rFaaS: Enabling High Performance Serverless with RDMA and Leases

Marcin Copik*, Konstantin Taranov†, Alexandru Calotoiu*, Torsten Hoefler*
*Department of Computer Science, ETH Zürich, Zürich, Switzerland
†Microsoft
*firstname.lastname@inf.ethz.ch, †kotaranov@microsoft.com

***Abstract*—High performance is needed in many computing systems, from batch-managed supercomputers to general-purpose cloud platforms. However, scientific clusters lack elastic parallelism, while clouds cannot offer competitive costs for high-performance applications. In this work, we investigate how modern cloud programming paradigms can bring the elasticity needed to allocate idle resources, decreasing computation costs and improving overall data center efficiency. Function-as-a-Service (FaaS) brings the pay-as-you-go execution of stateless functions, but its performance characteristics cannot match coarse-grained cloud and cluster allocations. To make serverless computing viable for high-performance and latency-sensitive applications, we present rFaaS, an RDMA-accelerated FaaS platform. We identify critical limitations of serverless - centralized scheduling and inefficient network transport - and improve the FaaS architecture with allocation leases and microsecond invocations. We show that our remote functions add only negligible overhead on top of the fastest available networks, and we decrease the execution latency by orders of magnitude compared to contemporary FaaS systems. Furthermore, we demonstrate the performance of rFaaS by evaluating real-world FaaS benchmarks and parallel applications. Overall, our results show that new allocation policies and remote memory access help FaaS applications achieve high performance and bring serverless computing to HPC.**

***Index Terms*—Serverless, Function-as-a-Service, High-Performance Computing, RDMA**

rFaaS Implementation: https://github.com/spcl/rFaaS
rFaaS Artifact: https://zenodo.org/record/7657524

I. Introduction

The high-performance computing landscape is dominated by the Message-Passing Interface (MPI), the *de facto* standard distributed programming paradigm. With job batch scheduling and shared–memory frameworks for multithreading, MPI is the leading use case for clusters and supercomputers [1]. In the rigid HPC world, applications with varying parallelism achieve lower efficiency because adapting resource allocation to changing requirements is heavily constrained [2, 3]. On the other hand, the cloud has brought a major innovation in elastic resource management. While cloud computing has the hardware capability to support high-performance workloads, it lacks programming models for flexible parallelism. Thus, high-performance applications overprovision computing resources and increase the data center underutilization, a problem that has a significant impact as *”increasing utilization by a few percentage points can save millions of dollars”* [4]. Furthermore, HPC units in the cloud can be substantially more expensive than existing supercomputing resources[1]. The rigid structure of supercomputing systems limits their efficiency, and HPC applications cannot benefit from cloud elasticity due to lack of flexible frameworks.

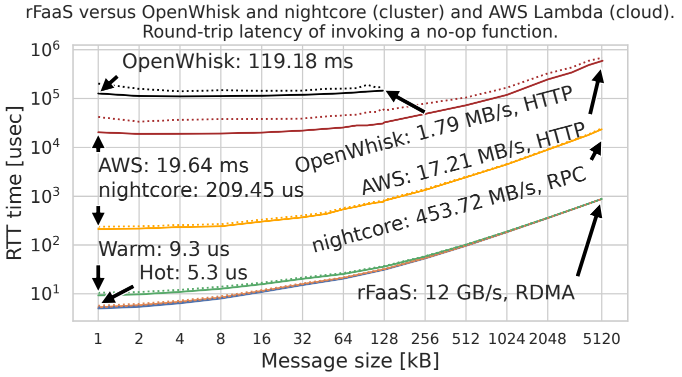


Fig. 1: **The remote invocations of an empty C++ function on serverless platforms and *rFaaS*: median (solid), and 99th latency (dashed) for a single function (details in Sec. V-C).**

The performance gap between public clouds and HPC data centers has been shrinking over time. The performance and attractiveness of *Infrastructure-as-a-Service (IaaS)* resources has improved [5–7]. Furthermore, virtual machines and containers have been found to be an efficient abstraction level for high-performance applications [8, 9]. Thus, parallel applications running in private data centers and scientific supercomputers started taking advantage of vast cloud resources [6]. However, high–performance applications must provision resources for the peak demand. Thus, they employ complex and domain-specific optimizations or resort to overprovisioning and underutilizing computing resources. As a result, the resource utilization in public clouds and private data centers, such as supercomputers, has always been low for both computing and memory resources (Sec. II-A). HPC users want to use flexible allocations and achieve higher efficiency without sacrificing performance guarantees. Unfortunately, the question of incorporating elastic resources into HPC frameworks remains open.

To bridge the gap between HPC clusters and the cloud, we must incorporate elastic allocations into supercomputers and decrease HPC computation costs in the cloud. This

[1]For example, nodes of the Piz Daint supercomputer start at 0.53 CHF (approximately 0.55 USD) per nodehour. Comparable pay-as-you-go cloud instances cost at least 1.5 and as high as 4.08 USD. Cheaper cloud instances require longer-term allocations for many months and provide no flexibility.

new programming model would need to satisfy two essential requirements. First, high-performance computing units should be immediately available to address workload and load balance changes. However, even as of 2021, acquiring new *Infrastructure-as-a-Service (IaaS)* resources in the cloud takes minutes, not seconds [10]. Similarly, HPC batch systems do not support dynamic resource reallocation. Second, the model should allow users to effectively use idle resources. These can be offered at a significantly lower cost since decreasing resource waste increases the overall efficiency of a data center. However, the rapid and frequent utilization changes on many such platforms (Fig. 2) indicate that persistent and long-running allocations cannot address these idleness gaps. Thus, the programming model should use fine-grained and ephemeral workers to allocate resources with short availability.

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 completely manages their scheduling. 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. Functions are used as elastic workers to fulfill *Service Level Objective (SLO)* requirements [11], and they could implement the *HPC-as-a-Service* [12]. However, functions must overcome crucial performance challenges to become a viable programming model for HPC (Table I).

We address these challenges in ***rFaaS***, an RDMA-capable serverless platform tailored to the requirements of HPC applications (Sec. III). We define new **RDMA abstractions** that hide the network stack complexity and preserve the elasticity and isolation of serverless. We improve serverless architecture with three innovations that integrate into existing designs. First, *rFaaS* employs a new resource management policy where **leases** replace centralized placement of invocations. Instead of routing every function to the same warm containers, leases allow to skip the control logic. Then, we accelerate the 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 microsecond latency invocations, we replace HTTP and REST interfaces with an **RDMA function dispatch protocol** that removes the milliseconds of OS latency [13]. We show **hot** invocations with an overhead of a little over 300 nanoseconds on top of the fastest network (Fig. 1).

We present a **C++ programming model** for straightforward integration of *rFaaS* functions into high-performance applications (Sec. IV). Our work is a major step towards increasing 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, functions, and HPC applications (Sec. V).

