Palette Load Balancing: Locality Hints for Serverless Functions

Mania Abdi*
Northeastern University

Samuel Ginzburg*
Princeton University

Charles Lin*
Anyscale Inc.

José M Faleiro*
Unaffiliated

Íñigo Goiri
Azure Systems Research

Gohar Chaudhry
Azure Systems Research

Ricardo Bianchini
Azure Systems Research

Daniel S. Berger
Azure Systems Research

Rodrigo Fonseca
Azure Systems Research

Abstract

Function-as-a-Service (FaaS) serverless computing enables a simple programming model with almost unbounded elasticity. Unfortunately, current FaaS platforms achieve this flexibility at the cost of lower performance for data-intensive applications compared to a serverful deployment. The ability to have computation close to data is a key missing feature. We introduce Palette load balancing, which offers FaaS applications a simple mechanism to express locality to the platform, through hints we term "colors". Palette maintains the serverless nature of the service – users are still not allocating resources – while allowing the platform to place successive invocations related to each other on the same executing node. We compare a prototype of the Palette load balancer to a state-of-the-art locality-oblivious load balancer on representative examples of three applications. For a serverless web application with a local cache, Palette improves the hit ratio by 6x. For a serverless version of Dask, Palette improves run times by 46% and 40% on Task Bench and TPC-H, respectively. On a serverless version of NumS, Palette improves run times by 37%. These improvements largely bridge the gap to serverful implementation of the same systems.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: Cloud Computing, Serverless Computing, Caching, Data-parallel processing

*Work conducted while at Microsoft.

1 Introduction

Minimizing data movement is key to the performance of distributed datacenter applications. Common applications like web API frontends and data analytics achieve this by exploiting and enforcing *locality*. For example, frontend servers often use a local cache and route requests from the same user to the same server to maximize cache effectiveness. Similarly, data analytics relies on tasks where one step outputs data that is consumed by subsequent tasks, and a scheduler places tasks so that network transfers of this data is minimized.

Recent research has shown the benefits of running distributed applications on Functions-as-a-Service (FaaS) serverless platforms (*e.g.*, [16, 20, 33–35, 47, 58, 73, 78]). FaaS offers ease-of-management and elasticity, while charging users only for consumed resources at fine-grained timescales. This gives the provider a lot of freedom to schedule, scale, and load-balance function instances. While there are many commercial [4, 6, 15, 21, 43, 64] and open-source [32, 48, 50, 61, 62, 81] FaaS offerings, existing platforms do not offer the same sense of *locality* that is available to serverful distributed applications. For example, no existing FaaS platform enables subsequent requests by the same user to be routed to the same function instance. Similarly, running data analytics on FaaS does not allow scheduling subsequent tasks that consume each other's outputs on the same instance. This limitation persists despite the basic ability of current FaaS platforms to keep state between invocations and existing efforts to explicitly add local caches to FaaS platforms [59, 70].

Because there is no generic or reliable way for one invocation to refer to state produced by a previous invocation, today's FaaS offerings are effectively 'data shipping' platforms [42, 52]. For example, caches at a web frontend implemented on FaaS would be ineffective as a given instance sees

essentially a random stream of users with few opportunities for cache hits. Similarly, data analytics tasks would need to transfer almost all outputs over the network.

Some approaches to address this problem include (1) adding faster – but still remote – key-value stores, such as Pocket [52], Jiffy [51], REDIS [68], Wukong [20], or R2E2 [33]; (2) using a serverful orchestrator to manage long serverless invocations as short-lived containers [34, 35, 86]; or (3) outright extending the interface to include workflows or DAGs [7, 10, 67, 78]. However, none of these approaches adds a generic sense of locality to a platform. This *lack of locality* has been noted as a major limitation in recent work [39, 42, 52, 71].

Our work. This paper explores a novel point in the design space by embedding locality as a first-order concern within the FaaS platform. We present Palette load balancing, which is a minimalist (and thus practical) extension to FaaS platforms. Palette exploits the fact that, in practice, virtually all FaaS platforms reuse instances of the same function to avoid "cold starts" [72, 83], and that there *is* state that outlives an invocation such as global variables and local files [8, 14, 45].

Palette extends the FaaS abstraction with optional *locality hints* to function invocations. These hints, which we refer to as *colors*, are simply an extra opaque parameter to a function invocation, and enable invocations to express affinity to both previous invocations, and to data produced by them.

The color abstraction maintains FaaS's simplicity: users (optionally) express which invocations would benefit from running in the same instance, without ever directly referring to, allocating, or managing instances. In effect, an application tells the platform: 'try to run this invocation in the same instance that you ran this other previous invocation', without having to specify the exact instance.

Interpreting colors as hints also preserves the benefits of FaaS for the provider. The provider can leverage colors to select the instance to send a request to. At the same time, because colors are hints, ignoring them does not affect application correctness. This preserves the provider's freedom to make allocation, scheduling, and scaling decisions.

To quantify the impact of locality hints on performance and efficiency, we implement the Palette load balancer and evaluate it with the open-source version of the Azure Functions Host runtime [11]. Specifically, we evaluate three representative examples from two broad classes of applications: web API frontends and serverless DAG processing.

We find that Palette recovers the performance lost in existing FaaS platforms. For web API frontends that use a local cache [17], Palette improves hit ratios by 6× compared to locality-oblivious FaaS load balancing. For DAG processing, we compare Palette to a state-of-the-art 'locality-oblivious' FaaS load balancer which frequently fetches data over the network. Palette reduces run times by 46% and 40% on Task Bench and TPC-H, respectively, implemented on Dask [22]. For NumS [29], we show Palette reduces run times by 37%
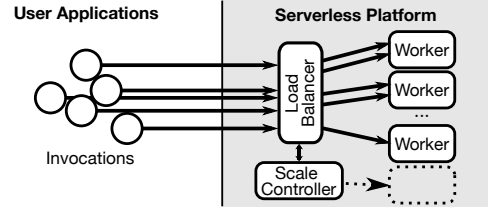

Figure 1. Main components of a FaaS platform.

on average compared to a locality-oblivious FaaS platform. Further, Palette's flexibility to choose a coloring policy can reduce run time by 2.6x in some cases. We also compare Palette to serverful implementations. Palette closes the performance gap to Dask and to NumS running on Ray [67].

Contributions. In summary, our main contributions are:
• The Palette abstraction, a simple extension to current FaaS APIs that allows users to express an application's locality and affinity properties.
• An implementation of Palette in a load balancer than enables an end-to-end performance evaluation.
• Implementations on Palette of three applications with two coloring policies, showing its generality and flexibility.
• Extensive evaluation of Palette's design and the performance and efficiency of locality hints.

2 Background and Motivation

This paper proposes an extension to the current API of serverless FaaS offerings. Thus, we start by discussing current characteristics and challenges of FaaS.

FaaS Architecture. The basic FaaS programming model comprises event-driven, independent invocations of stateless functions. A developer registers a function with the platform, together with parameters such as triggering events and data bindings [6, 13]. As in Azure Functions, we assume functions are grouped into applications, which are the unit of resource allocation. Functions from an application are loaded together, share instances, and, optionally, data that remains on an instance after an invocation.

Figure 1 shows the main components of a prototypical serverless FaaS system: a frontend, and multiple workers that run application instances in containers or VMs. In this paper we assume there is a one-to-one correspondence between an application instance and a worker, and we may use the terms interchangeably. The frontend comprises a load balancer that routes function invocation requests to the instance(s), and a scale controller that decides independently per application whether to add or remove workers (even down to 0), depending on that application's load.

A key goal of FaaS is simplicity: there are almost no configuration knobs. AWS Lambda, for example, only asks for the desired amount of memory (which also implies the number of cores) [6], and Azure Functions has no required settings [11]. A second goal is generality: there is no prescribed

application structure, and users can bring their own code in multiple languages. Use cases span, for example, API services, event processing pipelines, ETL, software testing and monitoring [28], video processing [35], data analytics [58, 73], gaming [25], and machine learning [46].

Instance Reuse. Virtually all serverless platforms keep instances around after an invocation [72, 76, 83]. This amortizes the initialization cost and avoids *cold starts.* Applications may exploit this instance reuse to store data as a simple cache [8, 12] (*e.g.*, in global variables, or in the local file system). This is useful for read-only data that is expensive to retrieve or compute. Recent works [59, 70] take this further and build proper distributed caches among instances of a function, and even exploit lingering function instances as a cheap in-memory cache for external usage [82].

FaaS Networking. Network bandwidth on current FaaS platforms is a scarce resource. FaaS providers seek to maximize the number of functions running on a single host. For example, with thousands of function micro-VMs on a single node [2], AWS implements strict per-function limits on packet rate and bandwidth. Functions also cannot use kernel or hypervisor bypass functionality, leading to high communication latencies. Further, functions are not directly addressable, which forces them to communicate through non-local storage services.

