rFaaS: Enabling High Performance Serverless with RDMA and Decentralization.

Marcin Copik  
*ETH Zürich*  
Konstantin Taranov  
*ETH Zürich*  
Alexandru Calotoiu  
*ETH Zürich*  
Torsten Hoefler  
*ETH Zürich*

**Abstract**

The need for high performance is present in many computing platforms, from batch-managed and scientific-oriented supercomputers to general-purpose cloud platforms. At the same time, data centers and clusters still suffer from low utilization of computing resources. Function-as-a-Service, a modern cloud programming paradigm for pay-as-you-go execution of stateless functions, brought the elasticity needed to take advantage of ephemeral resources. However, its performance characteristics cannot match coarse-grained IaaS and cluster allocations. To make serverless computing viable for high-performance and latency-sensitive applications, we present rFaaS, the first RDMA-accelerated FaaS platform. We identify key limitations of modern serverless systems - centralized scheduling and inefficient network transport - and propose an overhaul of FaaS architectures with decentralized allocations and low-latency invocations. We show that our remote functions add only negligible overhead on top of the fastest available networks, and we improve the execution latency by orders of magnitude compared to contemporary FaaS platforms. Furthermore, we demonstrate the performance of rFaaS by evaluating real-world FaaS benchmarks and parallel applications. Overall, our results show that decentralization and remote memory access help serverless applications to achieve high performance while increasing server utilization.

1 Introduction

The landscape of high-performance applications has changed drastically over the last decade. Cloud computing brought two major innovations: elastic resource management and a 7-fold cost decrease [1]. The cloud’s elastic scalability fueled new applications in massive data analytics, image and video processing, and large-scale machine learning. As a result, high-performance, latency-sensitive, and parallel applications running in private data centers and scientific-oriented supercomputers started to embrace public clouds [2–4].

To take advantage of flexible cloud resources, high-performance applications need an elastic programming model. For example, applications performing real-time machine learning inference or multimedia processing must satisfy strict performance and Quality-of-Service (QoS) requirements. However, acquiring new Infrastructure-as-a-Service (IaaS) resources in the cloud is time-consuming, and even as of 2021, the time needed to allocate a virtual machine is counted in minutes, not seconds [5]. Without high-performance computing units that are immediately available, applications must employ complex and domain-specific optimizations such as inference-focused systems [6, 7] or resort to overprovisioning.

The problem of low resource utilization is as old as the computing centers themselves. Resources are overprovisioned to handle the peak demand, fulfill the Service Level Objective (SLO) demands, and quickly adjust to spatial and temporal imbalances in traffic. As a result, the resource utilization is historically low, from 5-20% utilization rate in pre-cloud data centers [8, 9], through 40-60% in modern data centers and clouds [10–13], up to 80-90% in highly competitive and batch-managed supercomputers [14–16]. Even though significant improvements have been made in recent years through resource reclamation [17, 18], transient cloud servers [19], and co-location of offline and online services [20–23], the problem has a significant impact as “increasing utilization by a few percentage points can save millions of dollars” [17].

![Figure 1: The remote invocations of an empty C++ function on serverless platforms and rFaaS: median (solid), and 99th latency (dashed) for a single function (details in Sec. 5.3).](image-url)
Furthermore, underutilization does not necessarily lead to energy savings [24], and wasted computing resources increase the overall energy cost and environmental impact. The rapid and frequent utilization changes on many such platforms indicate that this gap cannot be addressed with persistent and long-running allocations. Fine-grained and ephemeral programming models could take advantage of such resources - to benefit both users and providers, as idle computing and memory resources could be offered at lower costs.

**Function-as-a-Service (FaaS)** is a new cloud paradigm combining the full elasticity of cloud resources with a maximally simplified programming model: users program stateless functions and the cloud takes away from them the responsibility of scheduling invocations of such functions. Thanks to the fine-grained parallelism and the pay-as-you-go billing system, serverless functions could become a solution for all tasks that benefit from an elastic allocation of computing resources, but their applicability is limited by high invocation latencies (Fig. 1). Functions are used as costly but flexible elastic workers to fulfill SLO requirements [6, 25], and they could become the implementation of **HPC-as-a-Service** [26]. However, functions are not yet ready for general-purpose and high-performance applications. For FaaS to become a viable programming model for high-performance applications, it must overcome crucial performance challenges (Table 1).

We address these challenges in rFaaS, the first **RDMA-capable serverless platform** (Sec. 3). We revamp the fundamentals of FaaS design to define **RDMA abstractions** that hide the network stack complexity and preserve the elasticity and isolation guarantees of serverless. We focus on three directions that differentiate our work from other serverless architectures. First, rFaaS employs a **decentralized resource management** model where centralized and global managers are removed to facilitate faster and independent allocations. Then, we accelerate our serverless system by **reducing its invocation path** for high-priority and low-latency tasks: rFaaS invocations are handled directly between the client and a function executor. Finally, to achieve second latency invocations, we replace HTTP and REST interfaces with an **RDMA function dispatch protocol** that removes the milliseconds of OS latency [27]. We help to integrate FaaS executions into latency-sensitive and high-performance applications and show **hot** invocations with an overhead of a little over 300 nanoseconds on top of the fastest network (Fig. 1).

To incorporate serverless computing into high-performance applications, we present a **C++ programming model** for straightforward integration of rFaaS functions into new and existing C++ codebases (Sec. 4). Our work is a major step towards increasing data center and cloud efficiency by using idle and ephemeral resources for tasks demanding high performance. We demonstrate the elasticity, efficiency, and performance of rFaaS with an evaluation of microbenchmarks and real-world serverless functions (Sec. 5).

Our paper makes the following contributions:

- We present the design and implementation of the first RDMA-capable serverless platform, including (1) decentralized FaaS resource management and (2) a novel, low-latency, and zero-copy hot type of serverless invocations. rFaaS is available on an open-source license.\(^1\)
- We conduct an experimental verification against state-of-the-art open-source and commercial serverless platforms summarized in Figure 1 and show that rFaaS has a median overhead over pure RDMA transmission of little over 300 ns and achieves the available link bandwidth.
- We demonstrated rFaaS usability with real-world serverless functions, and show the invocation latency is sufficient for every the most demanding SLO.

| Requirements          | rFaaS               | Other solutions. |
|-----------------------|---------------------|------------------|
| Low-latency invocations | ☐                   | Nightcore [28]   |
| Direct allocations    | ☐                   | ☐                |
| High-speed networks   | ☐                   | ☐                |
| Decentralized scheduling | ☐             | ☐                |
| Efficient workflows   | ☐                   | ☐                |
| Direct communication  | ☐                   | ☐                |
| Fast and shared storage | ☐             | Open problem.   |
| Affordable cost       | ☐                   | Open problem.    |
| Consistent performance| ☐                   | Open problem.    |

Table 1: rFaaS solves (☐) and enables solutions (📍) to the major challenges of high-performance FaaS [34–37].