Our paper makes the following contributions:

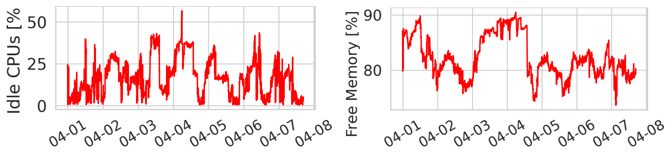
- We present the design and open-source implementation of the first RDMA-capable serverless platform, including (1) new FaaS resource management and (2) a novel, low-latency, and zero-copy *hot* type of serverless invocations.
- We conduct an experimental verification against state-of-the-art open-source and commercial serverless platforms summarized in Fig. 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 demonstrate *rFaaS* usability with real-world serverless functions and show how the invocation latency is sufficient to accelerate HPC applications.

Requirements	rFaaS	Other solutions.
Low-latency invocations	🖒	Nightcore [14]
Direct allocations	🖒	👎
High-speed networks	🖒	👎
Decentralized scheduling	🖒	Wukong [15], Archipelago [16]
Efficient workflows	👍	SAND [17], Wukong [15], Cloudburst [18].
Direct communication	👍	Boxer [19]
Fast and shared storage		Open problem.
Affordable costs		Open problem.
Consistent performance		Open problem.

TABLE I: ***rFaaS*** **solves (🖒) and enables solutions (👍) to the major challenges of high-performance FaaS [20–23].**

II. Background

rFaaS solves the utilization problems of data centers by identifying the opportunity to reuse idle resources (Sec. II-A). At the same time, modern FaaS platforms are too constrained (Sec. II-B) to take advantage of high-speed networks and remote memory operations (Sec. II-C), motivating improvements to the serverless architecture (Sec. III).



(a) Idle CPU cores rate (%). (b) Free memory rate (%).

Fig. 2: **Piz Daint supercomputer utilization on 31.03-7.04 2021: querying SLURM with a one-minute interval.**

A. Resource Utilization

Low resource utilization has always affected data centers and it had a vast impact on the financial efficiency of the system: wasted capital investments into idle resources and increased operating costs, as the energy usage of servers doing little and no work is more than 50% of their peak power consumption [24]. In highly competitive and batch-managed supercomputers, the average utilization of nodes varies between 80% and 94% [25–27]. Furthermore, on average three-quarters of the memory in HPC nodes is not utilized [28]. We observed similar underutilization problems in the supercomputing system Piz Daint. Since the idle nodes are available for a short time (Fig. 2a), opportunistic reuse for other computations must be constrained to short-running workloads. To support incoming large-scale jobs, reclaimed resources must be transient and easily retrievable by the batch system. Fortunately, large quantities of idle node memory open

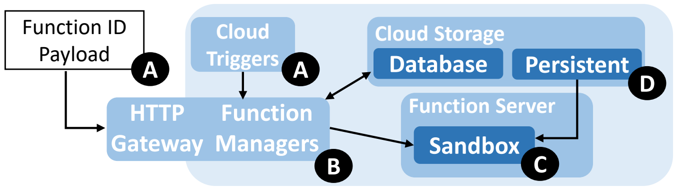


Fig. 3: **A high-level view of the FaaS architecture.**

the possibility of hosting the warm state of ephemeral workers (Fig. 2b).

Observation Stateless and short-lived functions are a natural fit for *opportunistic* computing, and *rFaaS* can employ *ephemeral* HPC resources.

B. *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.

Platform We characterize the FaaS platforms with a high-level overview presented in Fig. 3 and refer interested readers to a wider discussion in the literature [20, 21, 29]. 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.

Invocations *Cold* invocations occur when no idle sandbox is available for a given function, and must allocate a new one. The latency includes an allocation of a new one, downloading the function code from external storage, and starting an executor process. In a *warm* invocation, the function payload is sent directly to the executing process. While lightweight virtual machines are designed to support burstable serverless invocations, even warm invocations can incur significant overheads. On AWS Lambda, each invocation is processed by a dedicated management service to decide function placement [30]. The function input is limited to a few megabytes, so users must transmit larger payloads via the high-latency cloud storage. The critical path is even longer in OpenWhisk, as it includes a controller, database, load balancer, and a message bus [31]. Even though platforms optimize functions to execute in the same set of warm containers, each invocation includes the repetitive placement logic of control plane.

In AWS Lambda, the RTT latency changes from 19.5 ms on 1kB to over 600 ms on 5MB, and it varies between 30 ms and 75 ms on the size range typical for images passed to ML recognition functions (Fig. 1). Since routing and allocation takes at most 10 ms in warm invocations [30], the latency is dominated by network transmission. Following Amdahl's law, utilizing the fast network is the best opportunity to decrease serverless invocations costs by orders of magnitude.

Observation 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 While the elastic parallelism of FaaS has been used in compute-intensive workloads such as data analytics, video encoding, and machine learning training [22, 32, 33], it has only gained minor traction so far in high-performance and scientific computing [34] due to a lack of low-latency communication and optimized data movement. Although recent research improved serverless performance by including RPC [14], exploiting data locality, and co-locating invocations [17], latency-sensitive and parallel applications need fast remote invocations to achieve high scalability.

Observation 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.

C. *Remote Direct Memory Access*

RDMA-capable networks have become a standard tool for implementing high-performance communication libraries, distributed protocols, storage, and databases. Unlike in the TCP/IP stack, RDMA transfers are performed entirely by a dedicated network controller bypassing both the CPU and operating system. Data is forwarded over the PCI bus to the memory, allowing the communicating endpoints to directly read, write and atomically update memory of its remote counterpart. 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. Error-handling is solely the programmer's responsibility and achieving the best performance requires fine-tuning such as aligning memory, controlling device buffers, locking memory, and utilizing vendor-specific optimizations. RDMA devices are accessed in multi-tenant environments through PCI passthrough, paravirtualization, and virtual device functions [35].

Observation Cloud and HPC applications take advantage of low overhead and high performance of RDMA networks. *rFaaS* adapts FaaS computing to be RDMA-compatible.