The result is that most external and cross-function interactions happen across network hops, and these are much more expensive than local data access. The authors of Cloudburst [78], for example, report latencies between 2× and 9× lower (depending on the size) for summing 10 arrays from local caches, compared to when reading from an in-memory remote KVS ([78], Figure 5). Similarly, the authors of the Faa$T cache [70] report end-to-end latencies between 1.3× and almost 5× lower for three benchmarks, when reading data from the local cache, compared to reading across the network from remote caches [70].

Lack of locality. Despite the expensive non-local data accesses, current serverless systems have no way of preserving locality across executions. This is noted in several previous works as a major limitation [39, 42, 52].

Prior work attempts to overcome the missing locality in three main ways: with fast remote storage [20, 51, 52], by specializing the system (*e.g.*, making a DAG a first-class concept [10, 67, 78]), or by using functions as (short-lived) containers (*e.g.*, gg long-lived mode [34], numpywren [73], or Kappa [86]). Unfortunately, these approaches are either inefficient, or give up some of the key benefits of the FaaS paradigm. First, fast external storage can be significantly slower than local accesses, requires work to deploy and maintain, and can get expensive [70]. Second, by specializing the system to execute only DAGs loses generality, and still can't offer locality to non-DAG applications, such as API services where unrelated requests may access the same objects. Third,

using functions as containers greatly increases complexity. A (serverful) coordinator invokes function instances that serve as workers, manages the lifetime of these functions, and repeatedly schedules tasks onto these instances. With this approach, not only do we lose the the ease of deployment, but also the cost advantage of paying only for execution time, which are two key reasons for the popularity of FaaS.

3 Use Cases

Before we describe our approach to making locality a first-class concept in FaaS, we focus on two broad classes of applications that can greatly benefit from expressing locality when implemented on a serverless environment.

Locality use case 1: Web applications and APIs. Providing business logic behind APIs for web and mobile applications is a very appealing use case of FaaS. A recent survey found APIs to comprise 28% of serverless use cases [28].

A common pattern for web services and APIs is to do extensive caching in frontend servers [5, 18, 23, 40, 55, 60]. When implemented in FaaS, one function instance can serve multiple successive requests and can cache data in between invocations. By caching connection state, database query results, social graph nodes and edges, and other data, the system responds faster and the backend systems see significantly less load. In addition, the ability to cache state in a function instance additionally reduces cost as cloud backends typically charge per query.

To make caches within a function instance effective, we *need the ability to route requests that refer to similar objects to the same instance.* Naïve policies such as session stickiness may help, but are not sufficient. For example, consider a social networking service that is currently running 100 function instances in parallel. Suppose a user sends two requests to render their social feed, first from their phone, and then from their laptop. Without support for locality, the two requests are routed randomly, with a 1% chance of being executed in the same instance. This leads to low cache hit rates. If the two requests were routed to the same instance, the cache would be highly effective. In this particular case, selecting instances by user name would solve the problem. What is needed is a generic way of expressing this, that works with the arbitrary, user-defined functions of a FaaS environment.

At scale, locality in FaaS would enable partitioning or sharding users and objects as used by large-scale caching systems [5, 18, 23, 40, 55, 60]. Partitioning can significantly increase cache efficiency, which translates to lower cost (using less memory) and lower latency (enabling higher hit rates).

Locality use case 2: Distributed data processing. Data analytics typically involves multiple processing stages with data dependencies. There have been many recent proposals of running distributed analytics on serverless [20, 34, 47, 58,

67, 73, 78]. These computations can be expressed as Directed Acyclic Graphs (DAG) of deterministic tasks with one-way dependencies. This is a model that is vastly explored in the parallel computing literature (*e.g.* [75, 77]) and many recent frameworks [1, 19, 22, 44, 85].

A DAG specifies tasks with well-defined outputs and input dependencies. A task can run when all of its dependencies have already run and produced outputs. When scheduling a DAG on a distributed system, it is important to balance two conflicting constraints: increasing parallelism, and avoiding data transfers. Doing this optimally with realistic assumptions is an NP-hard problem [75], and there are many heuristics to do this both statically and dynamically, on both parallel and distributed systems.
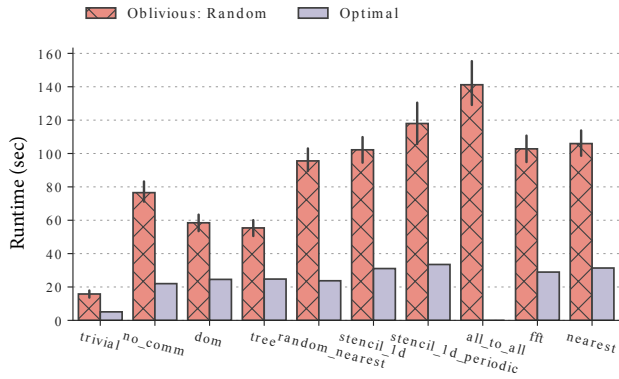

Figure 2. Performance gap of Dask on a locality-oblivious FaaS platform with a distributed in-memory cache, compared to an optimally-scheduled execution. Benchmarks are from Task Bench [77]. The optimal schedule is calculated offline based on pre-recorded runtimes and transfer sizes. Palette seeks to bridge the gap between Oblivious and Optimal via a practical change to the load balancer.

To illustrate the potential impact of locality, we implemented a serverless version of the Dask Distributed scheduler [22], on top of the open source Azure Functions host code [11], where each node in the DAG runs as a serverless invocation. Intermediate data is written to a remote object store, interposed by a modified version of the Faa$T serverless cache [70]. Figure 2 shows the results of running 10 benchmarks from the Task Bench suite [77] on four function instances, using the standard serverless load balancing strategy which is oblivious to locality ('Oblivious'). The setup is the same as the one described later in Section 7. We compare Oblivious to the runtime of an optimal schedule ('Optimal'), computed using a mixed-integer linear program [79]. Optimal schedules the tasks on the four function instances incorporating compute times and network transfer sizes from real executions. The figure shows that Opt reduces running
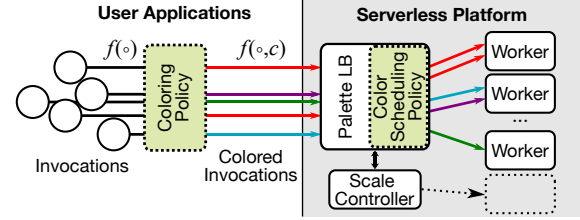

Figure 3. Palette architecture. Invocations ($f(\circ)$) carry a locality hint, or color ($f(\circ, c)$), and invocations with the same color go to the same set of instances. Users express locality via their *coloring policy*, and the platform maps colors to instances using a *color scheduling policy*.

times by more than half on eight workloads and by more than 1/3 on the remaining two workloads.[1]

4 Palette: Serverless Locality

From Section 3, exploiting locality can significantly benefit FaaS users and platforms. However, the challenge lies in exposing locality without removing the 'serverless' nature of the platform.

Colors: Locality Hints. Palette augments the interface to FaaS with a single opaque and optional parameter, which we call *color*. Like in the pFaaS proposal [71], an invocation f(params) becomes f(params, color), with the difference that a color is a *hint*. Colors have the following semantics:

> *The platform will route invocations with the same color (in a best-effort way) to the same instance or set of instances.*

Our goal is to find the smallest interface extension that enables clients to express locality so that the platform can optimize for it, without exposing details that remove the provider's flexibility to do resource management.

Colors have two important properties. First, they are opaque, and enable a strong separation of concerns that maintains the serverless nature of the platform. Users decide how to color invocations via a *coloring policy*, encoding their knowledge of locality and data dependencies, without referring to specific instances or workers. The platform, on the other hand, is responsible for routing invocations to instances via a *color scheduling policy*.

The second important property is that colors are hints, and not hard constraints. For the user, this means the application should still be correct even if the hint is ignored by the platform. Because of this, while the platform can use the hint to make decisions that improve the performance to the user, it still has the freedom to allocate and de-allocate resources. A hard constraint would get closer to an allocation request from the user, and might imply a different charging model.

Figure 3 shows the Palette architecture, in contrast to Figure 1. The namespace of colors is scoped to each application;

---

[1]We could not finish the Opt computation for All-to-All within a reasonable time window.

Palette does not introduce new data sharing or interference among different applications.
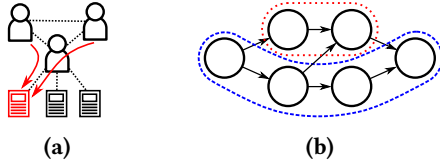

(a)   **(b)**

Figure 4. Sketches of coloring policies for serverless (a) social network and (b) DAG processing. In (a), requests referring to the same post get the same color (*e.g.*, post_id). In (b), chains of tasks in the DAG get the same color, such that most inter-task transfers land in the same node.

