_r > M_p$ since reserved memory is more valuable than preemptible memory. The cost for preemptible instances is:

$$M_p \times \textit{avg-memory} \times \textit{execution\_time} \tag{3}$$

If a preemptible instance is preempted mid-invocation, the customer is not charged, and its invocation is simply requeued.

3.3 Comparison with existing PFU family

NPFU can reproduce many of the advantages of SLIM. Specifically, NPFU offers both bounded performance and bounded cost by allowing users to specify minimal resources (with *reserved-cores* and non-preemptible *mem-cap*); and maximal resource boundaries (using *cpu-cap* and *preemptible* option). Although the cost can be less predictable when functions lend or borrow CPU, it ensures a clear upper bound: for functions with *reserved-cores*, the upper bound occurs when there are no available *spot-cores* and no CPU time is lent to other invocations; for functions without *reserved-cores*, the cost is determined by the used CPU time. Furthermore, the used/lent billing function ensures that providers can maintain the same revenue as SLIM.

In terms of simplicity, NPFU introduces three more knobs than SLIM, which is acceptable as simplicity is not merely a knob count; a single knob that must be tuned with high precision is arguably more complex to tune than a handful of less sensitive knobs. SLIM offers a single "size" knob that controls CPU and memory for all function invocations, making four unrealistic assumptions and requiring precise tuning to estimate the worst-case scenario: overestimating results in an overpayment, while underestimating causes the largest invocations to fail. NPFU can be configured to behave identically to SLIM by setting *cpu-cap* to be linear with *mem-cap*, *spot-cores* to 0, and *preemptible* to false; customers who prioritize simplicity could use these default values.

NPFU takes significant steps toward approximating SPFU. NPFU addresses SLIM's workload assumptions: it enables functions to sell their "allocated but unused" CPU time and memory (removing Assumption-SI and SE); decouples CPU and memory (resolving Assumption-L); and enables users to specify their urgency on requested resource by setting reserved (or spot) CPUs and non-preemptible (or preemptible) memory (removing Assumption I). NPFU allows batch function to set *reserved-cores* to zero and use preemptible memory, aligning their billing with SPFU; for interactive functions, NPFU offers a discount on wasted resources, making it more cost-effective than SLIM.

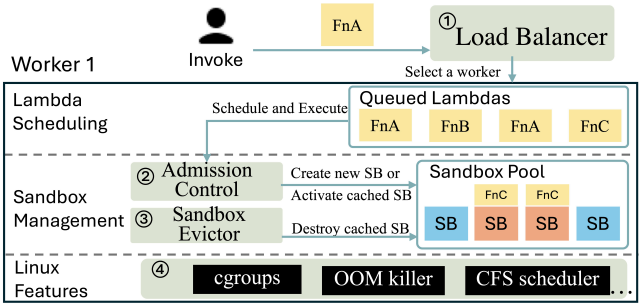

Figure 5: Overview of popular serverless platforms.

4 Billing-Aware Serverless

FaaS platforms are a composition of many subsystems, including load balancers, admission controllers, and kernel-level schedulers and resource controllers; efficiently supporting new billing models, such as NPFU, requires support from each of these subsystems. We give background for each subsystem (§4.1), describe the limitations of each for billing (§4.2), and propose Leapord (§4.3), a new FaaS platform with billing support built into every layer of the stack.

4.1 Background: Serverless Subsystems

Figure 5 illustrates the key components in a serverless platform. ① *Load Balancer:* dispatches incoming invocations to workers using a load-balancing strategy. ② *Admission Control:* selects the next request for execution and creates, or activates, a sandbox when sufficient resources are available. ③ *Sandbox Evictor:* removes cached sandboxes when too many are in memory ④ *Resource Isolation:* manages CPU and memory allocations across sandboxes. We now describe the typical implementations for each.

Load Balancing: In mainstream serverless platforms, after reaching the global controller, invocations are immediately dispatched to workers, eliminating memory usage for arguments and enabling scalability [4,34,45]. To avoid cold starts, the load balancer aims to minimize the number of workers handling invocations from the same function. For example, OpenWhisk [4] uses consistent hashing by mapping invocations to "home" workers; if a home worker is overloaded, the function is forwarded along the ring until a non-overloaded worker is found. Systems [9] built on Kubernetes [1] use greedy load balancing, sending requests to the first worker until it reaches capacity. Hermod [45] uses a greedy algorithm under low load and a least-loaded algorithm during high load.

Admission Control: The selected worker determines when to execute the incoming invocations, deciding how many invocations can run concurrently while ensuring that reserved resources are maintained for each invocation. For example, in OpenLambda [34], the provider manages the number of active sandboxes to ensure that total CPU/memory allocation across all active sandboxes does not exceed the physical resources available on each worker. Platforms typically use a priority queue; to improve average slowdown, many use Shortest Job First [16], estimating runtime based on historical data [45].

Sandbox Evictor: When a function completes, its sandbox is kept alive for future use. Cached sandboxes reduce latency but consume memory, so deciding when to evict cached sandboxes is critical. For instance, OpenLambda evicts the LRU sandbox when memory usage exceeds a threshold. Some approaches retain idle sandboxes for a specified duration [5,13,64]. FaasCache [26] evicts sandboxes only when memory is insufficient, evicting sandboxes with lower usage frequency, longer runtime, and larger memory footprints.

Resource Isolation: Resource isolation is crucial for maintaining performance in multi-tenant environments. To isolate requested CPU and memory resources, most serverless platforms use containers [4,9,34] and microVMs [13], such as Docker [3], Kubernetes pods [10], OpenLambda's SOCK [59], and AWS Lambda's Firecracker [11]. These solutions all rely on Linux cgroups. For memory management, cgroups provide the memory.max option, which enforces a memory limit for all tasks in the group; if a task exceeds this limit, the Out-of-Memory (OOM) killer is triggered and terminates one or more tasks based on metrics like memory consumption and age. Linux cgroups offer two mechanisms to manage the CPU: *CPU pinning* restricts tasks to specific CPUs, supporting both minimum and maximum CPU limits; *weighted sharing* relies on Linux's Completely Fair Scheduler (CFS) to allocate CPU time based on proportional shares. CFS divides available CPU cycles among threads in proportion to their weights. In multicore systems, each physical CPU has its own runqueue; load balancing periodically adjusts task distribution, moving tasks from burdened CPUs to lightly-loaded CPUs. With these mechanisms, containers can implement their own resource models. For instance, Kubernetes's "request" interface uses CPU sharing to reserve at least the requested amount of CPUs for a specific container [2]. The correctness and efficiency of its reservation depend on the implementation of cgroup.