III. RDMA-based Serverless Platform

rFaaS tailors serverless architectures to the needs of high-performance applications. In *rFaaS*, we combine the best of two worlds - resource flexibility offered by FaaS computing with the low overhead communication primarily available in

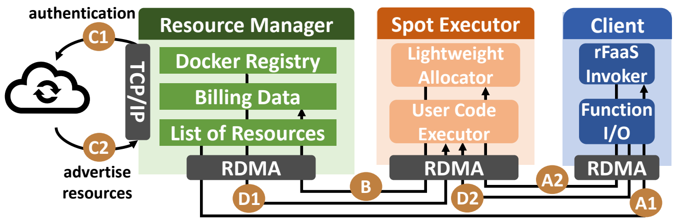


Fig. 4: ***rFaaS*: resource manager interacts with data center resources (C) and manages billing (B), clients acquire FaaS leases (A), and invoke functions (D).**

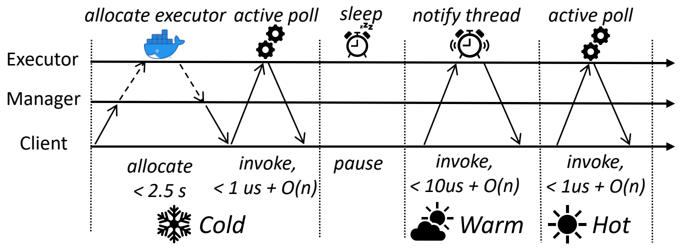


Fig. 5: **Lifetime of a function in *rFaaS*. Cold start times are dominated by sandbox initialization. Warm and hot invocation times include *rFaaS* overhead and latency of RDMA write of N bytes of payload.**

the cloud IaaS resources and HPC clusters. *rFaaS* improves the central FaaS paradigm of remote executions by replacing the REST and RPC invocations with direct memory operations on remote servers. The enhanced architecture provides the same semantics of executing user code on ephemeral workers while avoiding the major performance overheads of serverless.

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. Compared to other architectures (Sec. II-B), we limit resource allocation and authorization to cold startups, and remove both message queues and the bus for all warm invocations. First, we introduce **leases** to optimize the repeated allocation logic of FaaS control plane (Sec. III-B). Instead, our functions gain a direct RDMA connection to the user code executor without sacrificing their *serverless* nature (Fig. 4). As in other FaaS platforms, no specific assumptions about the underlying system and hardware are made. We capitalize on this gain further by implementing an RDMA-based invocation designed to minimize latency (Sec. III-C) and handle parallel executions (Sec. III-D).

A. *Components of* rFaaS

Resource Manager *rFaaS* optimizes the FaaS control plane by splitting allocation and invocation to avoid repeated function placement in the same small group of containers. Instead, clients request and receive leases on spot executors (A1). Cluster operators add and remove idle resources to the manager (C2), and each instance of resource manager is responsible for a subset of spot executors. With the lease concept, managers achieve the same load balancing and oversubscription targets as in other platforms, while keeping the overhead low - they are not involved in warm and hot invocations, which constitute the majority of serverless operations. Mangers use heartbeats to verify the status of spot executor, and announce to clients the lease termination to support fast resource reclamation.

Spot Executor When clients begin offloading tasks to *rFaaS*, they acquire leases on 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 connect to the lightweight allocator (A2), which 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. The allocator initializes an isolated execution context with an RDMA-capable execution process. Finally, clients can establish a direct RDMA connection with each executor process and invoke functions by writing function header and payload directly into their memory (D2). The results are returned to the client in a similar fashion, and the client caches the lease for consecutive invocations on warmed-up resources.

B. *Allocation Leases*

Decentralized resource management in form of leases is another improvement that *rFaaS* brings to serverless. To execute a function, clients involve the resource manager only once to acquire leases (A1). Clients cache connections to executor processes and use them for consecutive executions on warmed-up resources, helping to avoid the initialization costs for reliable RDMA connections. To support straightforward deallocation of on-demand executors, clients use the connection status to check if the process is alive. When users terminate the allocation before the lease expires, executors notify the manager to include their resources in future allocations.

C. *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 the data canter, it would be counterproductive to incorporate *rFaaS* functions into high-performance applications if we did not offer fast invocations. In addition to standard FaaS *cold* and *warm* invocations, we provide a new *hot* invocation type that guarantees zero-copy execution on pre-allocated hardware (Fig. 5).

Cold The *cold* invocation includes significant overheads caused by the initialization of an execution context. In *rFaaS*, clients acquire leases by requesting the desired core count, memory, and timeout for the allocation. Then, the lightweight allocator initializes an isolated execution sandbox and assigns the requested computing and memory resources to it. The user code executor starts in the sandbox, accesses the selected RDMA device, registers memory buffers, and 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. When the initialization is done, the client receives 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 (Sec. V).

Warm The client transmits the function payload using an RDMA connection to an allocated executor. Executor threads do not share RDMA resources, and they use blocking wait independently to receive completion events corresponding to new *warm* invocation requests. Using blocking wait increases latency but significantly decreases the pressure on computing resources compared to active polling. In the unlikely case of resource exhaustion on the executor, the invocation request is immediately rejected and the client redirects the invocation to another executor (Fig. 6). Compared to native RDMA performance of a round-trip communication, warm invocations have an overall overhead of fewer than 6 microseconds.

Hot The novel *hot* invocation improves 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. Executors can roll back to *warm* executions to free up the CPU after a configurable time without a new invocation, depending on user preferences. 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. In exchange, users gain always available computing resources, helping to incorporate functions into HPC applications and support iterative invocations.

D. 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 number of spot executors and clients in *rFaaS* is bounded by the size of the RDMA network. Since the network throughput of RDMA connections decreases significantly with the number of clients [36], the resource manager is replicated like in other FaaS platforms. While modern RDMA networks and supercomputers count many thousands of clients [13] the networks can scale globally in future cloud deployments. Resources are split between manager instances and round-robin scheduling allows handling the increasing number of lease requests from clients.

Parallel Invocations *rFaaS* allows for simultaneously dispatching function execution requests to threads of remote user executors. The user requests how many function instances should be used, and the client library manages lease allocations to reach the desired scale. The client has a direct RDMA connection to each thread worker and can invoke functions concurrently. Function workers operating on the same node are independent of each other and can all execute different functions. The scalability is achieved by exploiting the non-blocking nature of RDMA write operations and using disjoint memory buffers to store results. Multiple RDMA connections improve network utilization as more processing units of a network controller are involved [36]. Each executor thread switches between hot and warm invocations on its own, further aiding elasticity.

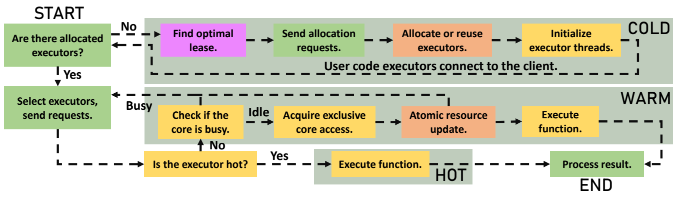

START
Are there allocated executors?
No
Find optimal lease.
Send allocation requests.
Allocate or reuse executors.
Initialize executor threads.
COLD
User code executors connect to the client.
Yes
Select executors, send requests.
Busy
Check if the core is busy.
Idle
Acquire exclusive core access.
Atomic resource update.
Execute function.
WARM
No
Is the executor hot?
Yes
Execute function.
HOT
Process result.
END

Fig. 6: ***rFaaS* invocations include RDMA communication between clients (🟢), resource manager (🟣), and spot executors (🟠) with user code executors (🟡).**

Oversubscription FaaS platforms oversubscribe resources since invocations often arrive independently, at different times, and consume different resources. This aligns well with the environments of scientific clusters, where large amounts of free memory can be used to retain more warm sandboxes than available CPU cores. However, many HPC applications cannot tolerate the overhead and imbalance introduced by oversubscribed execution, even if such an event is unlikely.