Using Colors. Coloring executions can improve the performance of functions in the two use cases from Section 3. Figure 4 provides intuition for possible coloring policies (we describe these policies in Section 6). Figure 4a shows requests from two users in a social network to a post by a common friend. If the two requests use the post_id as the locality hint, and get executed in the same instance, the second request can potentially read the post from a local cache. Figure 4b shows a simple DAG, where nodes represent tasks, and arrows, data dependencies. If there are two instances to run the middle tasks in parallel, coloring the tasks in the chains as shown minimizes the cross-node transfers.

There are of course applications which will not benefit from locality. These include truly stateless applications, where one invocation does not leave any state behind that is relevant for future invocations; and applications where there may be caching of data (such as an inference model that has to be loaded from storage), but where this data is the same for all invocations. Being optional, not using Palette hints in these cases does not impose any extra costs.

5 Implementation

Palette is designed to be easy to deploy into existing FaaS platforms. The API for application writers is simply an optional extra parameter in an invocation. For HTTP triggers, for example, we implement colors as a locality URL parameter, which is trivial to add from a client, and to interpret by the load balancer. The main questions for the provider when implementing Palette load balancing are the color scheduling policy (Figure 3) for how to map colors to instances, and how to potentially use colors to help with autoscaling.

Load Balancing. We implement Palette as a custom load balancer in Python that receives HTTP-triggered requests and sends them to workers running the open source version of the Azure Functions Host [11]. Our load balancer is integrated with the scale controller (Figure 3).

For the web API case, we run functions in unmodified functions host instances (§7.1). For DAG processing, we use a modified version of the Faa$T serverless cache [70].

	Hashing	Bucket Hashing	Color Table (LA)
Summary	$I(c) = \mathcal{H}_I(c)$	$I(c) = BT[\mathcal{H}_B(c)]$	$I(c) = LA[c]$
State	-	$O(B)$	$O(c)$ (capped)
LB	poor	better	best

Table 1. Comparison between the three color scheduling policies. $I(c)$ is the instance for color c. $\mathcal{H}_I$ and $\mathcal{H}_B$ are hash functions onto instances and buckets; $BT[\circ]$ and $LA[\circ]$ are bucket and color table lookups. We set both B and the cap for the LA table at 16,384.
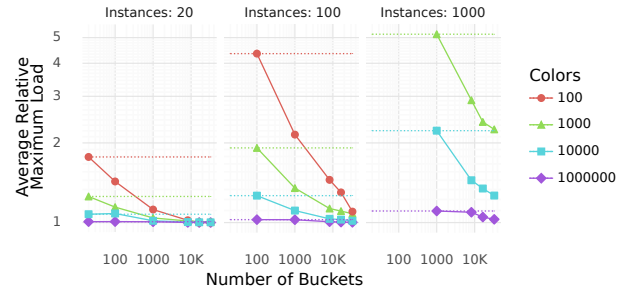

Figure 5. Relative maximum load (maximum / average colors per instance) for Bucket Hashing, for different numbers of instances, colors, and buckets. Results for each setting averaged over 20 simulations. The dashed lines show the relative maximum load for simple hashing.

Color Scheduling Policy. We implement three color scheduling policies in Palette, summarized in Table 1, with different trade-offs: (consistent) hashing, bucket hashing, least-assigned color. The goal of these policies is to map a color (received from the user invocation) onto an instance of the application. This mapping is kept by the load balancer per application. For the three options we assume the Palette load balancer has an up-to-date list of the application instances.

Hashing: the simplest color mapping policy is hashing: it requires no extra state besides the current list of instances, which the load balancer already has. The downside is that hashing is equivalent to random assignment of colors to instances [66], and produces load imbalance that can significantly impact the runtime of functions [57]. In our implementation we use Consistent Hashing [49] as it minimizes invalid mappings upon membership changes.

Bucket Hashing (BH): one approach to alleviate the load imbalance of hashing while bounding the storage needs is to introduce an indirection. We hash colors into a fixed set of B *buckets*, and then assign buckets to instances in a way that improves the load balance. While the optimal such assignment is NP-hard, a simple greedy algorithm that assigns a bucket to the instance with the least colors is a 2-approximation [41]. Variations of bucket hashing are used, for example, in Dynamo [23] and Redis [69].

Figure 5 shows the relative worst-case number of colors (maximum / expected) for simulations of the bucket hashing algorithm, averaged over 20 runs each, with different numbers of instances, colors, and buckets. For comparison, the figure also shows the maximum load for simple hashing. We see that for more than 1,000 colors, and a number of buckets 10, 000, we can keep the relative load ≤ 2, and often much closer to 1. In our implementation we choose the same number of buckets as Redis, 16,384. [2]

The load balancer keeps a fixed bucket table (BT) with B entries per application. It also keeps an approximate count of colors recently mapped to each bucket, using count distinct sketches (HyperLogLog [31]). More specifically, it starts a new HLL sketch every 30 minutes, while keeping a sketch for the previous 30 minutes. Every 30 minutes the LB merges the two last HLL sketches, and swaps buckets among instances until the maximum relative number of colors per instance is below a threshold, which we set to 2 according to Figure 5.

Color Table (Least Assigned): The last color scheduling policy we use is a table explicitly mapping colors to instances. When a new color arrives, the load balancer chooses the instance with the least assigned colors, breaking ties arbitrarily, and stores the mapping in an "LA" table. When the scale controller adds new instances, they naturally get new assignments; when it removes instances, colors from the removed nodes are distributed using the same policy.

To avoid this table growing without bounds, we evict mappings from this table using LRU. Since colors are hints, this has no effect in correctness, even though it may affect performance. This policy achieves the best load balancing, but is not suitable for applications with very large numbers of colors. To keep the memory usage similar to that of BH, our implementation caps the number of colors to 16,384, and truncates the color names at 32 bytes. With this setting, we use a maximum of 512KB of data per application.

Each application in our system can use these policies independently of other applications. The user chooses one of the policies upon registering the application. In practice, the real choice lies between Bucket Hashing and Least Assigned. Applications with more than 16K colors should choose BH, while applications with less colors that are sensitive to load imbalance should choose LA.

In addition to these three policies, we also implemented two baseline policies for comparison: oblivious, which always selects a random instance for each invocation, and oblivious round-robin, which ignores locality, but sends requests to instances in a round-robin fashion, to improve load balancing among instances.

Scaling. In this paper, the Palette load balancer assigns colors to existing instances, and scaling decisions are done separately by the scale controller. The only assumption we

---

[2]For these simulations we assume at most one instance per color, and it does not make sense to have less buckets or less colors than instances.

make in our prototype and evaluation is that there is a single active instance per color at any time (one instance can still have several colors). This is a point in the design space that is easy to implement and to reason about for the client, and is compatible with existing scaling policies (*e.g.*, [27, 36, 87]) with little to no modification. Scaling in and out will change the mapping of colors to instances, and locality – but not correctness – can suffer for colors that move. Other designs are also possible, for example, lifting the restriction of one instance per color, which can prevent hot spots, but also diffuses locality. We leave a deeper look at the interaction of the coloring and scaling policies, including the implications of instance churn, color rebalancing, and the use of colors as hints for rapid autoscaling, for future work.

5.1 Integration with the Faa$T Serverless Cache

For efficiently implementing the DAG use cases, we used a modified version of the open source Faa$T serverless cache [70] integrated with the open-source Azure Functions host. Both Faa$T and OFC [59] implement distributed object caches among multiple instances of a serverless application, while maintaining the caches for different applications independent and isolated.

The two requirements for our application are that (i) an object generated at a particular worker stays cached on that worker until it is evicted or the worker is stopped, and that (ii) other instances can find the object. The first avoids pushing objects to other nodes, and the second guarantees that we have a hit in a peer cache rather than a miss to the backing store. OFC satisfies these requirements, as it uses RAM-Cloud [63] as its in-memory cache. Faa$T originally does not, as it determines the home location of an object via consistent hashing of the object's name. We modified Faa$T by adding an optional hashing key to the object names. When the name of an object has a prefix separated by a token string ('___'), the cache uses this prefix as the hashing key to determine the home location. Redis uses the same approach to enable multi-key transactions [69].