4.2 Subsystem Support for NPFU: Challenges

Given that these subsystems were built without considering billing, each has limitations that make a correct and efficient implementation of NPFU difficult.

Challenge 1: cgroup API does not support efficient CPU reservations. NPFU uses *reserved-cores* to maintain performance and *spot-cores* to allow CPU sharing. The CPU scheduler should support two requirements. First, cgroups must be assured immediate access to their *reserved-cores* when needed (correctness); second, cgroups should be allowed to use idle *reserved-cores* of other co-located cgroups (efficiency).

The cgroup API and underlying CFS scheduler support both CPU pinning (via CPU sets) and weighted sharing (via CPU shares), but neither meets our requirements. To demonstrate the issue, we perform an experiment: two functions run concurrently, each of which has "paid" to reserve 16 of the machine's 32 cores. Function F1 runs 32 tasks continuously, and is willing to pay for additional spot cores. F2 starts with one task, fans out to 16 tasks, and does not use spot cores.

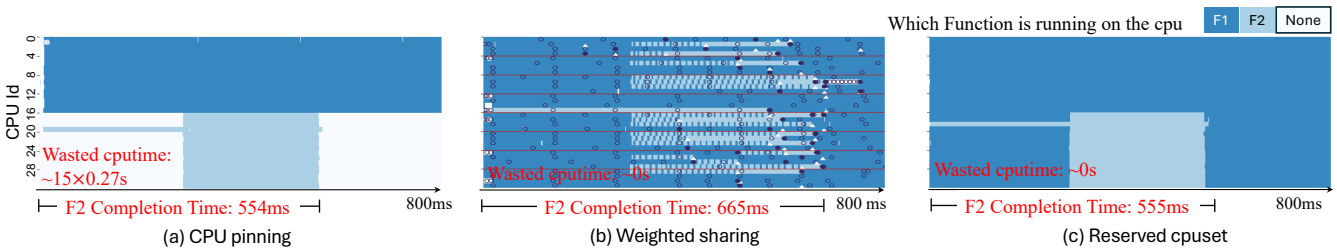

Figure 6: Colocation of Function 1 (32 processes) and Function 2 (1 process forks 15 processes at 270ms).

First, we evaluate CPU pinning via CPU sets in Figure 6(a). Reservations are implemented by applying a CPU set to all functions: the CPU set for F1 is cores 0-15, and for F2, it is cores 16-31. The figure shows that each function has sole use of its reserved cores, but the strictness of CPU sets wastes ∼4 seconds of CPU time on cores 16-31 that F1 would have been willing to purchase (at a reduced rate). Thus, CPU pinning only support inefficient reservations.

We evaluate weighted sharing with cpu.share in Figure 6(b); cpu.share the task to run on any core. We give both functions an equal share and expect each to be allocated approximately its reservation of 16 cores. The results show that no CPU is wasted because F1 uses whatever is available, but F2 is denied full access to its reserved CPUs and runs 20% slower. We conclude that cpu.share cannot correctly provide reservations.

CFS load balancing is resource-intensive, involving computational costs and cache misses, so CFS uses many heuristics to reduce overhead. For example, when CFS finds no imbalance, it doubles the interval before its next load balancing attempt and keeps the interval until the next successful migration. These heuristics undermine CFS's ability to promptly split CPU time among cgroups according to their cpu.shares. Figure 6(b) illustrates when CFS load balancing occurs: empty circles represent load balancing without migration, black circles represent migration, and white triangles show the source CPU of the migration. The figure shows that the interval between load balancing is around 100 ms. The infrequent rebalancing of cpu.shares is not sufficient for serverless workloads, where billing occurs at 1 ms granularity.

An alternative approach is to adjust the CPU set or the share weight of serverless functions over time dynamically, pushing their actual CPU usage closer to their proportional shares. However, it introduces overhead (*e.g.* adding a core to a VM takes approximately 0.1 milliseconds [14]), impacting millisecond-scale serverless invocations. The high overhead also limits the frequency of adjustments [32, 35, 46].

Challenge 2: sandbox evictor does not support preemptible memory. NPFU introduces a preemptible memory knob to allow efficient memory usage, and, to utilize idle reserved memory, NPFU allows preemptible sandboxes to be admitted even when all of memory is allocated. However, if memory is exhausted, the Linux OOM handler takes over before the sandbox evictor can react. The Linux OOM killer lacks access to sandbox-specific properties and may kill non-preemptible instances, violating the guarantees of NPFU. It also prevents the implementation of more sophisticated eviction strategies; for example, FaasCache [26] needs statistics for each sandbox such as execution recency, invocation frequency, and termination costs. Thus, NPFU requires improvements to the sandbox evictor and Linux OOM handler.

Challenge 3: NPFU complicates load balancing and admission control. By decoupling CPU and memory, NPFU introduces complexity for load balancing, leading to a multidimensional bin-packing problem and increasing the likelihood of stranded resources. This issue is worsened by locality-aware load balancing, which may assign memory- or CPU-heavy functions to the same "home" worker. Additionally, current loaded balancers rely on simple metrics like the number of invocations; consequently, interactive tasks may experience degraded performance even when another worker has sufficient unreserved resources but appears loaded due to batch tasks. Similarly, admission control that does not distinguish between spot CPUs and reserved CPUs will not know when it is safe to admit a given function invocation.

4.3 Leopard: Billing-Aware FaaS

As existing subsystems cannot handle novel billing models, we introduce a new billing-aware FaaS platform, Leopard, based on OpenLambda. Leopard implements spot CPUs and other NPFU features with several new subsystems. It introduces a new resource isolation mechanism for task groups, including CPU scheduler support for reservations (§4.3.1) and a billing-aware out-of-memory (OOM) killer (§4.3.2). It also presents a serverless-specific admission control policy at the worker level that accounts for spot CPUs (§4.3.3). Additionally, it introduces a serverless-specific cluster-level load balancer that considers both alignment and locality (§4.3.4).

4.3.1 Efficient CPU Reservations

Linux cgroups and CFS cannot support CPU reservations, so Leopard provides a new cgroup interface, *cpu.resv_cpuset*. Tasks in this new type of cgroup are guaranteed reserved CPUs on demand, but other cgroups may use the reserved CPUs if the reserver does not need them. To borrow a legal term, a cgroup reserving some cores has the *right of first refusal* for using them. Unlike the prior CPU-set interface, cpu.resv_cpuset does not prevent a cgroup from running on CPUs outside of the set; non-reserved CPUs may be used if computational needs exceed the reservation and there is a willingness to pay for additional spare compute.