To that end, *hot* invocations ensure that the executor occupies the CPU core and handles the request immediately (Fig. 6). Since this guarantee is not always needed and can be expensive. *warm* invocations are executed opportunistically on resources that might be oversubscribed. A successful warm invocation requires only a single, local RDMA communication between spot executors and its allocator to verify resource status, thus the additional latency is negligible for larger function payloads. When resources are unavailable, the request is rejected, and the client sends it again to another user code executor. Thanks to the RDMA networking, the rejection can be processed with microsecond latency, minimizing the performance hit. Warm invocations avoid interference with compute-intensive tasks on the same CPU core since rejection is a short and I/O-intensive process. Hot and warm invocations can be switched dynamically, providing the performance and flexibility needed for all types of HPC computations.

E. Isolation and Security

rFaaS aims to provide the same security level as serverless invocations in the cloud. Multi-tenant environments require that functions execute in isolation, and the user's code is prohibited from accessing any resources, data, and code not provided with the invocation. Thus, in addition to bare-metal executors, we include containerized executors. The main requirements imposed by *rFaaS* are virtualization support for RDMA-capable network controllers and negligible performance overheads. *rFaaS* uses Docker containers to implement isolated execution contexts for user functions. Single Root I/O Virtualization (SR-IOV) provides high-performance virtu-

Listing 1 rFaaS function interface.

```
uint32_t f(void* in, uint32_t size, void* out) {
 uint32_t in_len = size / sizeof(double);
 double* input = reinterpret_cast<double*>(in);
 double* output = reinterpret_cast<double*>(out);
 // Run function's code.
 uint32_t out_len = solve(input, in_len, output);
 // Return value defines the output size
 return sizeof(double) * out_len;
}
```

alized network controllers in a multi-tenant environment [37]. On platforms without SR-IOV, we can use software virtualization systems such as FreeFlow [38].

rFaaS leases shift the control plane involvement from each execution to cold invocations, but they do not introduce additional security challenges. Leases are time–limited and include user authentication. Thus, they are similar to batch system allocations that release resources to a job and perform authorization only once. Furthermore, modern RDMA extensions provide authentication, payload encryption, and memory protection ensuring secure transmission in multi-tenant networks [39–41].

F. 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.

a) Network: While our implementation manages RDMA networks with `ibverbs`, the *rFaaS* functionality is orthogonal to the device interface and can be implemented with higher-level concepts from `libfabric` [42]. *rFaaS* can be deployed on other networks providing RDMA-like semantics, such as the Elastic Fabric Adapter in the AWS cloud [43]. In addition, software virtualization can be employed in data centers without high-speed networks, offering RDMA semantics at the cost of higher overheads [38, 44].

b) Language: *rFaaS* supports C and C++ functions and native integration into C/C++ applications (Sec. IV). 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.

c) Sandbox: *rFaaS* functions can be served in other environments than bare-metal processes or Docker containers, e.g., in HPC container Singularity [45], gVisor [46], and in microVMs such as Firecracker [30, 47] that provide a higher level of isolation with negligible performance overheads. New sandbox types can be integrated effortlessly as long as a virtualization or passthrough to the RDMA NIC is provided.

IV. rFaaS in Detail

rFaaS functions are deployed as containers (Sec. IV-A). We improve over the HTTP-based REST interfaces in other FaaS platforms, and provide a user–oriented C++ interface to improve performance and hide RDMA complexity (Sec. IV-B).

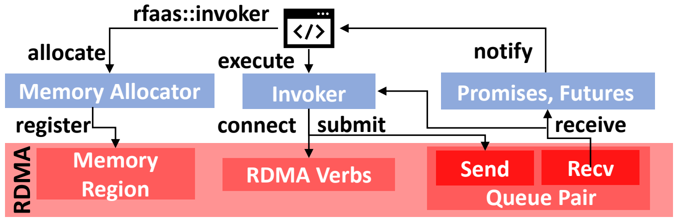


Fig. 7: **The programming model of *rFaaS*, inspired by C++ standarization efforts on the *executor* concept.**

Listing 2 Example of an rFaaS-accelerated application.

```
void compute(int size, options & opts) {
  rfaas::invoker invoker{opts.rnic_device};
① invoker.allocate(opts.lib, opts.size * sizeof(double),
    rfaas::invoker::ALWAYS_WARM_INVOCATIONS);
② auto alloc = invoker.allocator<double>{};
  // Automatically expanded with function's header
③ rfaas::buffer<double> in = alloc.input(2 * size);
  rfaas::buffer<double> out = alloc.output(2 * size);
  // Offload part of the computation to rFaaS
④ auto f = invoker.submit("task", in, size, out);
  local_task(in.data() + size, out.data() + size, size);
⑤ f.get();
⑥ invoker.deallocate(); // Release computing resources.
}
```

A. Function Deployment

Listing 1 presents the standard function interface in *rFaaS*. Input is written to memory buffers of the user code executor. 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.

rFaaS supports the execution of arbitrary functions, and similarly to the *function app* offered by Azure Functions, we enable the execution of different functions in the same worker process. Serverless functions are deployed as containers with code and all dependencies. The image is enriched with *rFaaS* RDMA executor and placed in a Docker registry.

B. Programming Model

To design the programming interface for *rFaaS*, we take inspiration from recent developments in the C++ standard for parallel and asynchronous *executors* [48]. The prior work on executors and their implementations proved that this concept is an efficient interface for dispatching tasks to accelerator devices [49, 50]. The programming model presented in Fig. 7 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 Allocator The *memory allocator* (②) provides RDMA-enabled memory buffers and encapsulates the memory region reserved for the function header (③). The allocator can be integrated effortlessly to serialize standard C++ containers such as *std::vector* and *std::array*, and all memory buffers are page-aligned to achieve the highest bandwidth on RDMA [51].

Invoker The client's *invoker* submits remote function invocations (④). It manages RDMA connections and implements the allocation and deallocation of leases. The status of the computation can be queried with busy polling to minimize latency, and we use the `std::future` to represent the result of unfinished executions. Users can query the status of each invocation, wait for its completion, and access the result later (⑤). Internally, the library runs a single thread that waits for RDMA completion events and modifies future's status when the corresponding invocation finishes. While blocking wait has a higher latency than active polling [52], the background thread sleeps, helping to reduce CPU consumption.

The allocation of *rFaaS* functions can be performed ahead of time (①) to hide the cold invocation latencies since warm executor threads are sleeping and not incurring major charges. Remote resources are allocated and deallocated as needed (⑥), adjusting to the varying parallelism and workload.