\(^1\)The anonymized code is available under the link [https://www.dropbox.com/s/1x3ucz6klfkidq/rfaas.zip](https://www.dropbox.com/s/1x3ucz6klfkidq/rfaas.zip)

2 Background

rFaaS solves the low server utilization problem of data centers by identifying the opportunity to resell idle and ephemeral resources (Sec. 2.1). At the same time, modern FaaS platforms are too constrained (Sec. 2.2) to take advantage of high-speed networks and remote memory operations (Sec. 2.3), motivating the complete revamp of serverless architecture (Sec. 3).

2.1 Resource Utilization

At the advent of cloud computing, server utilization has been estimated to be just 5-20% (2008, 2010) [8, 38]. The latter study indicated that 30% of servers in data centers do not perform any work. Low utilization negatively affects capital investments through wasted resources, and increases operating costs, as the energy usage of servers doing little and no work is more than 50% of their peak power consumption [39].

**Resources** Datacenter and cloud computing resources are heavily underutilized. In data centers, the CPU utilization was 5-10% (Yahoo!, 2008-2009) [40] and 18% (IBM, 2009-2011) [9]. Similarly, the utilization of private clouds was 10% (2013) [41] and less than 20% for 80% of clusters (2016) [42]. In Google data centers, the CPU utilization does not exceed 60% (2011) [43, 44], and while recent results indicate an increase through the “best-effort” batch jobs, the utilization still does not exceed 60% (2019) [10]. In the cloud, the average CPU utilization of virtual machines was 4-17% (AWS,
functions are a natural fit for the idea of opportunistic computing [48], and rFaaS can employ ephemeral cloud resources.

2.2 FaaS Computing

Function-as-a-Service (FaaS) is a cloud service concerned with executing stateless and short-running functions. The serverless functions are dynamically allocated in the cloud, and the users are freed from the usual responsibilities of managing resources. The cloud provider charges users only for the time and resources used in a function execution, and applications with irregular or infrequent workloads can benefit from the elastic allocation of computing resources and the pay-as-you-go billing system. For a cloud operator, the fine-grained executions provide an opportunity to increase system efficiency through oversubscription and flexible scheduling. Serverless is adopted by major cloud systems [49–52].

Platform We characterize the FaaS platforms with a high-level overview presented in Figure 2 and refer interested readers to a wider discussion in the literature [34, 35, 53]. Functions are invoked via triggers (A), including internal cloud events such as database update or a new entry in a queue, and the standard external trigger via a cloud HTTP gateway that exposes functions to the outside world. A function scheduler (B) places the invocation in a cloud-native execution environment (C), and the function code is downloaded from the cloud storage (D). Function are allowed to initiate connections to external cloud resources and services, and can also use the filesystem of its sandbox as a temporary storage. A sandbox instance handles many consecutive invocations, so resources are cached and reused across executions.

Observation #1 CPU and memory resource usage might be low even on allocated servers. Therefore, users would benefit from reselling unused cycles and the accompanying infrastructure to host serverless invocations.

Workload variability Studies analyzing cloud and cluster workloads reveal significant temporal, spatial, and diurnal variability. In the Alibaba cloud data center [46], some instances use less than 10% and others 60–90% of their CPU, and multi-core containers underutilize CPUs because of the temporal workload variability [13]. Analysis of enterprise workloads reveals heavy skew in the workloads, where CPU usage is unpredictable and dynamic for 20% of VMs [41], and peak usage is much higher than 90th and 99th percentile [47], forcing the system to be ready to release significant resources for the peak traffic. While overprovisioned resources could be reclaimed (Sec. 6), it usually requires extensive profiling and application requirements classification. Furthermore, the prime causes of overprovisioning are the low-latency demands in application SLOs, which force reclaimed resources to be transient and easily retrievable by the application.

Observation #2: Variable workloads make reallocations and under-provisioning challenging. Stateless and short-lived functions are a natural fit for the idea of opportunistic computing [48], and rFaaS can employ ephemeral cloud resources.

Observation #3: The multi-step invocation path is a barrier to achieving zero-copy and fast serverless acceleration. rFaaS removes the centralized cloud proxies from invocations.

High-Performance Serverless The elastic parallelism of FaaS has gained minor traction so far in the world of high-performance and scientific computing [59], but serverless is used in compute-intensive workloads such as data analytics, video encoding, linear algebra, and machine learning [60–67]. Such applications require low-latency communication and optimized data movement. They cannot tolerate the large overheads of invoking remote functions. They need a pricing model that is fair towards compute-intensive functions [34]. Although recent research improved serverless performance by including RPC [28], exploiting data locality, and co-locating invocations [31], latency-sensitive and parallel applications need fast remote invocations to achieve high scalability.
Observation #4: Connection latency and bandwidth are the fundamental bottlenecks for remote invocations, yet serverless platforms do not take advantage of modern network protocols. rFaaS integrates high-speed RDMA connections.

2.3 Remote Direct Memory Access

RDMA-capable networks have become a standard tool for implementing high-performance communication libraries, transactions, distributed protocols, storage, and databases [68–74]. Unlike in the standard TCP/IP stack, RDMA data transfers are performed entirely by a dedicated network controller bypassing both the CPU and operating system. Instead of exchanging OS-managed and buffered packets or datagrams, RDMA allows the communicating endpoints to directly read, write and atomically update memory contents of its remote counterpart. The CPU and cache hierarchy do not participate in this process, and the remote network controller forwards the arriving data over the PCI bus to the memory and avoids latency introduced by the kernel. Thus, the receiver is passive and potentially even unaware of the communication.

This communication protocol provides high-speed and rapid access to other server’s data with a lower CPU utilization at the cost of a simplified and crude interface. Achieving the best performance requires fine-tuning such as aligning memory, controlling device buffers, and utilizing vendor-specific optimizations, e.g., message inlining. Error-handling is solely the programmer’s responsibility, data transfers are restricted to locked memory pages, and security concerns might require additional mitigation mechanisms [75, 76]. The RDMA devices must be accessed in cloud environments through virtualization solutions such as PCI passthrough, para-virtualization, and virtual device functions [77]. Finally, RDMA can be used with InfiniBand, Cray and Intel networks, RDMA over Converged Ethernet (RoCE), and software virtualizations [78, 79].

Observation #5: Cloud applications take advantage of low overhead and high performance of RDMA networks, but FaaS needs a complete redesign to benefit from them. The design of rFaaS is RDMA-compatible from the start.

3 RDMA-based Serverless Platform

rFaaS is a serverless platform tailored for the needs of high-performance applications, combining the flexibility of FaaS systems with the low overhead executions primarily available in the cloud IaaS and HPC clusters. rFaaS implements the main FaaS paradigm of remote executions of stateless functions, yet it avoids the major performance overheads of serverless systems by replacing the REST and RPC invocations with direct memory operations on remote servers. However, we retain the serverless semantics of executing code on dynamically allocated and ephemeral computing endpoints.

Our philosophy in implementing rFaaS is to drastically reduce the critical path of invocations. We achieve this goal by reducing the number of parties involved in transmitting function data and removing the centralized gateway and resource manager from the invocation path. Figure 3 shows an overview of rFaaS: a decentralized allocation system where clients negotiate a lease of computing resources from a frequently refreshed list of available servers (Sec. 3.2). Our functions gain a direct RDMA connection to the user code executor without sacrificing their serverless nature: as in other FaaS platforms, no specific assumptions about the underlying computing and storage hardware are made. We capitalize on this gain further by implementing an RDMA-based invocation system designed to minimize invocation latency (Sec. 3.3).