The client DAG application, when specifying input (i) and output (o) objects of task f, adds the color of the producing tasks to the names of the objects (*e.g.*, i_c), in addition to adding the color to the task itself (cf):

$$f_{cf}(i_{c1}, i_{c2}, ..., i_{cn}) \rightarrow o_{cf}$$

When submitting this task, the Palette load balancer determines the instance $I(cf)$ that will run the task, and translates the colors of the inputs and outputs so that they hash to the correct instances. Because the consistent hashing function is the identity function when the argument is the name of one of the members of the ring (*i.e.*, $CH(I(c)) = I(c)$), Palette simply translates color ck to $I(ck)$, and the invocation above becomes:

$$f_{cf}(i_{I(c1)}, i_{I(c2)}, ..., i_{I(cn)}) \rightarrow o_{I(cf)}.$$

6 Application Coloring Policies

To evaluate the usefulness and effectiveness of Palette, we implemented three representative applications of the two use cases from Section 3.

6.1 Social Network

As a representative of a serverless API to a web application, we modified a FaaS implementation of the Social Network application from the DeathStar benchmark [38], obtained from the authors. The application is implemented in Python, as a collection of microservices that perform basic functions such as fetching user profiles, timelines, and posts. The data for the site is stored in MongoDB.

We augmented the function implementations with an in-memory read-only LRU cache that stores objects such as posts, images, and friends lists. We implement the cache in the function code via a global variable, which is part of the ephemeral state that remains in the instance local memory after function invocation. The cache is thus shared across successive invocations until the platform reclaims the instance. This is a pattern recommended by FaaS providers [8, 14], and requires no support from the platform.

To use Palette, the social network client adds color hints to requests, like in Figure 4a. Specifically, `get_user_timeline` uses the `user_id` as the color; each subsequent `get_post` and `get_media` calls are colored, respectively, with the post id and object id. This causes the Palette load balancer to direct all calls that fetch a specific object to a single instance, and enables the client to indirectly control the partitioning of the aggregate cache space, even though Palette is agnostic to the semantics of the application.

In Section 7.1, we evaluate the aggregate hit ratio of this coloring policy against two baseline policies: Oblivious (which routes requests to random instances) and Bucket Hash.

6.2 DAG Computations

6.2.1 Coloring DAGs. Palette provides great flexibility to express DAG computations in serverless. Intuitively, we want to color DAGs nodes so as to reduce unnecessary data transfers among nodes. We implement two DAG frameworks on FaaS, where each node of the DAG is one invocation, and their outputs are cached on the workers that run each node. We use the Faa$T serverless cache (§5.1), so that an invocation can fetch inputs from the remote workers where they were produced, if not evicted yet.

We describe two DAG coloring policies for Palette: coloring the DAG from first principles, and a nearly-transparent policy that reuses existing frameworks' schedulers. We compare these two approaches for a FaaS implementation of Dask [22], and demonstrate the flexibility of the second for an additional system, the NumS linear algebra library [29].

Chain Coloring. When executing a DAG in parallel, we want to exploit parallelism opportunities, and reduce data transfers. Because of the separation of concerns that colors create, we can treat the coloring of a DAG as a parallel scheduling problem with an unbounded number of workers (where each worker is a color), and leave the mapping of colors to instances to the serverless platform (cf. §5).

There is a vast literature on static scheduling of parallel DAG computations [75], with many applicable algorithms. As an example, we chose a simple heuristic which we refer to as Chain Coloring: we partition the DAG into chains (simple paths), and assign a separate color to each chain. Optimal chain partitioning of a DAG (minimum number of chains) takes $O(n^3)$ [37]. Instead, we use a simpler greedy partitioning based on recursively finding and removing longest paths on a topologically sorted DAG (see [74], algorithm B). This algorithm runs in linear time $O(v + e)$, and tends to get close to the minimum number of chains.

Chain coloring has three useful properties: (i) every node in a simple chain (a path in the DAG with no fan-ins or fan-outs) will have the same color (meaning no external data transfers); (ii) no nodes that can be executed in parallel will have the same color (giving the platform the opportunity to maximize parallelism); and (iii) for fan-outs and fan-ins, only one of the chains "continues". Chain coloring can achieve good performance, but not always: because it maximizes parallelism, if network transfers are too expensive their cost can dominate the execution.

Bring your own scheduler – Virtual Workers. An alternative to manually coloring a DAG is to reuse the scheduling policies that frameworks already have. Typically, schedulers in frameworks like Dask work with a set of workers that dynamically connect to it. They keep detailed information on the load, memory usage, and sets of stored objects for each worker. With this information they can produce a dynamic schedule that takes into account locality, load balancing, and memory usage of the cluster.

Coloring of tasks can be implemented by introducing the concept of "virtual workers". These workers do not have to map to actual instances in the system, but we can assign colors to each of them and the tasks they produce. From the point of view of the scheduler, each virtual worker is an actual worker, and it can continue to make scheduling decisions under that assumption. In reality, each virtual worker colors all of its invocations with its own color. This, in principle, is similar to user-level, or green threads,which appear as threads to applications, but are mapped onto hardware threads by the OS. To show the generality of this abstraction, we implement this on two DAG processing systems, Dask and NumS, and evaluate their performance in in Section 7.

6.2.2 Dask. Dask [22] is a popular Python library for parallel task-based DAG computing. We modified Dask to run on FaaS by first factoring out all intermediate state onto a remote object store. We then create and register with the FaaS framework a function that is essentially "eval": it receives the

serialized code to be run, references to inputs, and stores its output on the remote store. We use the open source version of Azure Functions [11], and use our modified version of the Faa$T cache to interpose on all accesses to the object store.

We implement two versions of Dask: (a) with a custom scheduling policy that performs chain coloring, and (b) with an unmodified scheduler using virtual workers. In case of the chain coloring policy, we modify the scheduler to color the graph as described in Section 6.2. Based on the assigned colors, the Dask scheduler then dispatches the requests to the Azure Functions backend. Providing colors as part of the requests allows the backend to efficiently retrieve the inputs from the instances that produced them.

For the virtual workers approach, we extend the worker interface in Dask and introduce a shim that acts as virtual workers. These pretend to be regular workers to the scheduler, but instead send each execution request to Palette as a serverless invocation. This shim layer assigns a color to each virtual worker and all invocations from that worker get 'painted' with the worker's color. Tasks scheduled for that virtual worker will then be routed by Palette's load balancer on to the same serverless instance(s). This approach transparently translates the scheduler's notion of locality to Palette's implementation of locality.

6.2.3 NumS.

The NumS [29] numerical library enables NumPy code to seamlessly run on distributed systems by scaling operations horizontally. It translates linear algebra operations written in the NumPy API into a DAG of smaller computation tasks. Currently, NumS uses Ray [67] as a backend for running the computations. The user registers a pre-provisioned fixed-size Ray cluster with NumS and the cluster exposes a set of "devices" on which NumS can schedule tasks. NumS has sophisticated algorithms to map blocked array operations onto the devices, to balance locality and parallelism. The actual computation is done asynchronously and only the retrieval of an output results in a blocking call.

We implement a Palette backend on top of Azure Functions running with the Faa$T cache, similar to the Dask implementation. We use the virtual workers approach: the NumS scheduler remains unmodified and we add a shim that interacts with Palette. This shim consists of only 114 lines of Python, the other part of the implementation being the code that runs as FaaS, with about 135 lines of Python. The serverless nature of this backend obviates the need to register a provisioned cluster of resources with NumS. This backend exposes a flexible number of devices to NumS, and internally does a one-to-one mapping of the deviceIds chosen by NumS for each operation onto Palette colors.

We let the scheduler pick from a large number of devices that are mapped to colors and as such may or may not map to actual instances on the backend. Output objects are also assigned the color of the instance that produces them to

make it possible for our backend to retrieve it from the correct one when required. We maintain the model that the existing Ray backend provides, of asynchronous computations and retrieval of outputs values only when required. To achieve this, we use "futures" [65] to encapsulate the state of dispatched tasks which are otherwise blocking calls to Azure Functions. The shim seamlessly handles the retrieval of results from these futures when required and ensures that the instances start executing tasks only when the dependencies have been satisfied by prior tasks. As a result, our implementation requires no changes to the NumS API and can run existing programs without modifications. We show in Section 7.2.4 that our use of the virtual workers approach produces schedules that are competitive with NumS on a serverful backend (*i.e.*, Ray).

6.3 Discussion

Dynamic Policies. While we do not evaluate these, clients can use colors to implement interesting dynamic policies. For a DAG, for example, in the case of a fan-in, we can defer coloring the downstream node until we know the sizes of all inputs, and choose the color of the largest input. Alternatively, we can implement prefetching. Suppose a *blue* task b_2 depends on a *blue* task b_1 and on a *red* task r_1, and that r_1 finishes first. The scheduler can create a dummy blue task b' that only depends on r_1. b' can run immediately, and has the only effect of causing the output of r_1 to be fetched by the instance running blue tasks. In the case of a fan-out, we can also use the same technique of dummy tasks to cause the output of the fan-out source to be pre-fetched by all the downstream colors.