C. Billing

rFaaS uses a pricing model similar to provisioned serverless functions to include active *hot* polling. The billing model includes three cost components: allocation time C_a, hot polling C_h, and active computation time C_c.

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

The total allocation t_a measured 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 were busy with executing functions and polling for new invocations, respectively. Thus, cluster operators can encourage warm invocations to boost utilization through resource overallocation, while applications requiring the highest performance pay the premium for nanosecond invocation overheads. The billing procedure is implemented in a global database associated with the resource manager using RDMA atomic *fetch-and-add* operations, providing lightweight allocators with an RDMA-native way of accumulating cost results.

V. rFaaS in Practice

To demonstrate the fitness of *rFaaS* for HPC, we answer critical questions in the form of extensive evaluation.

1) Is *rFaaS* fast enough for high performance, latency-sensitive applications?
2) Are the overheads for initialization prohibitively large?
3) Does the *rFaaS* bandwidth scale with larger payloads?
4) Does *rFaaS* scale with more parallel workers?
5) Does *rFaaS* integrate functions into HPC applications?
6) Is the performance of remote computing with *rFaaS* competitive compared to local computation?
7) Are short *rFaaS* functions usable in HPC computations?

Platform We deploy *rFaaS* in a cluster and execute benchmark code on 4 nodes, each with two 18-core Intel Xeon Gold 6154 CPU @ 3.00GHz and 377 GB of memory. The nodes are equipped with a Mellanox MT27800 Family NIC with a 100 Gb/s Single-Port link 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 Mellanox's SR-IOV plugin to run containers over virtual device functions. *rFaaS* is implemented in C++, using g++ 8.3.1.

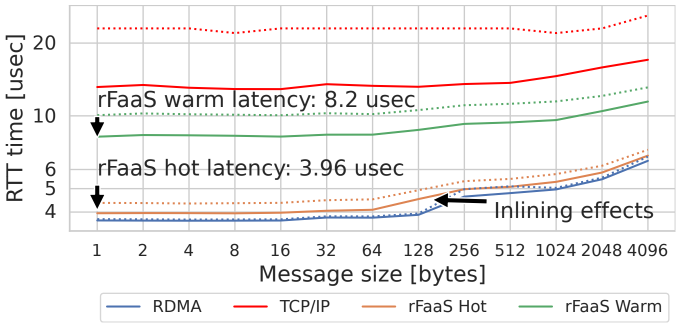


Fig. 8: **The RTT of an no-op *rFaaS* function and network transport, median (solid) and 99th latency (dashed).**

A. Invocation Latency

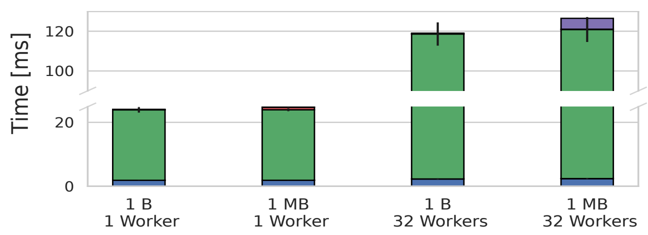
We begin by measuring the hot and warm invocation latency, using a no-op "echo" function that returns the provided input. We use a warmed-up, single-threaded, bare-metal executor with the main thread pinned to a CPU core, perform 10,000 repetitions, and report the median. We compute the non-parametric 99% confidence intervals of the median 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 report the median of the *ib_write_lat* benchmark executed with thread pinning. For the latter, we report the mean of *netperf* used with page-aligned buffers and process pinning.

Fig. 8 shows that the overhead imposed by processing a no-op function by *rFaaS* in a process is 326 ns on average, compared to the baseline RDMA data transmission. 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 transmission to improve performance of small messages [36]. However, the communication in *rFaaS* is asymmetric: we transmit 12 more bytes for the input, forcing us to use non-inlined write operations in one direction. The average overhead of a warm execution is 4.67 μs, and containerization adds a latency of ca. 650 ns.

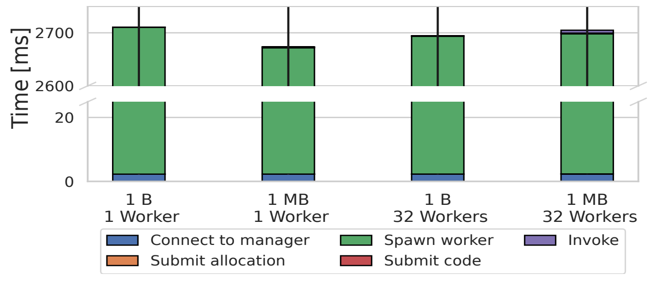
With slightly more than 300 ns of overhead, we enable remote invocations without noticeable performance penalty, conclusively answering: ***rFaaS* is fast enough for latency-sensitive, high-performance applications**.

B. Cold Invocation Overheads

Fig. 9a and 9b 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



(a) Bare-metal Linux process



(b) Docker container

Fig. 9: **Cold invocations of *rFaaS* functions.**

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. Therefore, we can claim that *rFaaS* does not introduce significant overheads in addition to sandbox initialization.

While the current version of Docker with SR-IOV shows an overhead of approximately 2.7 seconds to spawn workers, low-latency approaches can reduce this time to as little as 125 milliseconds [30]. Thus, it is compatible with proposed approaches to reduce cold startup latencies, such as container warming and reinitialization. Finally, the user's function can be deployed as a code package like in many other FaaS platforms, allowing executor managers to keep a pool of generic and ready containers and bypass the container startup latency.

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

C. Bandwidth scalability

To compare the performance of *rFaaS* and other platforms, we evaluate a non-op C++ function that returns the input provided in 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.

We compare with AWS Lambda, a state-of-the-art commercial FaaS solution, as neither Azure Functions nor Google Cloud Functions support C++ functions. Then, we compare against open-source FaaS platforms OpenWhisk and Nightcore [14], a low-latency and open-source serverless platform. Both open-source systems are deployed in the same RDMA-capable cluster as *rFaaS*, using the same network and CPU resources as our system. In Lambda, we deploy a native function implemented with C++ Runtime, 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 our cluster a standalone OpenWhisk using Docker with Kafka and API gateway. A C++ function in OpenWhisk is invoked as a regular application, accepting inputs not greater than 125 kB through *argc* and *argv*. Similarly, we deploy the function in a *nightcore* instance in the cluster.

We present the evaluation result in Fig. 1 (page 1). On all payload sizes, *rFaaS* clearly provides significantly better performance. *rFaaS* invocations are between 695x and 3,692x faster than AWS Lambda executions. Stable measurements on the no-op function indicate that the difference is not caused by shared CPU resources of the cloud, but by the low-latency network and native support for transmitting raw data. *rFaaS* is between 23x and 39x faster than Nightcore when running on the same hardware. Similarly, *rFaaS* provides a speedup between 5,904x and 22,406x when compared to OpenWhisk.

rFaaS provides significant performance improvements over current FaaS platforms and **scales well with message size**.