Leopard modifies the CFS scheduler to support the new cgroup reservation API. The new CFS mode has two goals:

(1) Do not rely on fairness to achieve isolation. The previous approach of weighted sharing in cgroups relies on fairness, which current CFS implementations do not enforce effectively due to the high overhead of monitoring other cores for load-imbalance. In contrast, a new approach that ensures isolation for reserved cores is simpler and less resource-demanding, as it can be achieved locally without inter-core checks.

(2) Allow tasks to use flexible policies on different cores. CPU pinning allowed different policies for different cores (*e.g.*, a sandbox was permitted or forbidden on each core), but this policy was too blunt. We propose an enhanced model, where the scheduler differentiates more refined policies. Specifically, on reserved cores, a task can have the highest priority allowing immediate preemption; conversely, on other cores, the same task can have a lower priority, following spot tasks rules. This approach ensures that the new scheduler achieves both reservations and flexible sharing.

Based on these goals, Leopard adds reservation support in CFS as follows. When CPUs are added to *cpu.resv_cpuset*, the core marks the tasks as *reserving tasks*; other tasks are treated as *spot tasks*. When a core scheduler picks a task, it always picks a reserving task if any are runnable; otherwise, it pulls a runnable reserving task from the busiest runqueue in its cgroup; if that fails, the scheduler picks a spot task on the core's runqueue or makes an *idle-balance* CFS call to find a spot task elsewhere. When a reserving task wakes up (*e.g.*, after an I/O completion) and a spot task is running, the reserving task preempts the core immediately.

To maintain fairness within a cgroup, Leopard keeps the CFS design for managing tasks inside cgroups, but divides the load metric of a runqueue to track classes of tasks. When one CPU triggers load balancing, if the current task is a reserving task, the scheduler only balances the reserving cgroup load; if the current task is a spot task, the scheduler balances the load of all spot groups.

Figure 6(c) shows that Leopard's new reservation API behaves as desired: there are no wasted cores, and each task receives its reserved allocations. Specifically, F1 expands to more cores when they are available, and F2's performance is not harmed because it has top priority on cores 0-15.

In addition, to compute CPU billing in Equation 1, Leopard tracks reserved CPUtime, spot CPUtime, and lent CPUtime of the invocations by modifying the cgroup CPU Accounting (*cpuacct*) subsystem, which reports the used CPUtime of a cgroup. Linux updates the used CPUtime of a cgroup incrementally by calling *cgroup_account_cputime(Task p, int delta_exec)* when a task's state changes.

We add three interfaces in *cpuacct*: *resv_usage*, *spot_usage*, and *lent_usage*. Each time *cgroup_account_cputime* is called for a task p, (1) if the current CPU is p's reserved CPUs, add *delta_exec* to *resv_usage* of p*'s* cgroup; (2) otherwise, add *delta_exec* to the *spot_usage* of p's cgroup, and to the *lent_usage* of current CPU's reserving cgroup. The overhead is negligible, as it involves simple arithmetic operations within *cgroup_account_cputime* when the task state changes. Once the invocation finishes, the CPU billing can be computed by reading these metrics from the cpuacct interface, with less than 1 ms per invocation.

4.3.2 Sandbox Evictor

NPFU allows preemptible sandboxes to be admitted even when all memory is allocated to non-preemptible sandboxes, so memory exhaustion may occur under heavy load. However, Linux's original OOM handler sometimes takes over sandbox eviction before the userspace sandbox evictor can react. To support NPFU, Leopard contains a mechanism that gives control to the sandbox evictor during OOM events.

Leopard enables customizable OOM victim selection for different cgroups, allowing evictors to use rich user-space information. Specifically, Leopard contains two new Linux cgroup APIs for handling out-of-memory scenarios: *memory.oom.listener* specifies a user-space process to perform victim selection; *memory.oom.victim* sends the selected victim's PID to the kernel. For example, in user-space, Leopard can estimate the revenue lost for preempting different sandboxes (since preempted sandboxes are not charged) and preempt the sandbox that loses the least revenue; this information is unknown in the kernel. Specifically, when receiving an OOM signal, the evictor first selects cached sandboxes (*i.e.*, the same policy as FaasCache [26]), but if none are available, chooses the active preemptible sandbox with the shortest runtime and using the fewest resources, in order to lose the least revenue.

With this mechanism, Leopard can easily support sophisticated preemption policies. The communication overhead of victim selection in userspace is minimal ($\sim$0.4 ms), and is mitigated by proactive eviction and admission control of preemptible sandboxes. In summary, these enhancements allow Leopard's evictor to make informed decisions during OOM events caused by including preemptible sandboxes.

4.3.3 Admission Control

Leopard uses the following criteria to determine whether to admit an invocation into a sandbox: interactive invocations with *resv-cores* or non-preemptible memory are admitted if sufficient unreserved resources are available; batch invocations with preemptible memory and no *resv-cores* are admitted if currently idle memory and CPU resources are sufficient.

To determine whether or not preemption is likely, Leopard collects historical CPU and memory usage from past invocations and monitors the total of these averages for current active invocations. If the sum of historical memory usage exceeds the total memory, there is a high likelihood of preemption. In such cases, to avoid the risk of OOM events and unnecessary preemptions, Leopard does not admit the next batch function. Overcommitting CPU is less harmful, as it does not trigger preemption; therefore, Leopard admits new batch functions as long as the total historical CPU usage remains less than twice the total CPU resources.

When reserved resources are released, Leopard admits queued interactive invocations using the Shortest Job First. When the utilization of a worker falls below a threshold, Leopard admits the batch request with the shortest runtime that does not request reserved capacity. Thus, Leopard ensures high utilization for each worker while ensuring that reserved resources for active sandboxes are available when needed.

4.3.4 Load Balancer

Leopard provides a new load balancer for invocations. At low load, function placement does not significantly impact performance, so Leopard prioritizes alignment (to strand fewer resources) and locality (to improve warm-up). For alignment, we follow Grandl *et al.* [29] and define the alignment score of a task relative to a worker, *score-align*, as the dot product between the vector of task's reserved resources and the worker's available resources; a higher score indicates a better fit. To preserve locality, Leopard calculates a *score-locality* for each worker based on its priority scheduling with the traditional hashing sequence [4]. At low load, Leopard places tasks based on a combined score for each worker as $\textit{score-align} + \alpha\ \textit{score-locality}$.