Who colors? Palette enables flexible and dynamic policies, and, when applicable, using virtual workers (cf. § 6.2) makes this transparent. We expect that in many cases framework and application developers, and even query planners, but not end users, will generate coloring hints.

7 Evaluation

We evaluate our prototype of Palette on: a web application; the Dask Python DAG execution engine [22] with three different workload sets; and the NumS distributed framework which implements the NumPy API.

Setup. Unless otherwise noted, we run our experiments using a client VM, and a cluster of up to 48 worker VMs. The client VM is an Azure Standard E8asv4 VMs with 8 vCPUs (AMD EPYC 7452) and 64GiB of RAM, running Ubuntu 18.04. Each function worker runs on an Azure Standard D4sv3 with 4 vCPUs (Intel Xeon E5-2673) and 16 GiB of RAM, running Ubuntu Linux v20.04. The bandwidth between all VMs is 1.86Gbps. For all of the DAG applications, we select the problem sizes so that the output sizes of individual nodes in the DAG fit within serverless function memory limitations. To approximate current non-premium commercial serverless

offerings (§2), we throttle the network to 1Gbps. For easier understanding of the results, we limit each function worker (and Dask worker) to a single vCPU. When using the Faa$T cache, we set the cache size per function instance to 8GB. The goal of this paper is to evaluate the potential benefit of locality for FaaS, and not the performance of the caching system, which was well explored in the literature [59, 70]. As such, we avoid cache evictions, and configure Faa$T to keep intermediate data only in memory, instead of writing it back to storage.

7.1 Web Application

In our first experiment, we augment a serverless implementation of a Social Network web application benchmark [38] with a local, in-memory cache for objects (§6.1).

Methodology. The goal of this experiment is to demonstrate the advantage of locality hints to enable cache pooling. We compare two color scheduling policies: Oblivious (standard FaaS scheduling), and Palette with Bucket Hashing. The client uses the object id as the locality key (§6.1).

We run the social network client and the MongoDB backend in our client VM, and use 1-24 function worker VMs. The goal of this experiment is not to measure the performance or load capacity of the components, but to examine the aggregate hit ratio on the local caches.

We preload the MongoDB with the socfb-Reed98 [30] social network graph and 20 posts per user. Each post has a random text with size drawn from a uniform distribution between 64B and 1KB, and a random number of media uniformly distributed between 1 and 5. The media files have an average size of 1MB, with 25th, 50th, 75th, and 100th percentiles respectively 62KB, 1MB, 2MB, and 8MB. We obtained this distribution by scrolling through Instagram in September 2021 and recording the sizes of 1465 images downloaded by a desktop browser. Our client selects users from a Zipf distribution with parameter 0.9 and sends 72,000 ReadHomeTimeline and ReadUserTimeline requests (with 50% ReadHomeTimeline and 50% ReadUserTimeline). We replay this same trace in all the social network experiments. Since we were just interested in the benefits to locality of the local caches, we emulate a read-only workload. The resulting trace has 2.6 million accesses to 1.1 million unique objects, which comprise a total of 115GB of data. We further assume requests are close enough in time to avoid application instances from being unloaded for inactivity.

Finding 1: *Locality hints effectively allow a serverless web application to transparently partition its local caches.* Figure 6a plots the hit ratio in the in-memory local caches, aggregated across all instances, and all request types, as a function of the number of instances. As the number of function workers increases from 1 to 24, we find that Oblivious' cache hit ratio stays around 4%. This is expected: as similar request gets routed to different caches, Oblivious has a high rate of cold
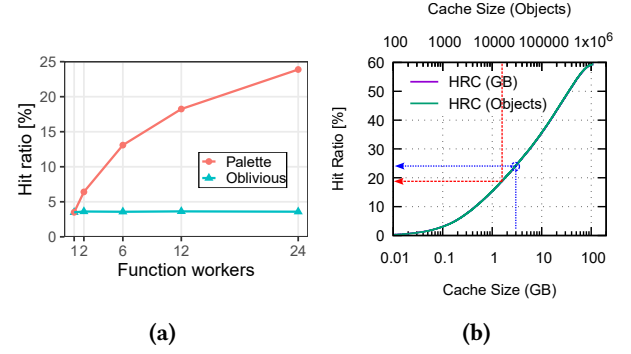

Figure 6. (a) Cache effectiveness in the Social Network benchmark, comparing Oblivious and Bucket Hashing color scheduling policies. (b) Simulated hit ratio vs all cache sizes for ideal LRU cache with the Social Network workload. With 3GB of space we achieve ∼ 24% HR (blue dotted arrow), but if limited to 16K objects, we only achieve ∼ 19% HR (red dashed arrow).

misses, and wastes a lot of the aggregate cache space with redundant copies of the most popular objects. In contrast, we find that Palette's cache hit ratio increases from 4% to 24%. This finding is aligned with prior work that documented the benefits of partitioning for caching effectiveness [5, 18, 23, 40, 55, 60]. Our evaluation shows that locality hints enable similar effectiveness in serverless applications.

Finding 2: *For applications with a large number of colors, Bucket Hashing achieves the best trade-off between space and load balancing.* As discussed in §5, Palette's Color Table can achieve optimal load balancing but caps the number of colors to 16,384. This means that the 16,385-th color leads to an eviction from the Color Table (LRU). The next time that evicted color is accessed, Palette does not remember its previous assignment which means that a cache hit for this color's objects is unlikely. The Color Table limits the effective cache size to 16k objects. Figure 6b shows a complete hit ratio curve for this workload. The blue dotted line shows the hit ratio of our experiment with an aggregate cache size of 3GB. If limited to 16K objects, the hit ratio is limited to less than 20% (red dashed line). More generally, a Color Table has to grow in proportion to the aggregate cache size not to become the limiting factor for the hit ratio. In contrast, Bucket Hashing can scale to any cache size and achieves 24% hit ratio, which means that it partitions the cache size near perfectly: with N instances, it achieves an effective aggregate cache size $N\times$ larger than a single instance.

The color table limitation here is an example of what happens when the platform cannot heed the advice of the color hints: performance, but not correctness, may be affected. Figure 6b also shows that only remembering 1,000 colors would lead to a hit ratio of less than 5%.
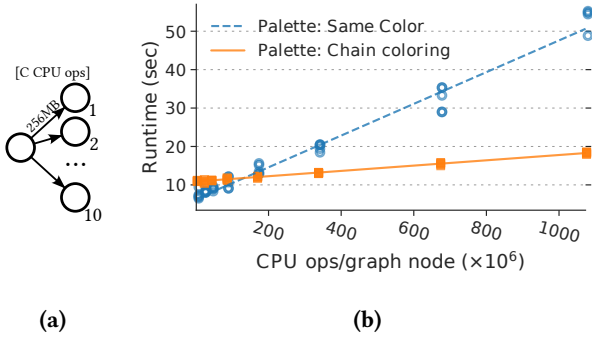
Figure 7. (a) Fanout test DAG; and (b) the total time for two coloring policies as the CPU cost increases relative to the network transfer cost.

7.2 Data Analytics

We evaluate serverless data analytics on two systems, Dask [22], and the NumS distributed framework which implements the NumPy API [29]. For the coloring policy we use either the chain coloring or virtual workers (cf. §6.2).

We compare Palette to Oblivious which emulates state-of-the-art FaaS systems that cache objects in far memory [51, 52] via a fast protocol to access objects in other workers' caches. Our baseline is a serverless, unmodified version of Dask and NumS. In addition, we use Dask as a way to validate a key implementation choice: by comparing the Consistent Hashing and Color Table color scheduling policies, we can separate the benefits of locality and better load balancing.

We use multiple sets of benchmarks. First, a synthetic microbenchmark helps us explore a key tradeoff that will help in understanding subsequent evaluation results. Second, we use Task Bench [77], which is a benchmark for evaluating parallel and distributed programming runtimes. Third, we use TPC-H 3.0 [80], which consists of a suite of business oriented large data queries with a focus on queries with a high degree of complexity. In both benchmarks, we focus on the end-to-end run time, i.e., the time until the last task/dependency has completed and the result is returned. Our figures show barplots with standard error indicated by black lines.

7.2.1 Dask Microbenchmark.
Using the Task Bench framework, we create a small DAG, which we call `fanout`, shown in Figure 7a. The DAG has a single task whose 256MB output is consumed by 10 other parallel tasks. For each experiment each of the (10+1) task runs C CPU operations, and we vary C from 2^{20} to 2^{30} operations per task.