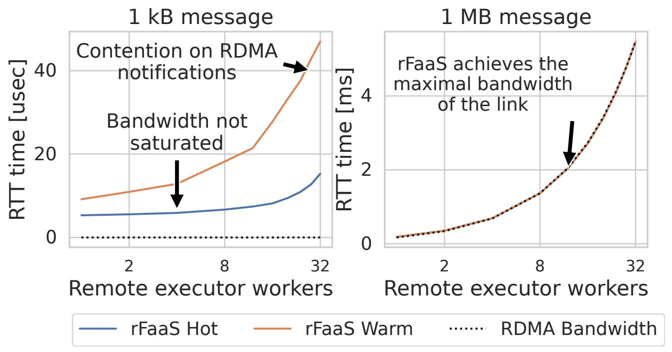


Fig. 10: ***rFaaS* invocations on parallel executors.**

D. Parallel scalability

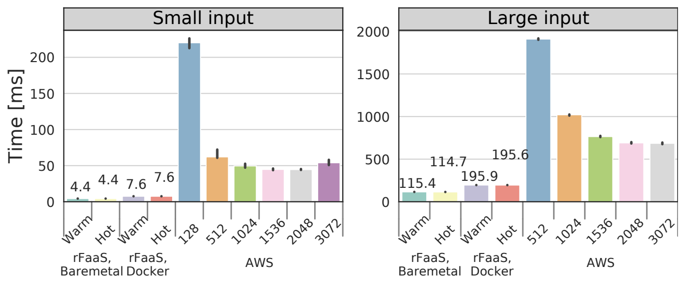
To verify that RDMA-capable functions scale efficiently to handle integration into scalable applications, we place managers on 36-core CPUs and evaluate parallel invocations. We execute the no-op function on warmed-up, bare-metal executors having allocated from 1 to 32 worker threads.

Fig. 10 presents the round-trip latencies for invoking functions with 1 kB and 1MB payloads, respectively. The overhead of handling many concurrent connections is insignificant on hot invocations with a smaller payload. While the Docker executor shows performance increases (hot) and decreases (warm) on the 1 kB payload, the difference on 1MB 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**.

E. Use-case: serverless functions

To evaluate the effectiveness of *rFaaS* in integrating serverless functions into high-performance applications, we select real-world serverless functions from the SeBS benchmark [21]. We take the *thumbnailer* benchmark as an example of general-purpose image processing and the *image-recognition* benchmark performing ResNet-50 prediction as an example of


(a) Image processing: thumbnail generation: *AWS Lambda* and rFaaS.

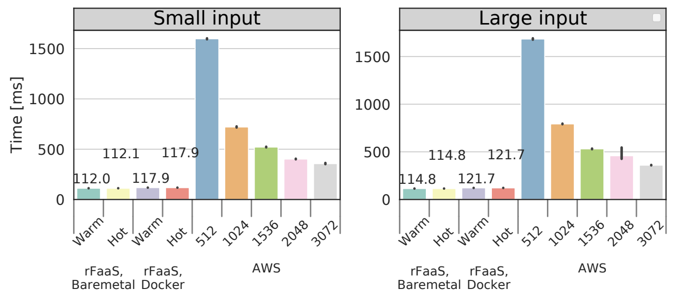

(b) Image recognition with ResNet-50 and PyTorch: *AWS Lambda*, rFaaS.

Fig. 11: ***rFaaS* on serverless functions.**

integrating deep-learning inference into applications. We reimplement the Python benchmarks in C++ and deploy them as Docker images on *rFaaS* and AWS Lambda. We repeat each benchmark 100 times.

a) Image processing: We implement the thumbnail generation with OpenCV 4.5. We evaluate functions with two images, 97 kB *small* one and a 3.6 MB *large* one (Fig. 11a). For the AWS Lambda function, we need to submit the binary image data as a base64-encoded string in the POST request, which adds the overhead of encoding and conversions. On the other hand, *rFaaS* functions benefit from the payload format not constrained by cloud API requirements.

b) Image registration: We implement the benchmark with the help of PyTorch C++ API, using *OpenCV* 4.5, *libtorch* 1.9, and *torchvision* 0.1. We convert the Python serialized model included with *SeBS* into the recommended TorchScript model format. The model is included with the Docker image and stored in the function memory after the first invocation. We evaluate functions with two inputs, a 53 kB *small* image and a 230 kB *large* one (Fig. 11b). There is a growing interest in using machine-learning inference to speed up computations and simulations [53–55]. The results demonstrate that ***rFaaS* functions can efficiently implement inference tasks in high-performance applications**.

F. Use-case: parallel offloading

We want to answer the next question: can *rFaaS* offload computations efficiently to remote serverless workers? We study offloading of massively parallel computations with significant data movement. We select the Black-Scholes solver [56] from the PARSEC suite [57] parallelized with OpenMP threading. Black-Scholes solves the same partial differential equation for different parameters, and we dispatch

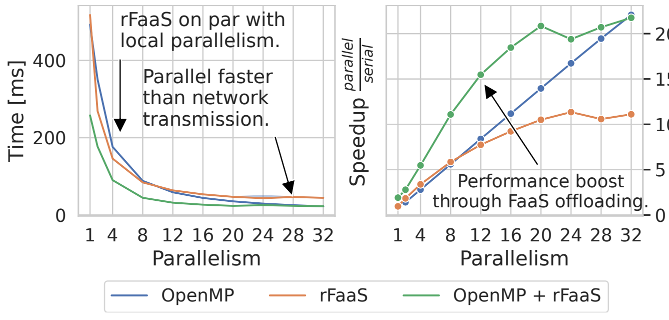
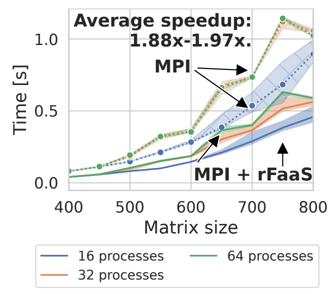
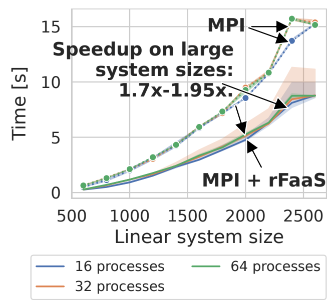


Fig. 12: **Parallel serverless computing with *rFaaS* and *OpenMP*. Medians with non-parametric 95% CIs.**



(a) Matrix-matrix multiplication.



(b) Jacobi method, 100 iterations.

Fig. 13: **MPI (solid) versus MPI + *rFaaS* (dashed), reported medians with non-parametric 95% CIs.**

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 Fig. 12.