3.1 Components of rFaaS

Resource Manager A global database of active resources is a necessary component of each serverless platform. Servers become available for function execution as soon as the database contains an entry with a description of their resources and connection details. The role of the resource manager is to update and distribute a ranked list of spot executors. The cluster and cloud operators add and remove servers (2), and the manager uses heartbeats to verify the status of spot executor periodically. Clients read the list of spot executors (1).

Spot Executor When clients begin offloading serverless tasks to rFaaS, they select spot executors to achieve the desired number of parallel workers. These servers offer idle and unused hardware resources (CPU cores, memory) to support the dynamic execution of serverless functions. Clients negotiate an allocation of computing resources with the lightweight allocator (2). The dedicated allocator process is responsible for connecting new clients, managing user code executors, removing processes that are idle for a long time or exceed specified time limits, and accounting for resource consumption. When an allocation is successful, the allocator initialize an isolated execution context with an RDMA-capable execution process. Finally, clients can establish a direct RDMA connection with each allocated executor process and invoke functions by writing function header and payload directly into their memory (12). The results are returned to the client in a similar fashion, and the client caches allocation status for consecutive invocations on warmed-up resources.

3.2 Decentralized Allocation

Direct and decentralized resource management is another feature differentiating rFaaS from other FaaS systems. To
execute a function, clients do not involve the resource manager. Instead, they randomly select spot executors from the resource list (A1) and send allocation requests directly to those servers. Upon successful allocation, managers allocate execution contexts, initialize RDMA-aware execution processes, and notify clients who start sending invocations. Connections to user executor processes are cached by clients to provide fast consecutive executions on warmed-up resources. To support straightforward deallocation of on-demand executors, clients use the connection status to check if the process is alive.

3.3 Low-Latency Invocation

A critical feature of rFaaS is ensuring invocations have the lowest overhead possible. While an on-demand allocation of idle resources improves the economics of cloud systems, it would be counterproductive to incorporate rFaaS functions into high-performance and latency-sensitive applications if we did not offer low-latency invocations. In Figure 4, we present the steps and overheads of various invocation models in rFaaS. Our platform preserves the FaaS semantics of cold and warm invocations, and we extend the invocation models with a new type of hot invocation that guarantees zero-copy executions on pre-allocated and -billed hardware. We now detail the characteristics of different invocation types and the mechanisms offered to enable parallel invocations.

Cold The cold invocation includes significant overheads caused by the initialization of an execution context. In rFaaS, clients negotiate the allocation directly with spot executors by requesting the desired core count, memory, and timeout for the allocation. Clients send allocation requests until they succeed in allocating the desired number of computing resources.

The lightweight allocator initializes an isolated execution sandbox and assign the requested computing and memory resources to it. The user code executor starts in the sandbox, accesses the selected RDMA device, registers memory buffers, creates worker threads pinned to assigned cores. Each executor process has a configurable number of thread workers who work independently, and each one corresponds to a single function instance. Thus, clients can allocate multiple workers in a single allocation request. When the initialization is done, the client receives the executor’s connection settings, establishes connections to all threads, and invokes functions by writing requests directly to the workers’ remote memory. Overall, sandbox initialization adds on average 25 ms and 2.7 seconds of overhead for bare-metal and Docker-based executors, respectively, on an HPC node (Section 5).

Warm When the sandbox and user code executor are already allocated, the client transmits the function payload using an RDMA connection. Executor threads do not share RDMA resources, and they use blocking wait independently to receive completion events corresponding to new warm invocation requests. Once a completion event arrives, the thread wakes and executes the request. Using blocking wait increases latency but significantly decreases the pressure on computing resources compared to active polling. Compared to native RDMA performance, warm invocations have an overall overhead of fewer than 6 microseconds for a round-trip invocation.

Hot The novel hot invocations improve the performance of warm FaaS executions by adding the obligation that threads actively poll for invocation requests. The busy polling decreases the invocation latency since threads do not enter a blocked state to wait for an interrupt generated by the RDMA driver. The thread enters the hot invocation mode immediately after execution and polls RDMA events without sleeping to improve the performance of consecutive invocations. We roll back to warm executions after a configurable time without a new invocation, freeing up the CPU. This configuration decreases the overall overhead for a round-trip invocation to ca. 300 nanoseconds on average. However, it comes at the cost of occupying the CPU core and preventing other functions from using the computing resources. Therefore, the hot polling time should be accounted as active computation time.

3.4 Scalability

A high-performance serverless platform must handle scaling in three directions: number of spot executors, number of rFaaS users, and the number of functions invoked by a client.

Horizontal Scaling The size of a cloud RDMA network is the upper limit for the number of spot executors and clients in an rFaaS instance. While modern cloud RDMA networks count many thousands of clients [27, 80], the networks can scale globally in future deployments. Since the network throughput of RDMA connections decreases significantly with the number of clients [81, 82], we replicate the resource manager where each replica serves data on spot executor availability. While the replicated cloud datastores employ expensive, strongly consistent transactions to prevent stale reads, these are not a concern for us. A late announcement of new executors can lead to a slightly smaller availability of resources for a subset of clients and only for a short time. Similarly, removing resources from the manager is not a problem since rFaaS clients must tolerate the volatility of transient resources. As a result, our replicated resource manager can implement a less expensive but scalable eventual consistency [83].
Clients receive the spot executor data, distribute it internally, and cache it locally for future use. Spot executor data can be periodically refreshed. It could be tempting to employ unreliable datagrams and multicast to distribute delta updates from managers to connected clients automatically. However, such group communications are rarely supported in cloud environments because of scalability and security issues.

**Parallel Invocations** rFaaS implements parallel invocations by simultaneously dispatching function execution requests to threads of remote user executors. Since a client has a direct RDMA connection to each thread worker, it can invoke functions independently. The scalability is achieved by exploiting the non-blocking nature of RDMA write operations and using disjoint memory buffers to store invocation results. Multiple RDMA connections improve network utilization as more processing units of a network controller are involved [82]. Each executor thread switches between hot and warm invocations on its own, further aiding elasticity.

### 3.5 Fault tolerance

rFaaS clients can experience failures in three ways: the spot executor is evicted, the executor shuts down uncontrollably, or the function crashes the execution process. Even though the likelihood of a failure caused by the eviction of harvested cloud resources is very low (< 0.002%) [48], we provide mitigation strategies for a seamless user experience. Executor connections are cached in the client library for further invocations, and a server failure is detected through a disrupted RDMA connection. The library repeats the invocation on other servers for a finite number of retries to avoid an infinite loop on a broken function. Furthermore, the spot executor manager frequently verifies the status of its executors, and it notifies the client when it detects a premature exit and failure.

### 3.6 Isolation and Security

In addition to bare-metal executors, we include containerized executors to ensure privacy and security in the multi-tenant execution in rFaaS. The main requirements imposed by rFaaS are virtualization support for RDMA-capable network controllers and negligible performance overheads. The current implementation uses Docker containers to implement isolated execution contexts for user functions. We limit the user’s code from accessing any resources, data, and code not provided with the invocation. We use the Single Root I/O Virtualization (SR-IOV) to virtualize network controllers in a multi-tenant environment. Virtual network functions provide isolated but high-performance access for different users [84].