At high load, the queueing delay is the most critical factor. Leopard balances the load while accounting for the QoS requirements of each function. Specifically, for interactive invocations, Leopard chooses the worker with the lowest *reserved-load*, which is the amount of reserved resources needed to complete all queued interactive jobs; $\textit{resv-load} = \sum_r r.avgRT \times (r.resv\text{-}cores + mem\text{-}cap \times !preemptible\text{-}mem)$, where r is the queued request in this worker. In contrast, for batch invocations, Leopard chooses the worker with the lowest *usage-load*, which is the actual resources used to complete all queued jobs; $\textit{usage-load} = \sum_r r.avgRT \times (r.avg\text{-}CPU + r.avg\text{-}Mem)$, where *avg-CPU* and *avg-Mem* are the historical CPU and memory utilization for the function. Thus, the load balancer ranks workers differently for interactive versus batch tasks.

4.3.5 Limitations

Further improvements on Leopard are worth considering. First, resv_cpuset does not consider placement: whether the reserved CPUs should reside on the same NUMA node and hyperthreads from the same core. Second, resv_cpuset cannot support the reservation of a fraction of a core, which may be desirable for some applications [25, 37]. Finally, conflict between alignment and locality goals can make the combined load balancer score ineffective [52]; deployment in a disaggregated cluster could be a potential solution [30].

5 Evaluation

We describe a new methodology for evaluating billing (§5.1). Then, we explore four questions: *How does Leopard with NPFU perform relative to other models* (§5.2)? *Can Leopard adapt to workload changes* (§5.3)? *Does Leopard scale effectively* (§5.4)? *How does building a serverless platform from billing-aware components improve efficiency* (§5.5)?

5.1 Methodology for Billing Model Evaluation

We discuss the limitations of existing methods when applied to billing and present a billing-oriented benchmark.

5.1.1 Existing Workloads

The Microsoft Azure dataset [64] provides the most complete details on invocations in a major serverless platform, including 52,000 functions invoked 8.8 billion times over 14 days. The dataset is divided into three major components: *Invocations*, the number of function invocations per minute; *Execution Time*, average and percentile execution-times per function; *Memory Usage*, average and percentile memory usage per application. However, the Azure dataset does not contain CPU usage data or details on resource fluctuations over time. Thus, researchers often use synthetic workloads and make some assumptions about unknown information [26, 45, 65] (*e.g.*, Hermod [45] and Orcbench [31] simulate spin loops for each execution, which assumes that the CPU utilization for each invocation is 100%).

Existing serverless benchmarks measure aspects of performance and cost [20, 31, 68, 71], but rarely focus on billing, and their function suites exhibit simplistic resource usage patterns. For example, SeBS [20] contains functions with a maximum CPU usage of one. Moreover, prior work [20, 71] considering customer cost does so only for SLIM-like billing.

5.1.2 BilliBench

We introduce a new billing-oriented benchmark, BilliBench (BB). We describe BB's essential components and how it can be used to evaluate billing models.

BB Functions: We utilize the function suites collected earlier (§2.3.3) as BB Functions. Unlike previous serverless benchmarks, our function suite exhibits varied resource usage behaviors (as shown in Figures 1, 2, and 3).

BB Trace: To realistically compare different billing models, invocation traces must contain detailed resource usage fluctuations. Our key insight is that, given summarized coarse-grained trace data from major cloud providers (such as Azure), we can supplement incomplete information with detailed fine-grained data from real-world serverless functions (*e.g.*, BB functions). By combining these two sources of information, we can generate realistic traces of function invocations.

The BB Trace includes three components: functions, invocations, and resource-usage phases. A function contains a list of invocations, and each invocation contains a list of resource-usage phases; each phase specifies a duration and the resource utilization in that interval (*i.e.*, active thread number and memory size). Each function in the Azure dataset, F_{Azure}, is represented as a function in the BB Trace; for each F_{Azure}, we randomly select a function F_{BB} from the BB function suite (shown in Table 1) to provide detailed data on resource usage.

We then translate the Azure dataset into a list of function invocations, following previous work [26, 45, 64]. After that, BB represents each invocation as a sequence of resource-usage phases: for each phase, we randomly choose the number

of active threads below F_{BB}'s maximum parallelism, ensuring the invocation's average parallelism across all phases matches that of F_{BB}; next, we randomly sample the memory size from the distribution of F_{Azure} in the Azure dataset; finally, we randomly select the runtime of the entire invocation from F_{Azure}'s runtime percentile in the Azure dataset. The number of resource-usage phases is based on F_{BB}, and the duration of each phase is the total runtime divided equally among the phases. The QoS of each invocation is based on whether the corresponding F_{BB} is interactive or batch.

Given the constructed dataset, we synthesize a function F_S with Python code for the function F_{Azure} that imitates its specific resource usage phases during invocation.

The above default BB Trace leverages CPU usage behavior and QoS of BB functions. To mitigate bias from the limited function suite, BB allows users to configure the trace parameters. Users can generate a trace with a configured average CPU utilization across all functions; BB randomly selects the active thread number for each phase to maintain the overall average. Users can also configure batch task proportions. These configurable synthetic traces offer a method for evaluating billing models across diverse workloads. In our experiments, we evaluated NPFU with the default BB Trace (§5.2) and BB Traces with varying average CPU utilization (ranging from 10% to 90%) and batch task proportions (ranging from 10% to 60%) (§5.3). Each combination of these two parameters generates a unique trace.

Knob Tuning: To compare billing models with different configuration options, we configure the functions in the BB Trace with four base limits to imitate real users. We set *cpu-cap* to the function's maximum parallelism and *mem-cap* to its maximum memory usage; *preemptible-memory* is set to false for interactive functions and true for batch functions; for interactive functions (*e.g.*, ML inference and database clients), *spot-cores* is set to zero, while for other interactive functions, *spot-cores* is set to *large_parallelism* minus *medium_parallelism* for performance variability.

BB can run with simple and complex models. Models that enforce strict memory-to-CPU ratios can round-up either *mem-cap* or *cpu-cap*. Platforms that do not differentiate between interactive and batch jobs can ignore *spot-cores* and *preemptible-memory*, assuming that all cores are reserved.

BB Metrics: We compare billing models using five metrics across providers and customers. For performance issues, we measure resource utilization and throughput for providers and job completion time for customers. Customer cost (per invocation) and provider revenue (per machine) are two other interrelated metrics. We explicitly separate billing from pricing, since pricing merely shifts value between producers and consumers (*e.g.*, reducing prices by 10% lowers revenue by 10%), while better billing models can create additional value. To compare billing models, we fix provider revenue per machine per time unit by adjusting the pricing constants (*e.g.*, M and C in Table 3); keeping provider revenue constant lets us compare models using a single metric: user cost. When models allow tenants to tune both memory and CPU, we fix the C/M ratio at 9.6 to match Google Cloud Functions [8].