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 up the OpenMP application 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. V-C), 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.**

G. Use-case: HPC Applications

The next question we want to answer is: how much performance can be gained by offloading complex tasks to the cheap and spare capacity of HPC clusters?

a) Matrix-matrix multiplication: We run an *MPI* application where each rank performs a matrix-matrix multiplication, averages it over 100 repetitions, and we measure the median kernel time across MPI ranks. MPI ranks are distributed across two 36-core nodes, and we pin each rank to a single core. Then, we deploy an *MPI + rFaaS* application where each rank allocates a single bare-metal *rFaaS* function. *rFaaS* executors are deployed on two other 36-core nodes, and with such concentration of MPI and *rFaaS* computing resources, we show that sharing the network bandwidth does not prevent efficient serverless acceleration. Due to a high computation to communication ratio, we split the workload equally, and both MPI rank and the function compute half of the result matrix. Figure 13a shows *rFaaS* provides a speedup between 1.88x and 1.94x depending on the number of MPI processes. Functions with a good ratio of computation to unique memory access can be accelerated with *rFaaS*. As long as this condition is satisfied, ***rFaaS* improves the performance of HPC workloads**.

b) Jacobi linear solver: To show a serverless acceleration of a bulk synchronous type of problem, we consider the Jacobi linear solver, where half of each iteration is offloaded to *rFaaS*. Here, we perform a classical serverless optimization of caching resources in a warmed-up sandbox. Since the matrix and right-hand vector do not change between iterations, we submit them only for the first invocation. As long as the allocated function is not removed, we send only an updated solution vector in subsequent iterations.

We evaluate the approach in the same setting as matrix multiplication (Section V-G0a), with MPI ranks averaging the Jacobi method with 1000 iterations over ten repetitions, and measure a speedup between 1.7 and 2.2 when *rFaaS* acceleration is used. Since each iteration takes just between 1 and 15 milliseconds, the results must be returned with minimal overhead to offer performance comparable with the main MPI process. **the low-latency invocations in *rFaaS* apply to millisecond-scale computations.**

VI. Related Work

FuncX [58] is a federated platform that brings 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 [14] is a high-performance FaaS runtime designed for microservices, with optimized internal function calls — invocations that can be satisfied locally, without inter-node communication. SAND [17] optimizes workflows of serverless functions with grouping of functions and dedicated message buses. In contrast, *rFaaS* exploits co-location via explicit parallelism of executor allocation and optimizes invocation latencies through RDMA communication. Archipelago [16] and Wukong [15] perform latency-aware scheduling of directed acyclic graph (DAG) of functions. 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. SmartNICs have been shown to provide fast dispatching and orchestration in FaaS [59, 60]. However, functions are limited by restricted implementation language and low-performance RISC cores. Other improvements and optimization strategies, such as warming, provisioning and fast startup solutions for sandboxes [61, 62], are orthogonal to *rFaaS* and can be implemented in our platform as well. RDMA has been used in the serverless context for resource disaggregation [63] and heterogeneous systems [64]; *rFaaS* optimizes FaaS architecture and is tailored for HPC computing.

Remote Invocations Remote Procedure Calls (RPC) [65] and Active Messages [66] invoke a procedure remotely on another machine. Active Networks include capsules with user code that can be executed on selected routers [67]. 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* does not send code with invocation, we provide a protection boundary between caller and callee needed to access private resources, e.g., in ML inference serving.

VII. 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 emerging solutions in FaaS.

Which workloads will benefit from *rFaaS*? High-performance and parallel applications need scalable invocations of remote workers (Sec. V-D) and high network bandwidth to support simultaneous invocations by parallel processes on the same node (Sec. V-C). Furthermore, data-intensive workloads will benefit from RDMA-accelerated FaaS computing since other platforms cannot achieve high throughput on networks that support the transmission of gigabytes of data per second. Examples include, but are not limited to, machine-learning inference, data analytics, GPU-accelerated functions with short computation time, and task-based applications with no memory sharing between tasks [68].

On the other hand, HPC applications that will likely not benefit from *rFaaS* offloading include memory-bound operations with a low ratio of computation to accessed data. Furthermore, applications that already achieve high resource utilization have little motivation to look into serverless in the first place, e.g., applications with static parallelism and homogenous resource requirements.

Can *rFaaS* improve serverless workflows? In workflows, functions are composed to build serverless applications, using a coordination service to orchestrate invocations and data propagation [69]. While SmartNICs offer fast orchestration [59], they are limited by the cost and availability of dedicated NICs. Instead, implementing orchestrator with *rFaaS* executors achieves two performance goals: single-digit microsecond latency overhead of invocations and efficient data movement.

Can *rFaaS* support the diverse world of HPC systems? Thanks to its modular design, *rFaaS* supports extensions into new environments and hardware. *rFaaS* functionality and RDMA abstractions are orthogonal to the device interface, and network management with `ibverbs` can be extended with new network drivers and software virtualization for RDMA [38]. *rFaaS* supports native integration into C/C++

applications (Sec. IV), but the language choice is independent of the platform itself. Functions and integration can be implemented through ABI compatibility and foreign-function interfaces, supporting languages such as Fortran and Python. *rFaaS* functions can be served in containers other than Docker, e.g., in HPC container Singularity, as long as they provide access to the RDMA NIC.

VIII. Conclusions

Fine-grained and granular computing need systems designed to handle microsecond-scale workloads [23, 70], 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, providing serverless programmability in high-performance systems and applications, and paving the way for future low-latency and fine-grained computing.

Acknowledgment

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement EPIGRAM-HS, No. 801039, and grant agreement RED-SEA, No. 955776), and from the Schweizerische Nationalfonds zur Förderung der wissenschaftlichen Forschung (SNF, Swiss National Science Foundation) through Project 170415. We would also like to thank the Swiss National Supercomputing Centre (CSCS) for providing us with access to their supercomputing machines Daint and Ault. We also thank anonymous reviewers for helping us improve the manuscript.