Furthermore, modern RDMA extensions provide authentication, payload encryption and memory protection that ensure secure transmission in cloud networks [76, 85, 86].

### 3.7 Modularity

The world of high-performance applications and cloud systems is rich and diverse. Thanks to its modular design, rFaaS supports extensions into new environments and hardware.

### Network

While our implementation manages RDMA networks with libverbs, the rFaaS functionality is orthogonal to the device interface and can be implemented with higher-level concepts from libfabric [87]. rFaaS can be deployed on other networks providing RDMA-like semantics, such as the Elastic Fabric Adapter in the AWS cloud [88]. In addition, software virtualization can be employed in data centers without high-speed networks, offering RDMA semantics at the cost of overheads and increased CPU usage [78, 79].

### Language

rFaaS supports C, C++, and Python functions and native integration into C/C++ applications (Sec. 4). The language choice is, however, independent from the platform itself. rFaaS functions can be implemented effortlessly in languages ABI-compatible with C, such as Rust, and with the help of foreign-function interface in languages prevalent in the serverless community, such as Python and Node.js.

### Sandbox

rFaaS functions can be served in other environments than bare-metal processes or Docker containers, e.g., in HPC container Singularity [89], gVisor [90], and in microVMs such as Firecracker [56, 91] that provide a higher level of isolation with negligible performance overheads. New sandbox types can be integrated effortlessly as long as a paravirtualization or passthrough to the RDMA NIC is provided.

### 3.8 rFaaS versus traditional FaaS

rFaaS is a fundamental building block for bringing RDMA abstractions into serverless. While rFaaS implements the essential semantics of FaaS computing - remote invocations on transient and multi-tenant resources with pay-as-you-go billing - we tailor the design of the serverless platform to the demands of high-performance and low-latency. On top of the standardized interface of rFaaS, additional features can be implemented according to the needs of specific programming frameworks: native datatypes and serialization, collective operations, logging of invocations. Similarly, rFaaS does not come with a dedicated authorization system, as that can be provided through the existing cloud solutions.

### Triggers

The trigger mechanism is replaced with decentralized allocations and direct connection to an executor process, removing the proxies and caching the connection to minimize latency and unnecessary copies. However, our RDMA abstractions are compatible with the serverless execution model and can be used to provide centralization known from other FaaS systems at the cost of higher overheads. For example, such a platform could utilize RDMA-aware queues and storage [92, 93] to implement a default entry point for function triggering. Users can still achieve orders of magnitude improvement in invocations latencies with the end-to-end latency of as low as 100 microseconds in RDMA Kafka [94] and single-digit microseconds for RDMA native publish-subscribe service [93]. Furthermore, centralized FaaS gateways can benefit from RDMA acceleration in the cloud network by using rFaaS abstractions in the backend, preserving a consistent interface for users.
4 rFaaS in Detail

rFaaS functions are aligned with existing cloud frameworks (Figure 3, p. 4). Functions are deployed (Sec. 4.1) and invoked with a user-oriented and high-level C++ programming interface (Sec. 4.2). rFaaS is not only a standalone serverless platform — it is designed as a pluggable component into existing cloud systems (Sec. 4.3), and we provide a simple yet effective billing procedure (Sec. 4.4).

4.1 Function Deployment

Listing 1 presents the standard function interface in rFaaS. The function input is written to memory buffers of the user code executor (➊), while the RDMA immediate value contains an invocation identifier and a function index. The function returns the number of bytes in the output array sent back to the client. The input buffer contains a twelve-byte header with an address and access key for a buffer on the client’s side, and the executor writes the output directly to the client’s memory. Thus, users gain the flexibility to invoke functions concurrently and return results into different memory regions. rFaaS supports the execution of arbitrary stateless functions, and similarly to the function apps offered by Azure Functions [50], we enable the execution of different functions in the same worker process. We offer users two ways of distributing their serverless functions: code package and executor threads (➋). Alternatively, users can deploy a Docker image containing the function’s code and dependencies. The image is enriched with rFaaS RDMA executor and placed in a hosted Docker registry (➌).

4.2 Programming Model

To design the programming interface for rFaaS, we take inspiration from recent developments in the C++ standard for parallel and asynchronous executors [95]. The prior work on executors and their implementations proved that this concept is an efficient interface for dispatching tasks to accelerator devices [96, 97]. The programming model presented in Figure 5 hides the complexity of RDMA verbs under a lightweight C++ abstraction. As a result, it can be easily integrated into existing parallel applications as presented in Listing 2, and it can be adapted in the future to full compatibility with C++.

| Memory Alloc | RDMA | Invoker | Register | Connect | Submit | Send | Recv | Queue Pair |
|--------------|------|---------|----------|---------|--------|------|------|------------|
| allocate     |      | execute | notify   |         |        |      |      |            |

Figure 5: The rFaaS programming model. The model is inspired by C++ standardization efforts on the executor concept.

Listing 2 Example of an rFaaS-accelerated application.

```cpp
void compute(int size, options & opts) {
    rfaas::invoker invoker{opts.rnic_device};
    invoker.allocate(opts.lib, opts.size * sizeof(double), rfaas::invoker::ALWAYS_WARM_INVOCATIONS);
    auto alloc = invoker.allocate<double>();
    // Automatically expanded with function's header
    rfaas::buffer<double> in = alloc.input(2 * size);
    rfaas::buffer<double> out = alloc.output(2 * size);
    // Offload part of the computation to rFaaS
    auto f = invoker.submit("task", in, size, out);
    local_task(in.data() + size, out.data() + size);
    f.get();
    invoker.deallocate(); // Release computing resources.
}
```

| Memory Alloc | Invoker | Promises, Futures |
|--------------|---------|-------------------|
| allocate     | execute | notify            |

void compute(int size, options & opts) {
    rfaas::invoker invoker{opts.rnic_device};
    invoker.allocate(opts.lib, opts.size * sizeof(double), rfaas::invoker::ALWAYS_WARM_INVOCATIONS);
    auto alloc = invoker.allocate<double>();
    // Automatically expanded with function's header
    rfaas::buffer<double> in = alloc.input(2 * size);
    rfaas::buffer<double> out = alloc.output(2 * size);
    // Offload part of the computation to rFaaS
    auto f = invoker.submit("task", in, size, out);
    local_task(in.data() + size, out.data() + size);
    f.get();
    invoker.deallocate(); // Release computing resources.
}
executor while verifying the token in the background with the help of the resource manager (3); only the cloud-managed resource manager has permission to access the token verification system. After a successful verification, the resources are released to the client, and the resource manager shares remote billing buffer details with the lightweight allocator. Thus, rFaaS makes no assumptions on the provided authorization systems, the credentials verification is conducted entirely on the cloud provider side, and the RDMA token transmission can be encrypted and secured with modern systems [76].