Baselines	Tunable Knobs				Billing Function	
	cpu cap	spot core	mem cap	preemptible	CPU Bill (per time unit)	Memory Bill (per time unit)
SLIM			✓		mem-cap / $r \times C$	mem-cap$\times M$
SIM	✓		✓		cpu-cap$\times C$	mem-cap$\times M$
SPFU	✓		✓		avg-cpu$\times C$	avg-mem$\times M$
NPFU	✓	✓	✓	✓	resv-cores$\times C_r$ +avg-spot-cpu$\times C_s$ −avg-lent-cpu$\times C_s$	mem-cap$\times M_r \times \neg P$ +avg-mem$\times M_p \times P$ −avg-lent-mem$\times M_p$

Table 3: The tunnable knobs and billing functions of baselines. Billing functions contain the price constants that providers can adjust: C for per cpu-second, M for per GB-second. NPFU includes two CPU prices for reserved CPU (C_r) and spot CPU (C_s) (Similarly for M_r and M_p) and resv-cores = cpu-cap − spot-cores. r indicates the fixed memory-to-CPU ratio in SLIM. P indicates whether the user set preemptible-memory or not.

Billing Models: We have implemented the following mainstream billing models in Leopard, as summarized in Table 3. *SLIM:* AWS Lambda, GCP, and Azure's premium plan use SLIM-like billing models, where users specify a memory limit for each function, and CPU is proportionally allocated; these limits serve as both minimal and maximal thresholds. *SIM:* SLIM without Assumption L provides independent knobs for memory and CPU limits. *SPFU*[1]: Strict PFU has no knobs, since cost is a function of usage and not knob settings; however, a platform using SPFU may still expose resource limit settings (*e.g.*, Azure's consumption plan and Cloudflare use mostly fixed resource limits), and so we optimistically assume customers set these limits as restrictively as possible, despite having little billing incentive to do so. *NPFU:* NPFU contains four knobs (*cpu-cap*, *spot-cores*, *mem-cap*, and *preemptible-mem*) and supports paying more or less when resources are borrowed or lent, respectively. NPFU aims to closely approximate SPFU while remaining profitable in practice.

5.1.3 Experiment Setup

We conduct experiments on machines with two Intel Xeon Silvers, each with 10 physical cores. We disable hyperthreading to eliminate hardware-level interference, allowing us to focus on CPU isolation mechanisms in the kernel. We use Ubuntu 22.04 with the latest Linux kernel v6.7-rc2 using cgroups V1.

We build support for each of the above billing models in Leopard. We use BB Traces based on a random sample of 1,000 functions in the BB dataset. The client initiates a task to send the invocation at the designated timestamp and wait for the response. We use one load-balancing node to route invocations to one of eight worker nodes. For each worker node, we assigned 16 CPUs and 8 GB of DRAM to serve invocations and 1 CPU to handle worker logic, including admission control and communication with the controller.

[1] In §2, we claim that SPFU is an idealized billing model, but it's not very profitable in practice as it usually provisions resources that may go unbilled.

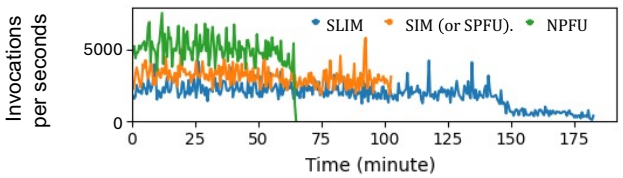

Figure 7: The throughput vary based on different billing knobs.

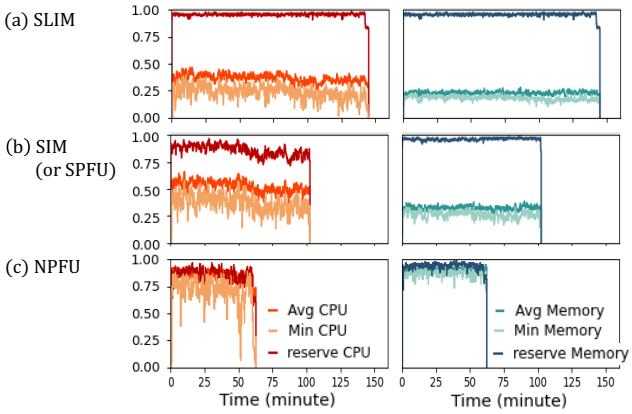

Figure 8: The utilization vary based on different billing knobs.

5.2 Evaluation of NPFU

We show that NPFU benefits both providers and users: providers obtain greater efficiency while users pay lower costs and still meet their QoS needs.

5.2.1 Provider Side

Figure 7 shows the benefits of NPFU for providers: the throughput for SLIM, SIM, and NPFU billing models in the first hour of invocations from the BB Trace. We observe that decoupling the linear memory-CPU relationship (*i.e.*, going from SLIM to SIM) leads to a 1.3x increase in throughput. Switching from SIM to NPFU provides an additional 1.6x improvement because Leopard NPFU can admit invocations when all resources are reserved but happen to be idle.

Figure 8 shows the average and minimum resource utilization in the cluster, explaining the higher throughput of NPFU. With SLIM, over 50% of CPU and 75% of memory are wasted, while SIM improves resource utilization by decoupling CPU and memory, and NPFU further increases memory utilization to 90% and CPU utilization to 80%, thanks to spot cores and preemptible memory. We observe that all billing models maintain high reservation rates: SLIM achieves near-100% CPU and memory reservation, while SIM and NPFU have lower CPU reservations due to CPU-memory decoupling. In summary, Leopard's load balancer and admission control ensure full resource rental under high load. These benefits come from the more expressive knobs of NPFU, allowing users to specify which portions of allocated resources can be borrowed without compromising their objectives. In contrast, SLIM lacks such knobs, as it makes four assumptions about resource allocation, preventing resource schedulers from sharing unused resources across invocations and leading to low utilization.

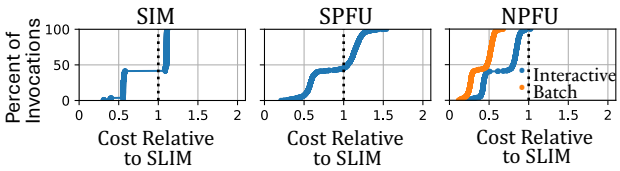

Figure 9: User cost relative to SLIM with different billing models.