We compare two extreme coloring policies for this DAG: chain coloring, which maximizes parallelism in a fanout, using 10 colors; and a 'Same Color' policy that maximizes locality and colors all 11 nodes with the same color. We run the experiment with Palette's Least Assigned color scheduling policy, with a total of 10 workers with one vCPU each, and 5 runs for each setting.

Finding 3: *Palette allows great flexibility in the choice of the coloring policy by the application and this flexibility helps*

improve run times. As shown in Figure 7b, with few CPU ops per task, the gains from parallelism do not compensate for the network latency, and 'Same Color' performs better. However, 'Same Color' schedules all tasks on the same worker, and their makespan increases linearly with the per-task CPU demand. Quickly it becomes better to shift to a policy that favors more colors, and more parallelism. Dask's native policy (which we use when using the Virtual Nodes coloring policy), tends to favor locality, and typically colors fanouts with the same color.

7.2.2 Task Bench on Dask.
Task Bench is a configurable set of DAG-based benchmarks. We consider two different configurations: the first has balanced computation and network transfer times (Fig. 8a) and the second one stresses computational resources, which leads Dask to distribute tasks across more workers (Fig. 8b) . Within each configuration, we order benchmarks by how frequently their tasks require inter-worker data transfers, which roughly corresponds to the density of edges in the computation DAG. The "trivial" and "no_comm" benchmarks require no inter-worker transfers whereas almost every task in the "fft" and "nearest" benchmarks requires inputs from a task on another worker.

For Palette, we use chain coloring as coloring policy, and compare four different color scheduling policies against serverful Dask (unmodified). The four serverless implementations differ along two orthogonal dimensions: whether they implement locality (Oblivious vs Palette) and how well they load balance computational load across workers (table to the right of Fig. 8a). For Oblivious, load balancing is implemented by placing function executions in a round-robin fashion on workers. For Palette, load balancing is implemented via the least-assigned (LA) Color Table approach (§5), whereas Palette with consistent hashing (CH) is less load balanced.

Finding 4: *Locality hints enable serverless analytics to fully leverage a local cache, outperforming locality-oblivious far-memory solutions by 46%.* Fig. 8a shows that both versions of Palette outperform Oblivious on every benchmark. On average, Palette reduces runtimes by more than 46% compared to Oblivious, independent of load balancing.

Finding 5: *Palette is competitive with Dask for computation DAGs with many inter-worker data transfers.* The left half of benchmarks in Fig. 8a has few inter-worker data transfers and the gap between Palette and serverful Dask remains significant. In contrast, in the right half of benchmarks, with many inter-worker transfers, the gap is less than 25%. This happens mostly because of the extra serialization overhead of our Palette implementation. In serverful Dask, workers only serialize an object when transferring to another worker. Our prototype serializes every object, and this happens on the critical path. The serialization overhead is not fundamental, and is a potential target for optimization.
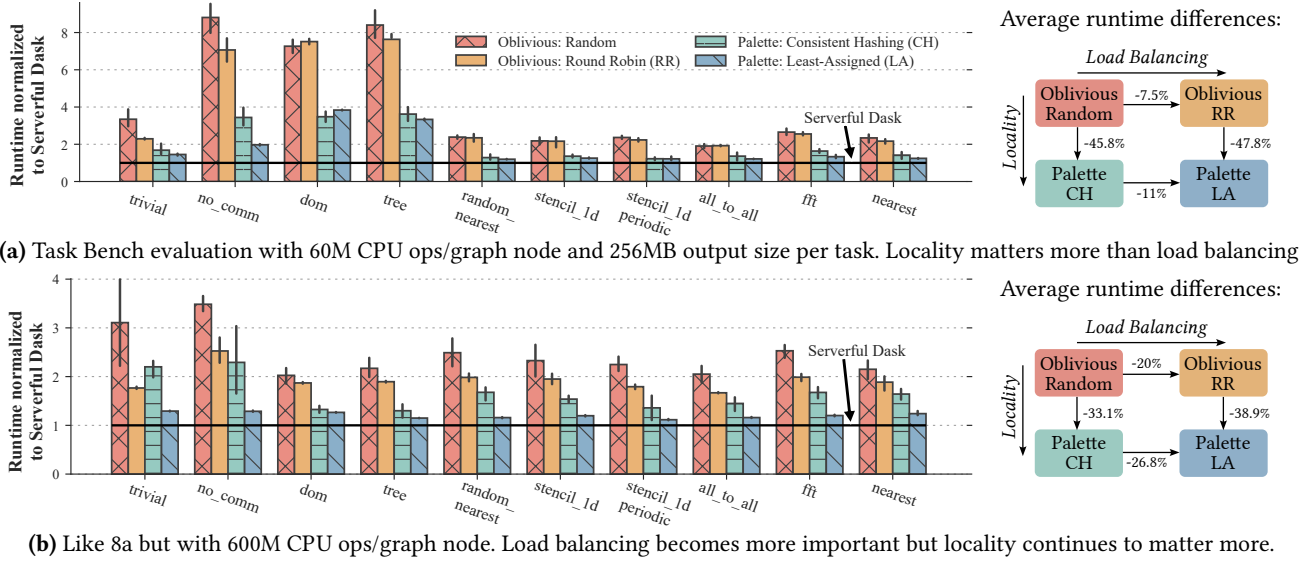
(a) Task Bench evaluation with 60M CPU ops/graph node and 256MB output size per task. Locality matters more than load balancing.

(b) Like 8a but with 600M CPU ops/graph node. Load balancing becomes more important but locality continues to matter more.

Figure 8. Task Bench results comparing variants of Palette to variants of the state-of-the-art FaaS load balancer (Oblivious), and to a serverless and unmodified version of Dask. The four serverless variants use the chain-coloring scheduler (§6.2).

Finding 6: *Palette is competitive with Dask for tasks that are compute intensive and require distributing across multiple workers.* Fig. 8b increases each task's computation load by 10× compared to Fig. 8a. This means that it becomes more important to parallelize work by distributing it across workers, which plays into the strengths of a serverless platform and Palette. Specifically, even the left half of benchmarks require some cross-worker transfers so that Palette's overheads become less relevant. Overall, the runtime of Palette with a least-assign Color Table is now within 15% of Dask on all benchmarks. We also observe that the impact of load balancing decisions becomes more important: we see a more than 20% runtime difference between Random and Round Robin, and Consistent Hashing and Least Assign, respectively. This shows the importance of supporting a Color Table in Palette.

7.2.3 TPC-H on Dask. TPC-H contains ad-hoc queries and concurrent data modifications selected for their industry relevance. Compared to Task Bench, TPC-H queries have significantly larger and less regular DAGs. We consider all 22 queries, with objects sized at 2GB and divided into 256MB blocks. We run the queries on serverless Dask using the virtual node coloring policy (there was no significant difference with chain coloring, other than the fact that VN used less colors, because of some fan-outs). We use two color scheduling policies, Oblivious Round Robin and Palette Least Assigned, and run on the same cluster of 48 function workers. Figure 9 shows the end-to-end time of all queries, normalized to the runtime of serverful Dask on the same nodes.

Finding 7: *Across all queries, Palette is on average 40% faster than Oblivious, and significantly closes the gap between far memory systems and serverful Dask.* We find similar reasons as in the Task Bench results: RR spends significantly more time transferring data, as most inputs are from remote

caches, and most outputs must be transferred to another cache which owns the object. Across all queries, the median RR query transfers over 5.9 times more data over the network than Palette. In this sense, Oblivious Round Robin is similar to FaaS data analytics that use far-memory storage, such as Pocket or Jiffy [51, 52].

Compared to serverful Dask, in 9 out of the 22 queries Palette is within 15% of the performance of Dask. In 5 of the queries Palette is more than 75% slower than Dask (queries 3, 4, 10, 12, and 17). These queries are some of the ones with the largest amount of data transfer. Serialization costs, as well as queueing in the Python async io event loop are responsible for most of the differences relative to Dask.

7.2.4 NumS. Methodology. For NumS, we compare the existing Ray backend with the Palette implementation, evaluated with three color scheduling policies: Oblivious Random, Oblivious Round Robin, and Least Assigned. We run each of the backends with equivalent resources. In the case of Ray, it is deployed on Kubernetes [54] with a VM SKU which has 16 vCPUs and 64GB of memory. 16 such VMs were assigned for running the Ray workers. We run the three Palette color scheduling policies on the same VM SKU, with a maximum of 16 workers available to them for a fair comparison.

We run three benchmarks. The first is a logistic regression workload using Newton's method on the Higgs boson dataset [26] (LRHiggs). We describe the individual phases of this workload using a code snippet as shown in Listing 1. The first two phases involve data movement and manipulation, and the last two phases are computation-heavy, where the model is fit and a prediction is made.