4.4 Billing

The pricing of rFaaS is presented in the equation below, and it includes three basic cost components: allocation of computing resources $C_a$, hot polling $C_h$, and active computation time $C_c$.

$$ C = C_a \cdot t_a + C_h \cdot t_h + C_c \cdot t_c $$

The total allocation $t_a$ measured in GB-second is calculated across all executors as a product of allocation time and memory requested. Whereas the active computation time $t_c$ and hot polling time $t_h$, measured in seconds, represent the total time all remote workers are busy with executing functions and polling for new invocations, respectively. Costs $C_a$, $C_h$, and $C_c$ are measured in $ per GB-second and $ per second and represent the total cost of occupying and actively using system resources (i.e., cores, memory) for a given period. The pricing system of rFaaS is similar to traditional FaaS systems for provisioned function invocations [100], where clients are charged for the pre-allocated cloud resources. However, unlike traditional FaaS, rFaaS does not charge for invocation calls, only for active CPU time. As a result, clients of rFaaS pay neither $C_c$ nor $C_h$ for times when remote executor threads sleep due to warm invocations. However, the cloud operators might increase the $C_h$ cost component to encourage warm invocations, as their low CPU overhead helps further to boost utilization through over-allocating resources on multi-tenant servers. Applications requiring the highest performance can pay the premium for nanosecond invocation overheads. The billing procedure is implemented in a global database associated with the resource manager (3). The manager exposes memory regions for RDMA atomic fetch-and-add operations, providing lightweight allocators with an RDMA-native way of accumulating cost results without consuming CPU resources. We accumulate charges with a granularity of one second, and billing data is updated with the same frequency. This avoids a loss of accounting data due to the abrupt termination of rFaaS spot executors. The contention of atomic operations is not an issue, as cost accumulation is never on the critical path of function invocation.

5 rFaaS in Practice

To demonstrate the fitness of rFaaS for high-performance programming, we answer critical questions in the form of extensive evaluation.

1. Is rFaaS fast enough for latency-sensitive, high-performance applications?
2. Are the overheads for initialization prohibitively large?
3. Does rFaaS scale with larger messages?
4. Does rFaaS scale with more workers?
5. Does rFaaS help to integrate functions into latency-sensitive applications?
6. Does rFaaS improve performance for typical high-performance applications?
7. Is the performance of remote computing with rFaaS competitive when compared to local computation?

Platform For the purpose of this evaluation, we deploy rFaaS in a local cluster and execute benchmark code on 4 nodes, each with two 18-core Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz and 377 GB of memory. Nodes are equipped with Mellanox MT27800 Family NIC with a 100 Gb/s Single-Port link that is configured with RoCEv2 support. Nodes communicate with each other via a switch, and we measured an RTT latency of 3.69 µs and a bandwidth of 11,686.4 MiB/s. We use Docker 20.10.5 with the executor image ubuntu:20.04, and we use the Mellanox’s SR-IOV plugin to run containers over virtual device functions. rFaaS is implemented in C++, using g++ 8.3.1. MPI benchmarks are implemented using OpenMPI 4.0.5.

5.1 Invocation Latency

![Figure 6: The RTT of an no-op rFaaS function and network transport, median (solid) and 99th latency (dashed).](image)

We begin with the most important characteristic for rFaaS: the latency of invoking a remote function. We measure hot and warm invocations of a non-op "echo" function that returns the provided input. We use a warmed-up, single-threaded, bare-metal executor with the main thread pinned to a CPU core, perform 10,000 repetitions, and report the median. We compute the non-parametric 99% confidence intervals of the median [101, 102], and find that the interval bounds are very tight (<1%). To assess the overheads of rFaaS invocations, we measure the latency of RDMA and TCP/IP transmissions. For the former, we use ib_write_lat from the perftest package, execute it with thread pinning and warm-up iterations, and report the median. For the latter, we use netperf with page-aligned buffers and process pinning, and report the mean.
Figure 6 shows that the overhead of a no-op function in rFaaS in a process is 326 ns on average, when compared to RDMA writes. The measurements for a Docker-based executor present additional ca. 50 ns overhead over RDMA writes when using a container. The only exception is the message size of 128 bytes, where the overhead increases to 630 ns. There, RDMA can use message inlining for both directions of the transmission which improves the performance of small messages significantly [82]. However, the communication in rFaaS is asymmetrical: we transmit 12 bytes more for the input. The maximal supported inlining size is 128 bytes on our device, forcing rFaaS to use non-inlined write operations for one direction. The average overhead of a warm execution is 4.67 µs. Here, the containerization adds a measurable overhead, and Docker-based warm executions have an additional latency of ca. 650 ns. 

With slightly more than 300 ns of overhead, rFaaS enables remote invocations with no noticeable performance penalty, conclusively answering: rFaaS is fast enough for latency-sensitive, high-performance applications.

![Figure 6: Cold invocations of rFaaS functions.](image)

5.2 Cold Invocation Overheads

Figures 7a and 7b present the overhead of a single cold invocation on a bare-metal and Docker-based executor, respectively. The data comes from 1000 invocations with a single no-op C++ function, compiled into a shared library of size 7.88 kB. In all tested configurations, the longest step is the creation of workers. All other steps: the connection establishment to the manager, submitting an allocation and code, and code invocation, take single-digit milliseconds to accomplish. We can therefore claim that rFaaS does not introduce significant overheads apart from the sandbox initialization.

While the current version of Docker shows an overhead of approximately 2.7 seconds to spawn workers, low-latency approaches such as Firecracker [91] exist, that reduce this time to as little as 125 milliseconds.

In HPC applications, the initialization time of MPI takes between 0.8 and 3 seconds for 32 processes depending on the MPI library used [103] — comparable to the rFaaS overhead of cold initialization. Developers can overcome this overhead by initializing rFaaS like calling `MPI_Init()`, and the non-blocking nature of this process should allow rFaaS workers to be ready to by the time MPI completes its own initialization.

We therefore claim that cold invocation overheads of rFaaS do not pose an obstacle for the use in HPC.

5.3 Scalability with payload size

To compare the performance of rFaaS and other platforms, we evaluate a non-op C++ function that returns the provided input on a payload range from 1 kB to 5 MB. Since other platforms cannot accept raw data, we generate a base64-encoded string that approximately matches the input size used in rFaaS.

We compare against AWS Lambda [49], a state-of-the-art commercial FaaS solution, as Azure Functions [50] and Google Cloud Functions [51] do not support C++ functions, OpenWhisk [57], an open-source FaaS platform, Nightcore [28], a low-latency serverless platform. In Lambda, we deploy a native function implemented using the official C++ Runtime [104], we expose an HTTP endpoint with no authorization, and run the experiment in an AWS t2.micro VM instance in the same region as the function. We deploy on the cluster a standalone OpenWhisk using Docker with Kafka and API gateway [105]. A C++ function in OpenWhisk is invoked as a regular application, accepting inputs no larger than 125 kB through `argc` and `argv`. We deploy a `nightcore` instance on the cluster with a non-op C++ function.

We present the evaluation result in Figure 1. On all payload sizes, rFaaS clearly provides significantly better performance. rFaaS invocations are between 695x and 3,692x faster than AWS Lambda executions, thanks to the performance attainable with a low-latency network and the native support for transmitting raw data that suits high-performance applications very well. rFaaS is between 23x and 39x times faster than Nightcore, another FaaS platform with microsecond-scale latencies. Similarly, rFaaS provides a speedup between 5,904x and 22,406x when compared to OpenWhisk.