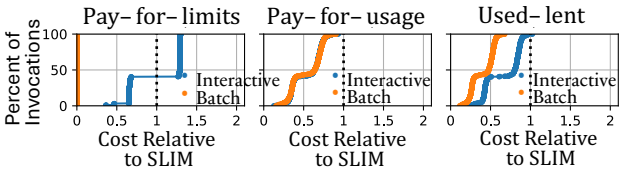

Figure 10: User cost of NPFU with different billing functions.

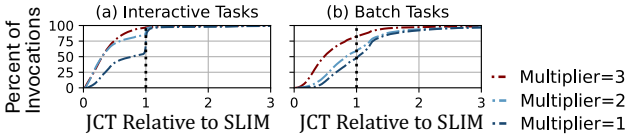

Figure 11: CDF of Job Completion Time (JCT) of invocations when running with NPFU (relative to that when running with SLIM).

5.2.2 User Side

For users, we examine both cost and performance.

User cost: To perform a fair comparison for each billing model, we adjust the unit prices of CPU and memory to ensure that the provider revenue remains the same as the SLIM. Figure 9 shows the CDF of invocation cost relative to the cost of running with SLIM. With SIM, approximately 50% of invocations save money since they no longer over-allocate CPU to meet the memory requirements of the linear model; however, other functions become slightly more expensive. For SPFU, maintaining provider revenue necessitates increasing SPFU's C and M price parameters by 1.6x; as a result, some functions cost more than 50% because SPFU distributes the cost of resource wastage across all users. Most importantly, NPFU reduces the cost of nearly every invocation, often significantly: about 40% of functions cut costs by at least half. Batch tasks save more than interactive tasks, as they pay lower rates for spare resources that interactive instances reserved. In total, the costs for NPFU clients are 66% (for interactive functions) and 41% (for batch functions) of those under SLIM.

In Figure 10, we examine the benefits of used-lent billing by comparing NPFU with pay-for-limits (like SLIM) and pay-for-usage (like SPFU). Pay-for-limits lets batch tasks run at no cost, as they did not require any reserved resources, making interactive tasks more expensive. Pay-for-usage charges interactive and batch jobs similarly and lacks incentives for users to relax their QoS requirements; it also does not charge for limits, giving users no incentive to set realistic restrictions.

Job Completion Time: We now examine whether interactive functions meet their QoS. To simulate different load levels, we repeat each invocation a *multiplier* number of times. Figure 11(a) compares job completion time (JCT), including queuing and execution time, with NPFU versus SLIM. Under

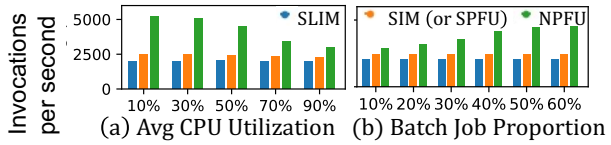

Figure 12: The throughput as the workload characteristics change.

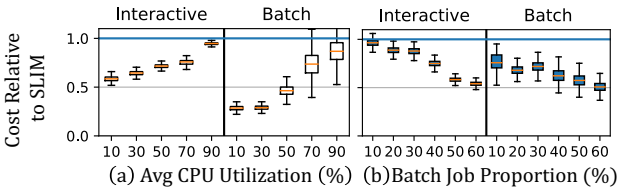

Figure 13: The cost of NPFU as the workload characteristics change.

high system load, when most invocations experience queuing, NPFU significantly reduces JCT for nearly all interactive functions. As the load decreases, the JCT for NPFU and SLIM becomes similar. A comparison with SIM (or SPFU) produced similar results. Batch tasks do not have strict latency requirements. As shown in Figure 11(b), batch tasks may take up to three times longer with NPFU compared to SLIM; however, some batch tasks can achieve shorter JCT as NPFU allows them to run when resources are not fully available.

5.3 Robustness to Different Workloads

We now analyze billing models for a range of BB Traces; we adjust CPU utilization and the proportion of batch tasks.

CPU Distributions: In a BB Trace, each invocation derives its parallelism from BB functions. In Figure 12(a), we vary the CPU utilization for each function following a normalized distribution with a given average. As CPU utilization increases for each invocation, NPFU delivers reduced throughput for the provider and smaller savings for customers, as Leopard admits extra batch tasks only when CPU utilization is below a threshold. Other billing models have unchanged throughput, as they admit tasks based on reserved resources, not usage. Still, the throughput of NPFU always remains higher than that of the other models. Figure 13(a) shows the user cost with varying CPU utilization; when CPU utilization exceeds 70%, a small number of batch invocations become slightly more expensive for customers with NPFU.

Batch Task Proportions: Previously, we assigned functions as interactive or batch based on BB functions. In Figure 12(b) we analyze how Leopard's performance changes with different batch task proportions. NPFU achieves higher throughput for the provider as the batch task proportion increases; for other models, the throughput stays constant. Figure 13(b) shows that NPFU offers greater financial benefits for users when there are more batch tasks; with more batch tasks, they can better use idle but reserved resources and boost throughput. Compared to SLIM, NPFU may not reduce the costs for a small portion of interactive tasks, as idle resources remain underutilized without sufficient batch tasks.

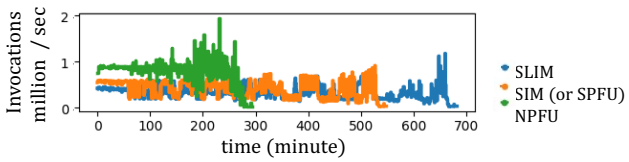

Figure 14: The provider throughput running large-scale BB Trace on different billing knobs.

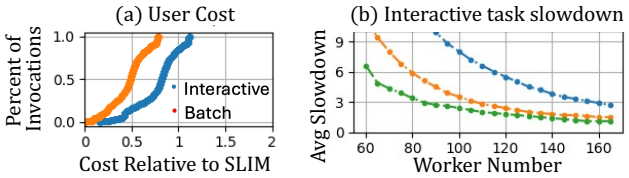

Figure 15: The NPFU's (a) user cost (b) JCT of interactive jobs running large-scale BB Trace.

In conclusion, NPFU outperforms SLIM for every point we explored in the space of CPU utilization and batch-job proportion. The benefits are greatest when interactive jobs have low CPU utilization relative to resource caps and there are many batch jobs to take advantage of those leftover resources.