The second and third benchmarks perform dense square matrix-matrix multiplication with 2GB and 16GB (MMM-2GB and MMM-16GB) of synthetic data, respectively.
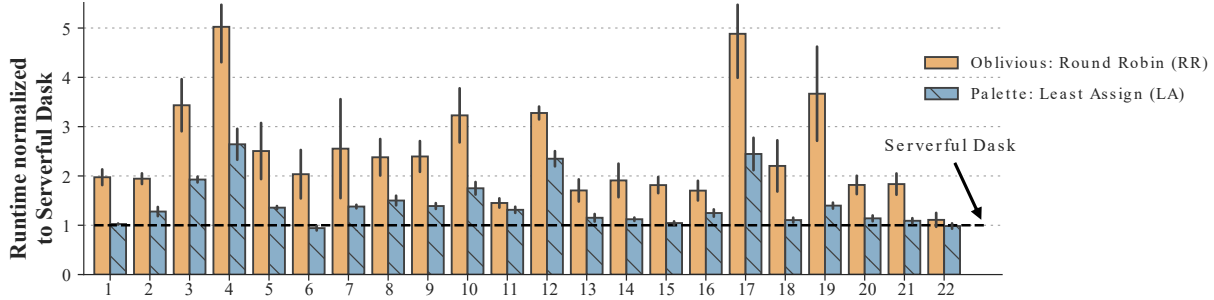
Figure 9. TPC-H evaluation results comparing Palette to a representation of today's FaaS load balancers (Oblivious), an implementation variant of Palette (Virtual worker) and (Chain-coloring), an unmodified version of Dask running on dedicated services (Serverful Dask). Here, Palette and Oblivious use the virtual worker policy (§6.2).

```python
from nums import numpy as nps
from nums.models.glms import LogisticRegression
import nps
# Phase 1 (reading data)
data = nums.read_csv("HIGGS.csv")
# Phase 2 (splitting data)
y, X = data[:,0].astype(nps.int32), data[:,1:]
# Phase 3 (fitting the model)
model = LogisticRegression(solver="newton-cg")
model.fit(X, y)
# Phase 4 (prediction and accuracy)
y_pred = model.predict(X)
accuracy = (nps.sum(y == y_pred)/X.shape[0]).get()
```

Listing 1. Phase-breakdown of the LRHiggs benchmark.

Finding 8: *Serverless NumS on Palette outperforms locality-oblivious serverless NumS, and sometimes even outperforms NumS on Ray.* Figure 10a shows the mean runtime, 5%, and 95% confidence intervals for the three benchmarks running NumS with Palette (Oblivious Random, Oblivious Round Robin, and Least Assigned (LA)) and with Ray. From Figure 10a, we observe that LA outperforms Oblivious Random by 27% in case of LRHiggs, 25% in case of MMM-2GB and 61% in case of MMM-16GB. It also outperforms Oblivious Round Robin in all the benchmarks, showing that minimizing load on individual workers alone is not sufficient to achieve high performance. Workloads involving potentially higher data exchange between phases of computation benefit more from Palette because it tries to schedule tasks where the data is.

When comparing between backends, Ray dominates both Oblivious policies. However, with Palette's locality API, FaaS functions become competitive with Ray. In fact, in the LRHiggs benchmark, we show that Palette outperforms Ray by 41%.

In Figure 10b, we focus on the performance in individual phases of the LRHiggs benchmark. While Ray is optimized to load data for NumPy use (Phases 1 and 2), Palette outperforms Ray in the computation-heavy stages of the workload by scheduling tasks closer to where their data is (Phases 3 and 4). Palette reduces the data copies in the overall system by minimizing the number of unique workers that would operate on a given data. This has two implications: (a) it
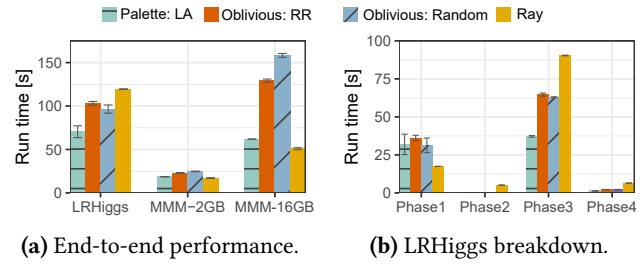

(a) End-to-end performance.　**(b)** LRHiggs breakdown.

Figure 10. Evaluation results with NumS.

enables running larger workloads using the same amount of resources, and (b) it reduces the cost of running the workloads in a serverless setting where users are charged for consumed resources (typically memory-seconds).

8 Related Work

Performance of serverless platforms and DAG frameworks have sparked a rich literature. We review related ideas on maintaining locality information, serverless caching, serverless DAG engines, and DAG processing on serverless.

Locality Information. Perhaps the closest proposal to locality hints is an extended abstract by Schleier-Smith [71], proposing to augment the serverless API with a "partition key" for addressing FaaS ephemeral state. Our color abstraction is similar, except that it is a hint. To the best of our knowledge, Palette is the first implementation and evaluation of locality hints for serverless.

Outside of serverless frameworks, Kubernetes has a concept of affinity tags [24] used at deployment time for placing pods. Like Palette's colors, these tags provide indirection, separation of concerns, and enable several policies by application owners. Web applications commonly use sticky load balancing, in which all requests for a user session get directed to the same instance. Palette's colors provide the first generic way to implement this stickiness for FaaS.

Archer *et al.* [9] improve the hit ratio of an in-memory cache in Google's web search backend, and increase the cluster throughput, by making the load balancer route requests to the replica with highest affinity to a query's search terms. They note that this approach is useful because the "replicas

carry over mutable state between requests". Colors are a generic and flexible way of achieving this in a FaaS backend, as we demonstrate in our social network example (§7.1).

Serverless Caching. Faa$T [70] colocates a cache with each instance of a FaaS application. While Faa$T uses consistent hashing to forward requests between caches, it does not influence FaaS invocation routing. Thus, applications such as Dask incur many remote hits with performance comparable to an external in-memory KVS. Our implementation builds on Faa$T and uses the Palette load balancer to outperform oblivious Faa$T by 45% in Dask and NumS (§7.2).

OFC [59] implements a transparent and opportunistic in-memory cache on top of OpenWhisk [62], using RAM-Cloud [63] as the distributed cache. OFC implements a basic form of execution locality: executions are placed where their inputs are located. This policy works well only for linear chains of functions. However, many queries have large fan-outs (*e.g.*, TPC-H queries 5, 7, 8, 10, and 12). For fan-outs, OFC schedules all outgoing nodes on the same node, which leads to poor performance as it does not allow parallelism. In addition, OFC doesn't specify how locality is implemented in a general way, as it assumes that the source object can be extracted from an invocation.

Infinicache [82] implements a distributed key-value cache using serverless functions. Applications (serverless or not) can access this cache over the network and achieve lower costs compared to traditional key value caches such as Redis. Infinicache is an orthogonal work to Palette and does not discuss or provide functionality for locality.

Serverless Scheduling. Fuerst and Sharma [36] propose a locality-aware load balancer for serverless clusters. They look at the orthogonal problem of selecting to which servers to send invocations of the same function, and propose an improvement over OpenWhisk's algorithm [62]. Concentrating these invocations in less servers reduces the number of cold starts. They do not, however, consider the locality w.r.t. the *data* accessed by these invocations. Given a set of servers with warm instances of a function, Palette's locality hints can be used to improve data locality.

Serverless DAG Engines. There are many options, with different tradeoffs, to execute DAGs on serverless functions.

One can use existing serverless frameworks and rely on an external cache like Redis (*e.g.*, gg short-lived [34]). This leads to poor performance, comparable to Oblivious (§7). Replacing serverless frameworks with containers (such as AWS Fargate) enables existing DAG frameworks to run. Another alternative is to use serverless functions as containers by setting up long-lived functions with reverse connections to a central controller [34, 73, 86]. Both container variants require provisioning servers and break serverless' simplicity.

Cloudburst [78], HaSTe [84], SONIC [56], SAND [3], Faastlane [53], and Ray [67] all implement DAG execution engines. By sending the whole DAG (instead of individual function calls), this programming model provides a wealth of information to the FaaS scheduler. The scheduler has complete information about the workers, including the load and contents of the caches, and can place functions where most of their input data is cached. SONIC selects how to best transfer data among stages of a DAG, and does data-aware function placement, based on its knowledge of the DAG. Palette explores a different point in the trade-off space between interface complexity and performance, and shows that we can achieve performance very close to schedulers with complete information, while still maintaining a simple and general interface.

Wukong [20] implements a serverless backend for Dask by running DAG nodes as serverless functions, and storing intermediate data in a key-value store (KVS), which has to be provisioned and maintained separately. Wukong modifies the Dask scheduler to find chains of tasks that can be executed on the same node, and sends these sections of the graph to the same function instance. Wukong achieves locality by fusing functions, and avoids serialization and deserialization for functions in the same instance. Instances coordinate through the KVS when there are fan-ins and fan-outs in the DAG. Wukong also removes the scheduler from the critical path by having functions schedule each other, at the expense of complexity in the functions' implementations. With locality hints and a serverless cache, one can achieve similar performance to Wukong for DAG computations, without having to maintain a separate KVS; by using the virtual servers approach (cf. §6.2) Palette also requires little modification to existing frameworks.

Parallel Task Scheduling. The extensive literature on scheduling parallel tasks and DAGs is documented in excellent surveys [75]. Palette enables PaaS users to implement advanced scheduling algorithms with ease.

9 Conclusion

Palette addresses the performance gap of serverless platforms for data-intensive applications by introducing colors as locality hints. This approach allows users to leverage the benefits of serverless systems while allowing the platform to place successive invocations related to each other in the same executing node. Our prototype evaluation shows that Palette closes the gap for web applications with near-perfect scaling of cache hit ratios and mostly closes the gap for data analytics like Dask and NumS.

10 Acknowledgments

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI*, 2020.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *ATC'18*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[4] Alibaba. Function compute. https://www.alibabacloud.com/product/function-compute. Accessed on September 23rd, 2022.

[5] Mehmet Altınel, Christof Bornhövd, Sailesh Krishnamurthy, Chandrasekaran Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: Paving the way for an adaptive database cache. In *VLDB*, 2003.

[6] Amazon. Aws lambda features. https://aws.amazon.com/lambda/features/. Accessed on September 23rd, 2022.

[7] Amazon. AWS Step Functions. https://aws.amazon.com/step-functions/. Accessed on September 23rd, 2022.

[8] Amazon. Best practices for working with AWS Lambda functions - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html. Accessed on September 23rd, 2022.

[9] Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-aware load balancing of data center applications. *Proc. VLDB Endow.*, 12(6):709–723, feb 2019.

[10] Microsoft Azure. Azure durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview. Accessed on September 23rd, 2022.

[11] Microsoft Azure. Azure functions host. https://github.com/Azure/azure-functions-host. Accessed on September 23rd, 2022.

[12] Microsoft Azure. Azure Functions Python developer guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-python. Accessed on September 23rd, 2022.

[13] Microsoft Azure. Azure functions triggers and bindings concepts. https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=python. Accessed on September 23rd, 2022.

[14] Microsoft Azure. Best practices for reliable azure functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices?tabs=python. Accessed on September 23rd, 2022.

[15] Microsoft Azure. Introduction to Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview. Accessed on September 23rd, 2022.

[16] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. Stateful serverless computing with crucial. *ACM Trans. Softw. Eng. Methodol.*, 31(3), mar 2022.

[17] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI'20*, pages 753–768, 2020.

[18] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *OSDI*, 2018.

[19] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. *ICDE*, 2011.

[20] 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 *SoCC*, 2020.

[21] CloudFlare. Cloudflare FaaS. https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/. Accessed on September 23rd, 2022.

[22] Dask. Dask: Scalable analytics in Python. https://docs.dask.org/en/latest/. Accessed on September 23rd, 2022.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[24] Kubernetes Documentation. Inter-pod affinity and anti-affinity. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity. Accessed on September 23rd, 2022.

[25] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. Towards supporting millions of users in modifiable virtual environments by redesigning Minecraft-Like games as serverless systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[26] Dheeru Dua and Casey Graff. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2017. Accessed on September 23rd, 2022.

[27] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *SoCC*, 2020.

[28] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. A Review of Serverless Use Cases and their Characteristics. *arXiv:2008.11110 [cs]*, January 2021. arXiv: 2008.11110.

[29] Melih Elibol, Vinamra Benara, Samyu Yagati, Lianmin Zheng, Alvin Cheung, Michael I. Jordan, and Ion Stoica. NumS: Scalable Array Programming for the Cloud, 2022. arXiv:2206:14276 [cs].

[30] Facebook. Network repository. https://networkrepository.com//. Accessed on September 23rd, 2022.

[31] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In Philippe Jacquet, editor, *AofA: Analysis of Algorithms*, volume DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, pages 137–156, Juan les Pins, France, June 2007. Discrete Mathematics and Theoretical Computer Science.

[32] Fn. Fn project. https://fnproject.io/. Accessed on September 23rd, 2022.

[33] Sadjad Fouladi, Kayvon Fatahalian Hanrahan, and Keith Winstein. R2e2: Low-latency path tracing of terabyte-scale scenes using thousands of cloud cpus. In *SIGGRAPH*, 2022.

[34] 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 *ATC*, 2019.

[35] 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 *NSDI*, 2017.

[36] Alexander Fuerst and Prateek Sharma. Locality-aware Load-Balancing For Serverless Clusters. In *HPDC*, Minneapolis, MN, 2022.

[37] Delbert Ray Fulkerson. Note on Dilworth's decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4):701–702, 1956.

[38] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.

[39] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. Serverless end game: Disaggregation enabling transparency. *arXiv:2006.01251 [cs]*, 2020.

[40] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. Scalable Query Result Caching for Web Applications. *VLDB*, 1(1):550–561, 2008.

[41] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[42] 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. In *CIDR*, 2019.

[43] Ibm cloud functions. https://cloud.ibm.com/functions/. Accessed on September 23rd, 2022.

[44] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.*, 41(3), 2007.

[45] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. In *OOPSLA*, 2019.

[46] 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*, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

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

[48] Chanwit Kaewkasi. *Docker for Serverless Applications: Containerize and orchestrate functions using OpenFaas, OpenWhisk, and Fn.* Packt Publishing Ltd, 2018.

[49] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[50] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. Towards Serverless as Commodity: A Case of Knative. In *WoSC*, 2019.

[51] 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.

[52] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[53] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.

[54] Kubernetes. Production-grade container orchestration. https://kubernetes.io/. Accessed on September 23rd, 2022.

[55] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *SIGMOD*, 2002.

[56] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[57] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent Hashing with Bounded Loads. In *SODA*, 2018.

[58] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *SIGMOD*, 2020.

[59] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. OFC: an opportunistic caching system for FaaS platforms. In *EuroSys*, 2021.

[60] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.

[61] OpenFaaS. Openfaas. https://www.openfaas.com/. Accessed on September 23rd, 2022.

[62] Apache OpenWhisk. Apache openwhisk. https://openwhisk.apache.org/. Accessed on September 23rd, 2022.

[63] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[64] Google Cloud Platform. Cloud functions. https://cloud.google.com/functions/. Accessed on September 23rd, 2022.

[65] Python. concurrent.futures – launching parallel tasks. https://docs.python.org/3.8/library/concurrent.futures.html. Accessed on September 23rd, 2022.

[66] Martin Raab and Angelika Steger. "Balls into bins" – a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[67] Ray. Scaling Python made simple, for any workload. https://github.com/ray-project/ray. Accessed on September 23rd, 2022.

[68] Introduction to Redis. https://redis.io/docs/about/. Accessed on September 23rd, 2022.

[69] Redis cluster specification. https://redis.io/docs/reference/cluster-spec/. Accessed on September 23rd, 2022.

[70] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *SoCC*, 2021.

[71] Johann Schleier-Smith. Serverless Foundations for Elastic Database Systems. In *CIDR Extended Abstract*, 2019.

[72] 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 *ATC*, 2020.

[73] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless Linear Algebra. In *SoCC*, 2020.

[74] Klaus Simon. An Improved Algorithm for Transitive Closure on Acyclic Digraphs. *Theoretical Computer Science*, 58(1), 1988.

[75] Khushboo Singh, Mahfooz Alam, and Sushil Kumar Sharma. A survey of static scheduling algorithm for distributed computing system. *International Journal of Computer Applications*, 129(2):25–30, 2015.

[76] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform. *CoRR*, abs/1911.09849, 2019.

[77] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler,

Patrick McCormick, and Alex Aiken. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *SC*, 2020.

[78] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. In *PVLDB*, 2020.

[79] Mark Freeman Tompkins. Optimization techniques for task allocation and scheduling in distributed multi-agent operations, 2003. Master's Thesis.

[80] TPC-H. TPC-H Version 3. http://www.tpc.org/tpch/. Accessed on September 23rd, 2022.

[81] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*, 2021.

[82] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *FAST*, 2020.

[83] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *ATC*, 2018.

[84] Yi Yao, Jiayin Wang, Bo Sheng, Jason Lin, and Ningfang Mi. Haste: Hadoop yarn scheduling based on task-dependency and resource-demand. In *SoCC*, 2014.

[85] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[86] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *SoCC*, 2020.

[87] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.