References

[1] P. Prabhu *et al.*, "A survey of the practice of computational science," in *SC '11*.
[2] D. G. Feitelson *et al.*, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, 1996.
[3] A. Raveendran *et al.*, "A framework for elastic execution of existing mpi programs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*.
[4] A. Verma *et al.*, "Large-scale cluster management at google with borg," in *EuroSys '15*.
[5] Y. Zhai *et al.*, "Cloud versus in-house cluster: Evaluating amazon cluster compute instances for running mpi applications," in *SC*, 2011.
[6] M. A. S. Netto *et al.*, "Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges," *ACM Comput. Surv.*, vol. 51, 2018.
[7] J. Zhang *et al.*, "High performance mpi library for container-based hpc cloud on infiniband clusters," in *ICPP*, 2016.
[8] W. Huang *et al.*, "A case for high performance computing with virtual machines," in *ICS '06*.
[9] C. Ruiz *et al.*, "Performance evaluation of containers for hpc," in *Euro-Par 2015: Parallel Processing Workshops*.
[10] J. Hao *et al.*, "An empirical analysis of vm startup times in public iaas clouds," in *CLOUD*, 2021.
[11] J. R. Gunasekaran *et al.*, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *CLOUD*, 2019.
[12] M. AbdelBaky *et al.*, "Enabling high-performance computing as a service," *Computer*, vol. 45, 2012.
[13] C. Guo *et al.*, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*.
[14] Z. Jia *et al.*, "Nightcore: Efficient and scalable serverless computing forlatency-sensitive, interactive microservices," in *ASPLOS '21*, 2021.
[15] B. Carver *et al.*, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *SoCC '20*.
[16] A. Singhvi *et al.*, "Archipelago: A scalable low-latency serverless platform," *arXiv:1911.09849*, 2019.
[17] I. E. Akkus *et al.*, "Sand: Towards high-performance serverless computing," in *ATC '18*.
[18] V. Sreekanti *et al.*, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, 2020.
[19] M. Wawrzoniak *et al.*, "Boxer: Data analytics on network-enabled serverless platforms," in *CIDR'21*.
[20] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv:1902.03383*, 2019.
[21] M. Copik *et al.*, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Middleware '21*.
[22] J. Jiang *et al.*, "Towards demystifying serverless machine learning training," in *SIGMOD 2021*.
[23] P. G. Lopez *et al.*, "Serverless predictions: 2021-2030," *arXiv:2104.03075*, 2021.
[24] L. A. Barroso *et al.*, "The case for energy-proportional computing," *Computer*, vol. 40, 2007.
[25] H. You *et al.*, "Comprehensive workload analysis and modeling of a petascale supercomputer," in *Job Scheduling Strategies for Parallel Processing*, 2013.
[26] T. Patel *et al.*, "Job characteristics on large-scale systems: Long-term analysis, quantification, and implications," in *SC '20*.
[27] M. D. Jones *et al.*, "Workload analysis of blue waters," *arXiv:1703.00924*, 2017.
[28] G. Panwar *et al.*, "Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support," in *MICRO '52*, 2019.
[29] L. Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *ATC '18*.
[30] A. Agache *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *NSDI 20*, 2020.
[31] M. Sciabarrà, *Learning Apache OpenWhisk: Developing Open Serverless Solutions*. O'Reilly Media, Incorporated, 2019.
[32] I. Müller *et al.*, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," *arXiv:1912.00937*, 2019.
[33] S. Fouladi *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *NSDI'17*.
[34] G. París *et al.*, "Serverless elastic exploration of unbalanced algorithms," in *CLOUD*, 2020.
[35] V. Mauch *et al.*, "High performance cloud computing," *Future Generation Computer Systems*, vol. 29, 2013.
[36] A. Kalia *et al.*, "Using rdma efficiently for key-value services," in *SIGCOMM '14*.
[37] Y. Dong *et al.*, "High performance network virtualization with sr-iov," in *HPCA*, 2010.
[38] D. Kim *et al.*, "Freeflow: Software-based virtual RDMA networking for containerized clouds," in *NSDI 19*, 2019.
[39] K. Taranov *et al.*, "sRDMA – efficient NIC-based authentication and encryption for remote direct memory access," in *ATC 20*, 2020.
[40] A. Singhvi *et al.*, "1rma: Re-envisioning remote memory access for multi-tenant datacenters," in *SIGCOMM '20*.
[41] "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.
[42] "libfabric, Open Fabric Interfaces," https://github.com/ofiwg/libfabric, 2021, accessed: 2022-01-12.
[43] "AWS Elastic Fabric Adapter," https://aws.amazon.com/hpc/efa/, 2021, accessed: 2022-01-12.
[44] "The Linux SoftRoCE Driver," https://www.youtube.com/watch?v=NumH5YeVjHU, 2017, accessed: 2022-01-12.
[45] G. M. Kurtzer *et al.*, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, 2017.
[46] E. G. Young *et al.*, "The true cost of containing: A gvisor case study," in *HotCloud 19*, 2019.
[47] "Firecracker," https://github.com/firecracker-microvm/firecracker, 2018, accessed: 2022-01-12.
[48] "P0443R14: A Unified Executors Proposal for C++," https://wg21.link/p0443r14, 09 2020, accessed: 2022-01-12.
[49] T. Heller *et al.*, "Closing the performance gap with modern c++," in *High Performance Computing*, 2016.
[50] M. Copik *et al.*, "Using sycl as an implementation framework for hpx.compute," in *IWOCL 2017*.
[51] A. Kalia *et al.*, "Design guidelines for high performance RDMA systems," in *ATC 16*, 2016.
[52] P. MacArthur *et al.*, "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*.
[53] A. Brace *et al.*, "Achieving 100x faster simulations of complex biological phenomena by coupling ml to hpc ensembles," *arXiv:2104.04797*, 2021.
[54] M. Wang *et al.*, "Gpu-accelerated machine learning inference as a service for computing in neutrino experiments," *Frontiers in Big Data*, vol. 3, 2021.
[55] J. Krupa *et al.*, "GPU coprocessors as a service for deep learning inference in high energy physics," *Machine Learning: Science and Technology*, vol. 2, 2021.
[56] A. Heinecke *et al.*, "A highly parallel black–scholes solver based on adaptive sparse grids," *Int. J. Comput. Math.*, vol. 89, 2012.
[57] C. Bienia *et al.*, "The parsec benchmark suite: Characterization and architectural implications," in *PACT '08*.
[58] R. Chard *et al.*, "Funcx: A federated function serving fabric for science," in *HPDC '20*.
[59] N. Daw *et al.*, "Speedo: Fast dispatch and orchestration of serverless workflows," in *SoCC '21*.
[60] S. Choi *et al.*, "λ-nic: Interactive serverless compute on programmable smartnics," in *ICDCS*, 2020.
[61] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *ATC 18*, 2018.
[62] D. Du *et al.*, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS '20*.
[63] Z. Guo *et al.*, "Resource-centric serverless computing," *arXiv preprint arXiv:2206.13444*, 2022.
[64] D. Du *et al.*, "Serverless computing on heterogeneous computers," in *ASPLOS '22*.
[65] B. J. Nelson, *Remote procedure call.* Carnegie Mellon University, 1981.
[66] T. v. Eicken *et al.*, "Active messages: A mechanism for integrated communication and computation," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*.
[67] D. Wetherall *et al.*, "Active network vision and reality: Lessions from a capsule-based system," in *SOSP '99*.
[68] M. Copik, T. Grosser, T. Hoefler, P. Bientinesi, and B. Berkels, "Work-stealing prefix scan: Addressing load imbalance in large-scale image registration," 2020.

[69] S. Burckhardt *et al.*, “Durable functions: Semantics for stateful serverless,” *Proc. ACM Program. Lang.*, vol. 5, 2021.

[70] C. Lee *et al.*, “Granular computing,” in *HotOS ’19*.