5.4 Large-Cluster Simulations

To study a larger cluster, we build a simulator to evaluate different billing models on a cluster of 160 workers and use the entire first hour of the complete BB Trace (instead of only a random sample of 1,000 functions). Figure 14 shows that the simulated results on a 160 workers obtain the same benefits as a real 8-worker cluster (Figure 7).

Figure 15(a) shows that improvements in user cost with NPFU relative to SLIM are similar to that shown for small-scale experiments (Figure 9); however, about 5% of interactive functions are now up to 15% more expensive than with SLIM. Given that NPFU differentiates between interactive and batch functions, it is natural that some revenue collection shifts from batch to interactive. Figure 15(b) shows that the QoS of interactive functions is still satisfied under NPFU as the number of workers changes, with consistently lower slowdown than in other billing models.

5.5 Evaluation of Leopard Components

CPU Reservations: We evaluate whether the cgroup interface cpu.resv_cpuset and modifications to CFS of Leopard efficiently support CPU reservations. We examine three settings: (1) SLIM with CPU pinning: each sandbox is assigned a fixed CPU set and exclusively uses its allocated CPUs. (2) NPFU with weighted sharing: all CPUs are shared among active sandboxes, where interactive sandboxes with reserved CPUs have a weight of $1024 \times resv\_cpu$ and non-interactive sandboxes have a weight of 1. (3) NPFU with reserved CPU set: sandboxes with reserved CPUs are assigned to a reserved CPU set; sandboxes can access their reserved CPUs when needed but still lend idled CPUs.

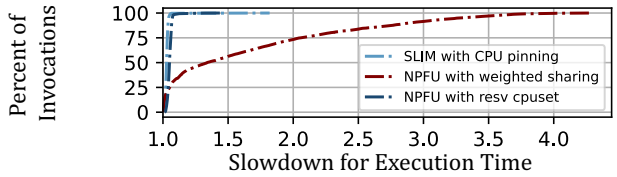

Figure 16: The CDF for the invocation execution time slowdown relative to running without contention, excluding queuing delay.

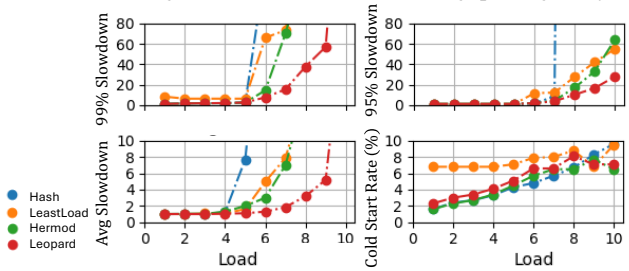

Figure 17: The slowdown and cold start rate when implementing different load balancers on Leopard.

Figure 16 shows the CDF of execution time slowdowns for interactive invocations in the BB Trace. Execution time excludes queue delay, so the slowdown is primarily due to CPU contention among co-located invocations. We observe that CPU pinning cannot support spot-cores; fair-share scheduling cannot correctly provide performance isolation for interactive jobs; and Leopard's reserved cpuset enables the spot-core while maintaining performance isolation for interactive jobs.

Load Balancer: With NPFU, we implement the following load balancers on Leopard: (i) Consistent Hashing scheduler from OpenWhisk [4], (ii) Least-Loaded Scheduler as in Kogias *et al.* [49], and (iii) Hermod [45] scheduler, which uses a greedy approach under low load and the least-loaded scheduler during high load. In Figure 17, we present the slowdown in job completion time (including queue delay) for all interactive functions in Leopard (with NPFU enabled) under different load balancers. A few key observations: First, under low load, the 99% slowdown of Least-Loaded balancer is higher compared to other load balancers since it does not account for locality (the cold start rate is depicted in Figure 17). Second, the performance of the Consistent Hashing deteriorates significantly as the load increases, as it is load-unaware. Finally, Leopard performs as well as, or better than, these common serverless schedulers across all load conditions.

6 Related Work

Improving Serverless Utilization: Existing research has explored ways to increase the resource utilization of serverless platforms. Owl [67], Golgi [50], and Jiagu [53] stress overcommitment of function instances. However, these approaches still use SLIM billing models. Overcommitment breaks the promise made by the SLIM. They rely on predicting user performance to avoid breaking the QoS of user requests, which is difficult for real workloads [17]. Other works [58,75] search for a near-optimal configuration for functions under SLIM or SIM. However, our evaluation shows that even if users choose precise resource limits, the cluster still suffers from low resource utilization. This problem arises because actual usage fluctuates both within individual invocations and across different invocations. We introduce a new perspective by formalizing existing billing models and proposing a novel model aimed at improving utilization.

Resource Harvesting: Cloud services (*e.g.*, EC2 [12]) offer on-demand and spot instances, while Harvest VMs [14] improve cost-efficiency by adjusting VM cores over its lifetime. Harvest VMs adjust CPU affinity with interprocessor interrupts, introducing high overhead (~0.1 milliseconds). These approaches are cost-effective, but VM size changes are infrequent compared to serverless workloads; supporting QoS-aware sandboxes on serverless platforms presents additional challenges. As shown in Figure 1, CPU demand fluctuates frequently, and adjusting CPU affinity with every change would incur unacceptable overhead. BigQuery [7] offers capacity and on-demand pricing, but the resources are static in a sandbox in both pricing models. Running interactive tasks alongside batch tasks with efficient resource schedulers [19,61] has been explored in multi-tenant datacenters. However, serverless platforms do not effectively support co-location, as their billing models assume that all functions are interactive.

CPU Reservation: The user-space schedulers [24,38,44,57,60] support CPU reservation by adjusting CPU affinity over time, using a centralized CPU for allocation and ensuring sub-microsecond tail latency. However, they are not well-suited for serverless environments. Their simple load balancing methods inside a task group (*e.g.*, work stealing) are difficult to maintain fairness within a group. Additionally, some require users to modify their code for process creation.

Serverless Trace Analysis: Many studies have offered comprehensive overviews of the serverless computing [33,43,51,56,69]. For example, Microsoft researchers [64] characterize serverless workloads in their production platform, revealing a clear gap between maximum and average memory usage, indicating dynamic memory usage. Similarly, FirePlace [17] shows the fluctuating CPU and memory usage of 200K micro VMs on AWS Lambda.

7 Conclusion

We introduced the NPFU billing model and built Leopard to support NPFU with kernel-level and cluster-level changes. NPFU improves throughput by more than 2x, and reduces customer costs while maintaining provider profitability. Our work demonstrates that billing models should be considered not as an afterthought, but as a central part of system design.