Therefore, we answer that rFaaS provides significant performance improvements over contemporary FaaS platforms and rFaaS scales well with message size.

5.4 Scalability with parallel workers

To verify that RDMA-capable functions scale efficiently to handle integration into scalable applications, we place managers on 36-core CPUs and evaluate parallel invocations. We execute the no-op function on warmed-up, bare-metal executors having allocated from 1 to 32 worker threads.
Figure 8 presents the round-trip latencies for invoking functions with 1 kB and 1 MB payloads, respectively. The overhead of handling many concurrent connections is insignificant on hot invocations with a smaller payload. While the Docker executor shows performance increase (hot) and decrease (warm) on the 1 kB payload, the difference on 1 MB payload is less than 1%. However, execution times increase significantly with the number of workers when sending 1 MB data, due to saturating network capacity (100 Gb/s). This shows that rFaaS scaling is limited only by the available bandwidth.

Therefore, we claim that parallel scaling of rFaaS executors is bounded only by network capacity.

Figure 8: rFaaS invocations on parallel executors.

We want to answer the next question: can toolname offload computations efficiently to remote serverless workers? To answer this question, we study offloading of massively parallel computations with significant data movement. We select the Black-Scholes solver [109] from the PARSEC suite [110] parallelized with OpenMP threading. Black-Scholes solves the same partial differential equation for different parameters, and we dispatch independent equations to bare-metal parallel executors. We evaluate the benchmark with approx. 229 MB of input and 38 MB of output and present results in Figure 10.

We show that offloading the entire work to rFaaS scales efficiently compared to OpenMP, as long as the workload per thread is not close to the network transmission time of approximately 20 ms. We can further speed the OpenMP application up by offloading half of the work to the same number
of serverless functions ($OpenMP + rFaaS$). Since other high-performance FaaS systems achieve a fraction of available bandwidth (Sec. 5.5), their runtime will be dominated by the transmission of 229MB of data to functions. Thus, we can conclude that $rFaaS$ offers scalable parallelism bounded by network performance only.

6 Related Work

High-performance FaaS

FuncX [111] is a federated and distributed FaaS platform designed to bring serverless function abstraction to scientific computing. Nonetheless, FuncX does not take advantage of high-speed networks and implements a hierarchical and centralized design with long invocation paths between clients and remote workers. As a result, even warm invocations take at least 90ms. Nightcore [28] is a high-performance FaaS runtime that optimizes internal function calls — invocations made by a running function that can be satisfied locally, without inter-node communication.

Cloudburst [32] brings stateful computations and consistency into serverless workflows with auto-scalable key-value storage. SAND [31] is a serverless platform optimized for workflows of serverless functions through a grouping of functions and dedicated message buses for subsequent invocations. In contrast, $rFaaS$ exploits co-location via explicit parallelism of executor allocation and optimizes invocation latencies through RDMA communication. Archipelago [30] and Wukong [29] allow users to submit jobs that are represented as a directed acyclic graph (DAG) of functions. They perform latency-aware scheduling of a submitted DAG: Wukong uses a decentralized and dynamic scheduling built on top of AWS Lambda, while Archipelago focuses on resource partitioning for decentralized schedulers and optimizing the control plane. In $rFaaS$ both allocation and invocation are decentralized and optimized with a direct client-worker connection.

The improvements and optimization strategies, such as the application-layer solutions in Wukong, sandbox warming up in Archipelago, container provisioning system SOCK [112], and the fast startup sandbox Catalyzer [113], are orthogonal to $rFaaS$ and can be implemented in our platform as well.

SmartNICs have been shown to provide fast dispatching and orchestration for FaaS platforms [114]. Choi et al. presented low-latency invocations on a SmartNIC runtime [115]. However, supported functions are limited by restricted C-like implementation language and low-performance RISC cores.

Remote Invocations

Remote Procedure Calls (RPC) [116] and Active Messages [117] provide the ability to invoke a procedure remotely on another machine. Active Networks include capsules with user code that can be executed on selected routers [118]. In comparison, $rFaaS$ provides the elasticity of executing on dynamically allocated resources with the pay-as-you-go billing instead of requiring provisioned resources. We enable multi-tenant computations on a single server by providing isolation. Since $rFaaS$ clients do not send code with invocation, we provide a protection boundary between caller and callee needed to access protected cloud resources.

Boosting utilization

Many approaches attempt to boost the utilization of cloud and cluster resources [17, 119–121]. These are focused on reclaiming idle resources and colocating offline batch jobs with online and latency-sensitive cloud services, and these approaches are orthogonal to $rFaaS$ as they cannot target ephemeral resources efficiently.

Zhang et al. [48] implement an OpenWhisk load balancer optimized for harvested idle resources. This method is targeted for centralized FaaS platforms and does not provide high-performance invocations.

7 Discussion

In this paper, we introduce RDMA abstractions into FaaS to facilitate the integration of functions into high-performance and latency-sensitive applications. $rFaaS$ can positively impact other aspects of serverless systems, and we now discuss how our protocols combine with other applications and emerging solutions in FaaS.

Will serverless workflows benefit from $rFaaS$ abstractions?

Serverless workflows have emerged as the method of composing functions to build serverless native applications. Workflows require dedicated coordination and triggering services to orchestrate invocations and data propagation efficiently [122]. SmartNICs can be used for this task [114], but this solution is limited by the cost and availability of dedicated NICs. Instead, workflow orchestrators can be implemented with $rFaaS$ executors and achieve two performance goals: single-digit microsecond latency overhead of workflow invocations and efficient RMA data movement.

Could RDMA help to accelerate stateful serverless?

Stateful functions use dedicated and shared storage to circumvent the limitations of stateless computations in FaaS [32, 123–125] at the price of increased latency and costs of cloud storage. Stateful FaaS would benefit from low overheads and high scalability of RDMA-accelerated distributed remote memory [72, 126, 127]. $rFaaS$ provides RDMA abstractions needed to incorporate RMA operations on the remote storage. Furthermore, functions co-located with storage decrease latency for simple data transformations [128], and overheads of many such functions can be reduced with RDMA.

8 Conclusions

Fine-grained and granular computing need systems designed to handle microsecond-scale workloads [37, 129], but FaaS platforms still operate at the millisecond latency. $rFaaS$ attempts to solve this problem at three levels: a novel direct and decentralized scheduling to reduce serverless critical path, incorporation of high-speed networks to achieve microsecond-latency, and inclusion of remote memory access to remove overheads of the OS control plane. With RDMA-capable functions, we demonstrate hot invocations with less than one microsecond of overhead and efficient parallel scalability, paving the way for future low-latency and fine-grained computing.
References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, 2009.

[2] Wenwu Zhu, Chong Luo, Jianfeng Wang, and Shipeng Li. Multimedia cloud computing. *IEEE Signal Processing Magazine*, 28(3):59–69, 2011.

[3] Marco A. S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L. F. Cunha, and Rajkumar Buyya. Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Comput. Surv.*, 51(1), January 2018.

[4] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Jianwei Hao, Ting Jiang, Wei Wang, and In Kee Kim. An empirical analysis of vm startup times in public iaas clouds: An extended report, 2021.

[6] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[7] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[9] Robert Birke, Lydian Y. Chen, and Evgenia Smirni. Data centers in the cloud: A large scale performance study. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 336–343, 2012.

[10] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Jing Guo, Zhihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2019.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’18, page 347–360, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Matthew D. Jones, Joseph P. White, Martins Innus, Robert L. DeLeon, Nikolay Simakov, Jeffrey T. Palmer, Steven M. Gallo, Thomas R. Furlani, Michael T. Showerman, Robert Brunner, Andry Kot, Gregory H. Bauer, Brett M. Bode, Jeremy Enos, and William T. Kramer. Workload analysis of blue waters. *CoRR*, abs/1703.00924, 2017.

[15] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Alcock, and Devesh Tiwari. Job characteristics on large-scale systems: Long-term analysis, quantification, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’20. IEEE Press, 2020.

[16] Haihang You and Hao Zhang. Comprehensive workload analysis and modeling of a petascale supercomputer. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 253–271, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on
[18] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term slos for reclaimed cloud computing resources. In Proceedings of the ACM Symposium on Cloud Computing, SOCC ’14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[19] Rahul Singh, Prateek Sharma, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan. Here today, gone tomorrow: Exploiting transient servers in datacenters. IEEE Internet Computing, 18(4):22–29, 2014.

[20] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA ’15, page 450–462, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. SIGARCH Comput. Archit. News, 41(3):607–618, June 2013.

[22] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In 2015 USENIX Annual Technical Conference (USENIX ATC ’15), pages 459–471, Santa Clara, CA, July 2015. USENIX Association.

[23] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, page 248–259, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimp, T. Sivabalavan, and Rajesh Subbiah. Worth their watts? - an empirical study of datacenter servers. In HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pages 1–10, 2010.

[25] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 199–208, 2019.

[26] M. AbdelBaky, M. Parashar, H. Kim, K. E. Jordan, V. Sachdeva, J. Sexton, H. Jamjoom, Z. Shae, G. Pencheva, R. Tavakoli, and M. F. Wheeler. Enabling high-performance computing as a service. Computer, 45(10):72–80, 2012.

[27] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 202–215, 2016.

[28] 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, New York, NY, USA, 2021. Association for Computing Machinery.

[29] 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, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform, 2019.

[31] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijjaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’18, pages 923–935, USA, 2018. USENIX Association.

[32] Vikram Sreekanti, Chenggang Wu, Xiaoyue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. Proc. VLDB Endow., 13(12):2438–2452, July 2020.

[33] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. Boxer: Data analytics on network-enabled serverless platforms. In 11th Annual Conference on Innovative Data Systems Research (CIDR’21), 2021.

[34] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion
Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[35] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoeffler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware ’21. Association for Computing Machinery, 2021.

[36] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Anki Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, June 2021.

[37] Pedro Garcia Lopez, Aleksander Slominski, Michael Behrendt, and Bernard Metzler. Serverless predictions: 2021-2030, 2021.

[38] James M Kaplan, William Forrest, and Noah Kindler. Revolutionizing data center energy efficiency. *McKinsey & Company*, pages 1–13, 2008.

[39] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[40] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 94–103, 2010.

[41] Siqi Shen, Vincent Van Beek, and Alexandru Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474, 2015.

[42] Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. Characterizing private clouds: A large-scale empirical analysis of enterprise clusters. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC ’16, page 29–41, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC ’12, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Peter Garraghan, Paul Townsend, and Jie Xu. Analysis of the server characteristics and resource utilization in google cloud. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 124–131, 2013.

[45] Huan Liu. A measurement study of server utilization in public clouds. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 435–442, 2011.

[46] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[47] Akshat Verma and Gargi Dasgupta. Server workload analysis for power minimization using consolidation. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association.

[48] Yanqi Zhang, Íñigo Goiri, Gohar Ifran 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.

[49] AWS Lambda. https://aws.amazon.com/lambda/, 2014. Accessed: 2022-01-12.

[50] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2016. Accessed: 2022-01-12.

[51] Google Cloud Functions. https://cloud.google.com/functions/, 2017. Accessed: 2022-01-12.

[52] IBM Cloud Functions. https://cloud.ibm.com/functions/, 2016. Accessed: 2022-01-12.

[53] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC ’18, page 133–145, USA, 2018. USENIX Association.

[54] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188. IEEE, 2018.
[55] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In Proceedings of the 21st International Middleware Conference, Middleware ’20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[56] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Ligouri, 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.

[57] Apache OpenWhisk. https://openwhisk.apache.org/, 2016. Accessed: 2022-01-12.

[58] M. Sciabarrà. Learning Apache OpenWhisk: Developing Open Serverless Solutions. O’Reilly Media, Incorporated, 2019.

[59] Gerard París, Pedro García-López, and Marc Sánchez-Artigas. Serverless elastic exploration of unbalanced algorithms. In 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), pages 149–157, 2020.

[60] 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.

[61] Ingo Müller, Renato Marroquin, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. ArXiv, abs/1912.00937, 2019.

[62] Pedro García López, Marc Sánchez Artigas, Simon Shillaker, Peter R. Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer. Servermix: Tradeoffs and challenges of serverless data analytics. CoRR, abs/1907.11465, 2019.

[63] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winston. 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.

[64] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. CoRR, abs/1810.09679, 2018.

[65] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In Proceedings of the ACM Symposium on Cloud Computing, SoCC ’19, pages 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[66] V. Ishakian, V. Muthusamy, and A. Slominski. Serving deep learning models in a serverless platform. In 2018 IEEE International Conference on Cloud Engineering (IC2E), pages 257–262, April 2018.

[67] L. Feng, P. Kudva, D. Da Silva, and J. Hu. Exploring serverless computing for neural network training. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 334–341, July 2018.

[68] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In Proceedings of the 2017 Symposium on Cloud Computing, SoCC ’17, page 94–107, New York, NY, USA, 2017. Association for Computing Machinery.

[69] Marius Pöke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC ’15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing Using RDMA and HTM, page 87–104. Association for Computing Machinery, New York, NY, USA, 2015.

[71] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16, page 355–370, New York, NY, USA, 2016. Association for Computing Machinery.

[72] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma. In Proceedings of the 2021 International Conference on Management of Data, SIGMOD/PODS ’21, page 1811–1824, New York, NY, USA, 2021. Association for Computing Machinery.

[73] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2012.
[74] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. Polardb serverless: A cloud native database for disaggregated data centers. In Proceedings of the 2021 International Conference on Management of Data, pages 2477–2489, 2021.

[75] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. Redmark: Bypassing RDMA security mechanisms. In 30th {USENIX} Security Symposium ({USENIX} Security 21), 2021.

[76] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – efficient NIC-based authentication and encryption for remote direct memory access. In 2020 {USENIX} Annual Technical Conference ({USENIX} ATC 20), pages 691–704. {USENIX} Association, July 2020.

[77] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High performance cloud computing. Future Generation Computer Systems, 29(6):1408–1416, 2013. Including Special sections: High Performance Computing in the Cloud & Resource Discovery Mechanisms for P2P Systems.

[78] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In 16th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI 19), pages 113–126, Boston, MA, February 2019. {USENIX} Association.

[79] The Linux SoftRoCE Driver. https://www.youtube.com/watch?v=NumH5YeVjHU, 2017. Accessed: 2022-01-12.

[80] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huai Xin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. Xrdma: Effective rdma middleware in large-scale production environments. In 2019 {IEEE} International Conference on Cluster Computing (CLUSTER), pages 1–12, 2019.

[81] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys ’19, New York, NY, USA, 2019. Association for Computing Machinery.

[82] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM ’14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

[83] Werner Vogels. Eventually consistent. Commun. ACM, 52(1):40–44, January 2009.

[84] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with sr-iov. In HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pages 1–10, 2010.

[85] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wasse, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. Irma: Re-envisioning remote memory access for multi-tenant datacenters. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM ’20, page 708–721, New York, NY, USA, 2020. Association for Computing Machinery.

[86] IPsec over RoCE, Mellanox. https://community.mellanox.com/s/article/ConnectX-6DX-Bluefield-2-IPsec-HW-Full-Offload-Configuration-Guide, 2021. Accessed: 2022-01-12.

[87] libfabric, Open Fabric Interfaces. https://github.com/ofiwg/libfabric, 2021. Accessed: 2022-01-12.

[88] AWS Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/, 2021. Accessed: 2022-01-12.

[89] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. PLOS ONE, 12(5):1–20, 05 2017.

[90] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gvisor case study. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), Renton, WA, July 2019. USENIX Association.

[91] Firecracker. https://github.com/firecracker-microvm/firecracker, 2018. Accessed: 2022-01-12.

[92] M. Haseeb Javed, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda. Characterization of big data stream processing pipeline: A case study using flink and kafka. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT ’17, page 1–10, New York, NY, USA, 2017. Association for Computing Machinery.
[93] Arjun Balasubramanian and Mohammed Danish Shaikh. FASTPS: A Fast Publish-Subscribe service using RDMA. https://arjunbala.github.io/FastPS_Report.pdf. Accessed: 2022-01-10.

[94] Konstantin Taranov, Steve Byan, Virendra Marathe, and Torsten Hoefler. KafkaDirect: Zero-copy Data Access for Apache Kafka over RDMA Networks. https://spcl.inf.ethz.ch/Research/ParallelProgramming/KafkaDirect/KafkaDirect.pdf. Accessed: 2022-01-10.

[95] P0443R14: A Unified Executors Proposal for C++. https://wg21.link/p0443r14, 09 2020. Accessed: 2022-01-12.

[96] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. Closing the performance gap with modern c++. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, High Performance Computing, pages 18–31, Cham, 2016. Springer International Publishing.

[97] Marcin Copik and Hartmut Kaiser. Using sycl as an implementation framework for hpx.compute. In Proceedings of the 5th International Workshop on OpenCL, IWOCL 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[98] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 437–450, Denver, CO, June 2016. USENIX Association.

[99] P. MacArthur and R. D. Russell. A performance study to guide rdma programming decisions. In 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems, pages 778–785, 2012.

[100] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/, 2020. Accessed: 2022-01-12.

[101] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’15, New York, NY, USA, 2015. Association for Computing Machinery.

[102] Jean-Yves Le Boudec. Performance evaluation of computer and communication systems. Epfl Press, 2010.

[103] Weikuan Yu, Jiesheng Wu, and Dhabaleswar K. Panda. Fast and scalable startup of mpi programs in infiniband clusters. In Luc Bougé and Viktor K. Prasanna, editors, High Performance Computing - HiPC 2004, pages 440–449, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[104] AWS Lambda C++ Runtime. https://github.com/awslabs/aws-lambda-cpp, 2020. Accessed: 2022-01-12.

[105] Apache OpenWhisk standalone Docker image. https://hub.docker.com/r/openwhisk/standalone, 2020. Accessed: 2022-01-12.

[106] Alexander Brace, Hyungro Lee, Heng Ma, Anda Trifan, Matteo Turilli, Igor Yakuskin, Todd Munson, Ian Foster, Shantenu Jha, and Arvind Ramanathan. Achieving 100x faster simulations of complex biological phenomena by coupling ml to hpc ensembles, 2021.

[107] Michael Wang, Tingjun Yang, Maria Acosta Flechas, Philip Harris, Benjamin Hawks, Burt Holzman, Kyle Kloepfel, Jeffrey Krupa, Kevin Pedro, and Nhan Tran. Gpu-accelerated machine learning inference as a service for computing in neutrino experiments. Frontiers in Big Data, 3:48, 2021.

[108] Jeffrey Krupa, Kelvin Lin, Maria Acosta Flechas, Jack Dinsmore, Javier Duarte, Philip Harris, Scott Hauck, Burt Holzman, Shih-Chieh Hsu, Thomas Kljinsma, Mia Liu, Kevin Pedro, Dylan Rankin, Natchanon Suaysom, Matt Tram, and Nhan Tran. GPU coprocessors as a service for deep learning inference in high energy physics. Machine Learning: Science and Technology, 2(3):035005, apr 2021.

[109] Alexander Heinecke, Stefanie Schraufstetter, and Hans-Joachim Bungartz. A highly parallel black–scholes solver based on adaptive sparse grids. Int. J. Comput. Math., 89(9):1212–1238, June 2012.

[110] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT ’08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[111] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, HPDC ’20, page 65–76, New York, NY, USA, 2020. Association for Computing Machinery.
[112] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 57–70, Boston, MA, July 2018. USENIX Association.

[113] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[114] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Speedo: Fast Dispatch and Orchestration of Serverless Workflows, page 585–599. Association for Computing Machinery, New York, NY, USA, 2021.

[115] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ-nic: Interactive serverless compute on programmable smartnics. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pages 67–77, 2020.

[116] Bruce Jay Nelson. Remote procedure call. Carnegie Mellon University, 1981.

[117] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In [1992] Proceedings the 19th Annual International Symposium on Computer Architecture, pages 256–266, 1992.

[118] David Wetherall. Active network vision and reality: Lessons from a capsule-based system. In Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP ’99, page 64–79, New York, NY, USA, 1999. Association for Computing Machinery.

[119] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 285–300, Broomfield, CO, October 2014. USENIX Association.

[120] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In NSDI, volume 11, pages 22–22, 2011.

[121] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. SIGPLAN Not., 49(4):127–144, February 2014.

[122] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. Proc. ACM Program. Lang., 5(OOPSLA), oct 2021.

[123] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In Proceedings of the 20th International Middleware Conference, Middleware ’19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[124] Marcin Copik, Alexandru Calotoiu, Rodrigo Bruno, Roman Böhlinger, and Torsten Hoefler. Process-as-a-service: Faas stateful computing with optimized data planes. Accessed: 2022-01-10.

[125] 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, 2020.

[126] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Déguildard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyan, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In 2018 {USENIX} Annual Technical Conference ({ATC}’18), pages 775–787, 2018.

[127] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 401–414, Seattle, WA, April 2014. USENIX Association.

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

[129] Collin Lee and John Ousterhout. Granular computing. In Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS ’19, page 149–154, New York, NY, USA, 2019. Association for Computing Machinery.