8 Acknowledgment

We thank Dong Du, the anonymous reviewers, and students in ADSL for their feedback. This work was supported by the NSF under the award number CNS-2402859 and the taxpayers of Wisconsin and the USA. Opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of these institutions.

9 References

[1] Kubernetes. http://kubernetes.io, August 2014.

[2] Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, August 2014.

[3] Docker Project. https://www.docker.io/, 2015.

[4] IBM OpenWhisk. https://developer.ibm.com/openwhisk/, May 2016.

[5] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/, May 2016.

[6] Common Serverless Applications Scenarios. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/scenarios.html, May 2023.

[7] Google BigQuery. https://cloud.google.com/bigquery/docs/introduction, May 2023.

[8] Google Cloud Functions pricing. https://cloud.google.com/functions/pricing, May 2023.

[9] KNIX Serverless. https://github.com/knix-microfunctions/knix/, May 2023.

[10] Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, May 2023.

[11] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[12] Amazon. Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/, November 2023.

[13] Amazon. AWS Lambda. https://aws.amazon.com/lambda/, November 2023.

[14] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 735–751. USENIX Association, November 2020.

[15] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In Proceedings of the ACM Symposium on Cloud Computing, page 263–274, 2018.

[16] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, 1.10 edition, November 2023.

[17] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan (Murali) Narayanaswamy. FirePlace: Placing FireCracker virtual machines with hindsight imitation. In MLSys 2021, NeurIPS 2020 Workshop on Machine Learning for Systems, 2020.

[18] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, page 1–15, 2020.

[19] Shuang Chen, Christina Delimitrou, and José F. Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In Proceedings of the 22nd International Middleware Conference, page 64–78, 2021.

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 475–488, Renton, WA, July 2019. USENIX Association.

[22] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 363–376, Boston, MA, March 2017. USENIX Association.

[23] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 363–376, Boston, MA, March 2017. USENIX Association.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 281–297. USENIX Association, November 2020.

[25] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. SFS: Smart OS Scheduling for Serverless Functions. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22. IEEE Press, 2022.

[26] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 3–18, 2019.

[28] Google. Google Cloud Functions. https://cloud.google.com/functions, November 2023.

[29] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. SIGCOMM Comput. Commun. Rev., 44(4):455–466, aug 2014.

[30] Zhiyuan Guo, Zachary Blanco, Junda Chen, Jinmou Li, Zerui Wei, Bili Dong, Ishaan Pota, Mohammad Shahrad, Harry Xu, and Yiying Zhang. Zenix: Efficient Execution of Bulky Serverless Applications, 2024.

[31] Ryan Hancock, Sreeharsha Udayashankar, Ali José Mashtizadeh, and Samer Al-Kiswany. OrcBench: A Representative Serverless Benchmark. In 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), pages 103–108, 2022.

[32] J.L. Hellerstein. Achieving service rate objectives with decay usage scheduling. IEEE Transactions on Software Engineering, 19(8):813–825, 1993.

[33] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back, 2018.

[34] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openLambda. In Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing, HotCloud'16, page 33–39, USA, 2016. USENIX Association.

[35] G. J. Henry. The UNIX system: The fair share scheduler. ATT Bell Laboratories Technical Journal, 63(8):1845–1857, 1984.

[36] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24, page 1039–1053, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Al Amjad Tawfiq Isstaif and Richard Mortier. Towards Latency-Aware Linux Scheduling for Serverless Workloads. In Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies, SESAME '23, page 19–26, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23, page 466–481, 2023.

[39] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 152–166, 2021.

[40] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 406–419, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, 2019.

[44] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for second-scale Tail Latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 345–360, Boston, MA, February 2019. USENIX Association.

[45] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In Proceedings of the 13th Symposium on Cloud Computing, SoCC '22, page 289–305, New York, NY, USA, 2022. Association for Computing Machinery.

[46] J. Kay and P. Lauder. A fair share scheduler. Commun. ACM, 31(1):44–55, January 1988.

[47] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[49] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 863–880, Renton, WA, July 2019. USENIX Association.

[50] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing. page 32–47, 2023.

[51] Liang Wang and Mengyuan Li and Yinqian Zhang and Thomas Ristenpart and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 133–146, 2018.

[52] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The Gap Between Serverless Research and Real-world Systems. In Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC '23, page 475–485, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 1–17, Santa Clara, CA, July 2024. USENIX Association.

[54] Renato Losio. State of Serverless 2023 Report Suggests Increasing Serverless Adoption. https://www.infoq.com/news/2023/09/state-serverless-report/, September 2023.

[55] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 303–320, 2022.

[56] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. ACM Comput. Surv., 54(11s), sep 2022.

[57] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 1–18, Renton, WA, April 2022. USENIX Association.

[58] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. Parrotfish: Parametric Regression for Optimizing Serverless Functions. In Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC '23, page 177–192, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In USENIX Annual Technical Conference (USENIX ATC '18), Boston, MA, 2018.

[60] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 361–378, Boston, MA, February 2019. USENIX Association.

[61] Tirthak Patel and Devesh Tiwari. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 193–206, 2020.

[62] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 193–206, Boston, MA, February 2019. USENIX Association.

[63] Sambhav Satija, Chenhao Ye, Ranjitha Kosgi, Aditya Jain, Romit Kankaria, Yiwei Chen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kiran Srinivasan. Cloudscape: A Study of Storage Services in Modern Cloud Architectures. In 23rd USENIX Conference on File and Storage Technologies (FAST 25), pages 103–121, Santa Clara, CA, February 2025. USENIX Association.

[64] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference, 2020.

[65] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A Scalable Low-Latency Serverless Platform. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Prasoon Sinha, Kostis Kaffes, and Neeraja J. Yadwadkar. Online Learning for Right-Sizing Serverless Functions. In Architecture and System Support for Transformer Models (ASSYST @ISCA 2023), 2023.

[67] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud. In Proceedings of the 13th Symposium on Cloud Computing, page 78–93. Association for Computing Machinery, 2022.

[68] Erwin van Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, and Alexandru Iosup. Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark. In Companion of the ACM/SPEC International Conference on Performance Engineering, page 26–31, 2020.

[69] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 443–457, 2021.

[70] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[71] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing Serverless Platforms with ServerlessBench. In Proceedings of the ACM Symposium on Cloud Computing, 2020.

[72] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 1187–1204. USENIX Association, November 2020.

[73] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 653–669. USENIX Association, April 2021.

[74] Zili Zhang, Chao Jin, and Xin Jin. Jolteon: Unleashing the Promise of Serverless for Serverless Workflows. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 167–183, Santa Clara, CA, April 2024. USENIX Association.

[75] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery.