Serverless Computing on Heterogeneous Computers

Dong Du
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai, China
Dd_nirvana@sjtu.edu.cn

Qingyuan Liu
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai, China
liu_qy@sjtu.edu.cn

Xueqiang Jiang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai, China
jiangxq@sjtu.edu.cn

Yubin Xia
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai Artificial Intelligence Laboratory
Shanghai, China
xiayubin@sjtu.edu.cn

Binyu Zang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai, China
byzang@sjtu.edu.cn

Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai, China
haibochen@sjtu.edu.cn

ABSTRACT

Existing serverless computing platforms are built upon homogeneous computers, limiting the function density and restricting serverless computing to limited scenarios. We introduce Molecule, the first serverless computing system utilizing heterogeneous computers. Molecule enables both general-purpose devices (e.g., Nvidia DPU) and domain-specific accelerators (e.g., FPGA and GPU) for serverless applications that significantly improve function density (50% higher) and application performance (up to 34.6x). To achieve these results, we first propose XPU-Shim, a distributed shim to bridge the gap between underlying multi-OS systems (when using general-purpose devices) and our serverless runtime (i.e., Molecule). We further introduce vectorized sandbox, a sandbox abstraction to abstract hardware heterogeneity (when using domain-specific accelerators). Moreover, we also review state-of-the-art serverless optimizations on startup and communication latency and overcome the challenges to implement them on heterogeneous computers. We have implemented Molecule on real platforms with Nvidia DPUs and Xilinx FPGAs and evaluate it using benchmarks and real-world applications.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Operating systems**.

KEYWORDS

Cloud computing, serverless computing, heterogeneous computers, function-as-a-service, operating system

ACM Reference Format:
Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3503222.3507732

1 INTRODUCTION

Serverless computing [62] has become an emerging paradigm of today's cloud and data center infrastructures [6, 9, 12, 21]. It uses one single-purpose service or function as the basic computation unit, which eases computing in several ways. First, it helps application developers focus on the core logic, and leaves infrastructure-related tasks like auto-scaling to the serverless platform. Second, it adopts the "pay-as-you-go" model with fine-grained charging granularity (e.g., 1ms [7]) so that users can save costs for unused computing resources. Third, serverless computing also benefits cloud providers such that they can manage their resources more efficiently.

There are already many implementations [4, 9, 18, 42] and optimizations [11, 16, 35, 43, 54, 61, 78, 84, 91, 92] of serverless systems. However, (almost) all of these works focus on serverless computing with a homogeneous computer, one that contains processing units (to run functions) with the same computation ability (typically same ISA and frequency). This homogeneous architecture is meeting its limitations in the face of several issues and recent trends in serverless platforms.

Challenges of serverless. First, serverless auto-scalability and the need for low communication latency require a serverless runtime to support high function density in a single machine. However, it is hard for homogeneous computers as the prospect of dark silicon [55] hampers general-purpose parallelism in computers that only have CPUs as the processing units. Second, nowadays, many important applications, e.g., machine learning, artificial intelligence, video classification, and genome analysis, rely on heterogeneous accelerators (e.g., FPGA, GPU, and ASIC) for faster computation. For example, genome sequencing analysis can achieve over ten times speedup utilizing AWS F1 FPGA [24]. Inability to leverage these heterogeneous accelerators restricts serverless computing to be only used in limited scenarios. Moreover, co-locating I/O stacks with computations in CPU can lead to worse resource utilization [71] and break performance isolation [64].

We believe that future serverless platforms will be deployed on *heterogeneous computers*, which are connected with CPU, DPU (data processing units [10, 31]), domain-specific accelerators [2, 67, 86], and smart devices like smartNIC and smartSSD. Heterogeneous computers greatly improve scalability (i.e., vertical scaling), performance for a wider range of applications, and resource isolation. With these benefits, heterogeneous computers are rapidly permeating data centers in recent years [19, 20, 23, 25, 40, 56].

This paper presents Molecule, the first serverless computing system on heterogeneous computers. Molecule takes both general-purpose devices (e.g., Nvidia DPU) and domain-specific accelerators (e.g., FPGA) into account. Molecule leverages DPU for better function density and FPGA for better application performance but can still provide high-level and easy-to-use programming models, which are beneficial to developers for two reasons. First, developers can easily utilize heterogeneous processing units to write their serverless applications. Second, Molecule retains all the benefits promised by serverless computing, e.g., auto-scalability.

Designing a serverless system on heterogeneous computers raises three major challenges. First, as the DPU and accelerators themselves become more complex [75, 76], it is not surprising that more and more of these devices are deploying an OS to manage the hardware resources, e.g., OSes on DPU and network devices [27, 75], RTOS on SSD controller [68]. This trend makes heterogeneous computers *multi-OS* systems. However, existing serverless systems rely on *single* OS to manage the lifecycle of functions, control resource allocation, and enforce permission controls, e.g., Catalyzer [54] leverages OS fork to create new instances. This brings a new challenge to serverless computing: how can we achieve the same functionalities with multi-OS systems?

Second, a design to abstract away low-level heterogeneous hardware and software details for serverless systems is desired but still missing. In heterogeneous computers, different processing units (PUs) can have different ISAs (e.g., X86 and ARM), frequencies, cache sizes, I/O bandwidth (better for smart devices), and other specific characteristics. Many of these hardware details need to be exposed to applications for better performance (e.g., RDMA connection between DPU and CPU). However, this violates the principle of serverless computing, which aims to provide a high-level and hardware-agnostic view for applications. The software stack on heterogeneous PUs does not help in this case — PUs may deploy different OSes (or a simple firmware for devices/accelerators) with different services; that further increases the heterogeneity.

Third, heterogeneous computers also complicate communication between serverless functions. Most serverless applications adopt *function chain* [82], which leads to intensive inter-function communication. Existing serverless systems rely on OS primitives, e.g., IPC [43, 61], to reduce the communication latency when functions are running on the same PU. However, they still use the network for communication between two PUs and thus miss great optimization opportunities due to the wrong assumption of the underlying

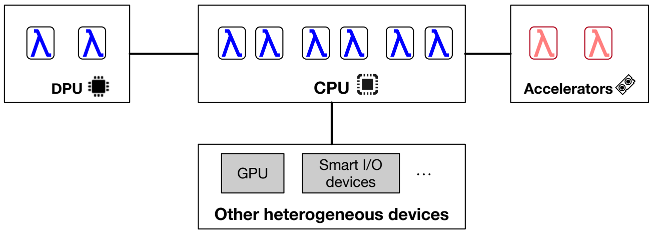


Figure 1: Serverless on heterogeneous computers.

hardware and the inability to communicate with PUs on the same machine directly (e.g., DMA, RDMA, or Intel CXL [13]).

To tackle these challenges, we first propose a generic serverless abstraction, *vectorized sandbox*, to overcome the challenges of heterogeneity. The vectorized sandbox extends existing abstraction to support concurrent sandbox creation and invocation, which is the key to enable serverless functions on domain-specific accelerators like FPGA. As we require each device (or PU) to implement the interfaces required by the abstraction, a serverless runtime can manage heterogeneous functions without considering the underlying hardware and software details. Based on this abstraction, we have implemented serverless sandboxes on CPU, DPU, and FPGA.

Second, this paper presents XPU-Shim, which is an indirection layer between a *single* serverless runtime and *multiple* OSes. XPU-Shim has two key primitives, *neighbor IPC* and *distributed capabilities*. Neighbor IPC utilizes hardware interconnects, e.g., DMA, to allow applications on different PUs to communicate efficiently, while distributed capabilities provide a unified way to enforce permission control in a multi-OS system. XPU-Shim bridges the gap between existing serverless mechanisms and the multi-OS system.

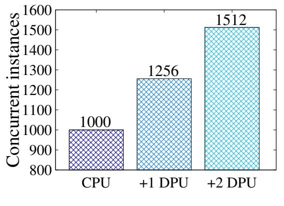
Furthermore, we successfully build Molecule, the serverless system for heterogeneous computers. Molecule is built upon XPU-Shim and vectorized sandbox. We also review state-of-the-art serverless optimizations on startup and communication latency and overcome the challenges of implementing them on heterogeneous computers. We have implemented Molecule on real platforms with Nvidia DPU and Xilinx UltraScale+ FPGA and evaluated it using benchmarks [66, 93] and real-world applications. The results show that Molecule can achieve up to 50% higher function density with 2 DPU devices, and significantly improve application performance by supporting FPGA serverless functions (up to 34.6x better). Molecule also overcomes many challenges to achieve low startup and communication latency, i.e., 10x less startup latency and 13x less communication latency than baseline systems.

2 MOTIVATION

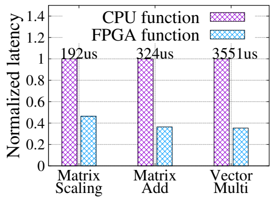
2.1 Background

2.1.1 Heterogeneous Computers. Nowadays, datacenter and cloud tend to deploy heterogeneous devices, which are connected with CPU, DPU (data processing units [10, 31]), domain-specific accelerators (e.g., ASIC, FPGA, GPU) [2, 67, 86], and smart I/O devices with embedded computing capability like smartSSD and smartNIC [28–31, 37]. The architecture with connected heterogeneous devices is coined *heterogeneous computer*.

Heterogeneous computers greatly improve scalability (i.e., vertical scaling), performance for a wider range of applications, and



(a) DPU for higher density.



(b) FPGA for better performance.

Figure 2: Benefits of serverless on heterogeneous computers. ***(a) We use the Python image processing function from ServerlessBench as the application and evaluate on an x86 server machine with two DPUs. (b) We use three functions implementing matrix operations (i.e., scaling, addition, and multiplication) and evaluate on EC2 F1 instances (x86 server with one FPGA device).***

resource utilization. With these benefits, we believe that future serverless platforms will be deployed on heterogeneous computers.

Multi-OS architecture. As the heterogeneous devices themselves become more complex, device vendors tend to deploy an OS to manage the hardware resources on the device [27, 68, 75]. For example, Nvidia DPU leverages Linux to manage the NIC devices, accelerators (e.g., crypto engine), onboard memory and persistent storage. This approach decouples the resource management in the device and the host OS (on CPU), which is more flexible than using a single OS to manage all. The architecture with multiple OSes on a single computer is coined *multi-OS architecture*.

In our experimental server, there are three Linux systems (i.e., Ubuntu 20.04) with two Bluefield DPUs: one on the CPU and two on the DPUs. It is still unknown how to implement serverless functionalities with the multi-OS architecture.

2.1.2 Serverless Platform and Sandbox. Serverless computing [62] uses one single-purpose *function* as the basic computation unit, which eases computing in several ways, e.g., auto-scaling and pay-as-you-go model. Usually, application developers send their functions to a *serverless platform*, which compiles the functions offline together with a *language runtime*. When requests arrive, the platform will schedule a worker machine with a *serverless runtime* to provide sandboxed execution environments to invoke the function.

The sandboxed execution environment is called *serverless sandbox* in the paper, which could be container [15, 78], gVisor [22], AWS Firecracker [17], or Kata Container [26]. Although the isolation mechanisms and guarantees are different for serverless sandboxes, they usually implement the same set of runtime interfaces, e.g., OCI[33]. With the standard interfaces, a serverless platform does not need to care about a sandbox's internal design and can only use the interfaces to manage function instances.

Next, we give use cases leveraging heterogeneous computers to overcome the challenges of existing serverless computing (e.g., low function density and limited scenarios).

2.2 Case-1: DPU for Higher Density

DPUs (data processing units) [27, 75, 76] are a new class of programmable devices that move and process data around the data centers. DPU is a system on a chip (or SoC). It is capable of running commercial OSes and performance-critical applications like serverless functions. Usually, it includes a high-performant multi-core processing unit (e.g., 2.75GHz ARM cores in Bluefield-2 DPU), an efficient network interface capable of processing and transferring data at (almost) line rate, and optional domain-specific accelerators. Therefore, a serverless runtime that can distribute functions to CPU and DPU can effectively improve the function density, as shown in the left of Figure 1.

However, designing such a system is challenging because the OS on DPU makes the CPU-DPU heterogeneous computers multi-OS systems while existing serverless mechanisms are built upon the single-OS system. In this paper, Molecule relies on a distributed shim (i.e., XPU-Shim) to overcome the challenge. As shown in Figure 2-a, Molecule can achieve 50% better scalability with 2 Bluefield DPUs in a single computer.

2.3 Case-2: Accelerator for Better Performance

Applications like machine learning and big data analytics are widely spread in the cloud and datacenter, and rely on domain-specific accelerators to finish computation tasks. For instance, Amazon EC2 F1 [2] allows developers to design and deploy accelerators on the cloud to boost applications like genome sequencing, big data analytics, and video processing. As shown in the right of Figure 1, building serverless platforms on heterogeneous computers with accelerators can enable more application scenarios in serverless computing. Figure 2-b shows that Molecule's serverless functions can achieve 3x better latency for a matrix computation chain (used in ML applications) by utilizing FPGA. Besides, the combination may help to improve the utilization of accelerators (ignoring fine-grained scheduler overheads), which is a serious problem in data centers now [60, 90]. With serverless, we can flexibly schedule tasks to accelerators in a fine-grained way.

Although bringing accelerators to serverless has many benefits, abstracting hardware heterogeneity into high-level interfaces for serverless computing is hard. In this paper, Molecule proposes a generic serverless abstraction, *vectorized sandbox*, to overcome the challenges of heterogeneity.

2.4 Other Representative Cases

Many other workloads [87, 89] require a combination of hardware acceleration and low execution latency and are invoked many times, which can benefit from the heterogeneous serverless computing. For example, Dorylus [89], a serverless application for GNN training, can only use CPU now, which can be improved by using accelerators like GPU with the help of Molecule.

2.5 Goals

The paper aims to propose a general and consistent set of abstractions for heterogeneous serverless computing. Based on the abstractions, we have built the Molecule serverless system, which supports CPU, DPU, and FPGA, and can easily extend to other PUs and devices. The paper further proposes a set of optimizations for heterogeneous serverless, e.g., optimizing startup and communication latencies. These contributions are shown in Table 1. Next, we will introduce the abstractions in §3, and elaborate the Molecule serverless design in §4.

Table 1: Overall contributions. *V.S. is short for vectorized sandbox, and CPU-inter. is short for CPU-intercepted. ✓ means an abstraction or an optimization is supported in the target PU.*

PUs	Abstractions		Optimizations			Communication methods		
	V.S.	XPU-Shim	cFork	V.S. caching	nIPC DAG	to CPU	to DPU	to FPGA
CPU	✓	✓	✓		✓	IPC	RDMA	DMA
DPU	✓	✓	✓		✓	RDMA	IPC	CPU-inter.
FPGA	✓	✓		✓	✓	DMA	CPU-inter.	Shm.

3 ABSTRACTION

This section introduces two key abstractions for heterogeneous serverless computing, XPU-Shim (§3.1–§3.4) and vectorized sandbox (§3.5), which abstract away the hardware distribution and heterogeneity for a serverless platform.

3.1 XPU-Shim

Shim is a common software mechanism for application adaptability in computing. A shim is usually a library or a middleware that intercepts API calls and changes the arguments passed, handles the operation itself or redirects the operation elsewhere [34]. Following the principle, this paper proposes a distributed shim, *XPU-Shim*, to handle the distribution caused by heterogeneous computers (i.e., multi-OS).

XPU-Shim is an indirection layer supporting serverless runtime on the multi-OS environment. The overall architecture is shown in Figure 3. We describe three key features of XPU-Shim. First, XPU-Shim is working upon the OS or firmware at each PU (called *local OS*) and utilizes interfaces of vectorized sandbox to manage heterogeneous functions — this significantly improves the system flexibility as the local OS and PU could be very different. Second, XPU-Shim provides system call style interfaces to applications, called *XPUcalls* (shown in Table 2), that provide a unified abstraction to manage and utilize resources on different PUs. XPU-Shim is distributed at each PU with generic programming ability (e.g., x86 or ARM cores) and maintains the global states of the heterogeneous computer and forms the same view for applications on different PUs. It usually requires XPU-Shim to maintain the consistency of global states through some protocols [45]. Last, XPU-Shim supports efficient communication for applications even on different PUs.

XPU-Shim relies on two key primitives, *distributed capabilities* and *neighbor IPC*, to handle distributed hardware and OSes.

3.2 Distributed Capability

XPU-Shim maintains global resources and states for user-space applications using distributed objects and capabilities. Currently, we have defined the following two distributed objects: (1) **CAP_Group** is the object recording all capabilities of a process, and (2) **IPC** is the inter-process connection object. This is much simpler than traditional capability systems because XPU-Shim only utilizes the capability system to manage *permission* and *communication* among different PUs.

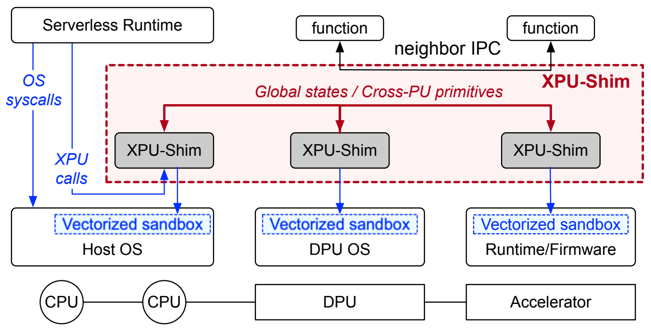


Figure 3: Abstraction for heterogeneous serverless.

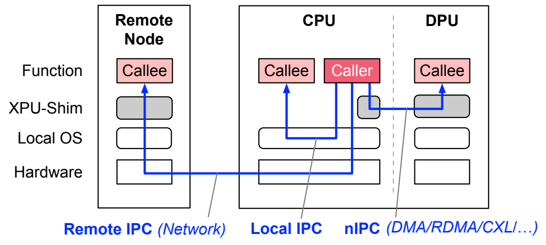


Figure 4: Three IPC methods. *XPU-Shim introduces* `nIPC` *as a complement to IPC and remote IPC (network).*

Global process. To utilize XPU-Shim's unified abstraction, a process[1] should be globally identifiable. This is not naturally supported, e.g., Linux PID is unique in the local PU, but can not guarantee global uniqueness. XPU-Shim maintains a CAP_Group for each process which has a *globally unique ID* (i.e., *xpu_pid*) to identify a process. The global ID is encoded by two numbers: PU-ID (the ID for each processing unit) and the UUID (e.g., PID) on the local OS. This encoding method statically partitions processes to each PUs and can mitigate synchronization costs.

Permission control. Since local OSes cannot maintain cross-PU permissions, we should design a permission management approach in XPU-Shim. This is achieved through the distributed capability system. CAP_Group (per-process) maintains a list of capabilities, including the target distributed object and the permissions. One special permission is *owner* — which can grant the permission to access the object to another process, using *grant_cap*. The owner can revoke the capability using *revoke_cap*. The capability (or permission) is checked in XPUcalls, e.g., a process can only connect to an XPU-FIFO using *xfifo_connect* when it has read or write permission.

3.3 Neighbor IPC

Neighbor IPC (**`nIPC`**) is a primitive to allow a process to communicate with another process (on a different PU), as shown in Figure 4. Compared with local IPC and remote communication (e.g., HTTP over socket), `nIPC` relies on similar network-based communication methods (e.g., PCIe). However, as a single machine, the connection between two PUs is much more reliable. Therefore, a serverless runtime can use a simple software stack for communication and does not need to go through an API gateway.

[1]We use *prcoess* to represent the unit of user applications in PUs.

Table 2: XPUcall. ***The table shows the major XPUcalls implemented in our prototype.***

	XPUcall	Description
Distributed Cap.	int grant_cap(xpu_pid, obj_id, perm)	Grant the capability (perm to access obj_id) to process xpu_pid
	int revoke_cap(xpu_pid, obj_id, perm)	Revoke the capability (perm to access obj_id) from process xpu_pid
Neighbor IPC	xpu_fd xfifo_init(local_uuid, xpu_uuid)	Init an XPU-FIFO
	xpu_fd xfifo_connect(xpu_uuid)	Connect to an XPU-FIFO
	int xfifo_close(xpu_fd)	Close an XPU-FIFO
	int xfifo_read(xpu_fd, buf, length)	Read data from an XPU-FIFO
	int xfifo_write(xpu_fd, buf, length)	Write data to an XPU-FIFO
Misc.	xpu_pid xSpawn(PU_id, path, argv, envp, capv)	Spawn a process in a neighbor PU (PU_id)
	xpu_pid get_xpupid()	Get current xpu_pid

Table 3: The vectorized sandbox abstraction.

OCI interfaces	Description
state <sandbox-id>	Query the state of a sandbox.
create <sandbox-id> <func-id>	Create sandbox with the ID and bundle path, a config.json file is required to indicate details.
start <sandbox-id>	Run a created sandbox.
kill <sandbox-id> <signal>	Send a signal to a created/running sandbox.
delete <sandbox-id>	Delete a sandbox.
Vectorized interfaces	**Description**
state ***vector****<sandbox-id>*	Query the state of a vector of sandboxes.
create ***vector****<sandbox, func-id>*	Create a vector of sandboxes.
start ***vector****<sandbox-id>*	Run a vector of sandboxes concurrently.
kill ***vector****<sandbox-id, signal>*	Send a signal to a vector of sandboxes.
delete ***vector****<sandbox-id>*	Delete a vector of sandboxes.

Currently, XPU-Shim supports a FIFO-styled communication mechanism, i.e., XPU-FIFO. Similar to existing OSes, we use distributed capabilities (or file descriptors) to manage FIFOs for applications. XPU-FIFO is initialized with local and global UUIDs using *xfifo_init*. The local UUID indicates the FIFO in the local OS, and the global UUID is used to allow any processes to connect this FIFO (i.e., *xfifo_connect*). After that, processes can read and write an XPU-FIFO through *xfifo_read* and *xfifo_write*, which is similar to existing FIFO's interfaces. The techniques to implement `nIPC` will be elaborated in §5.

3.4 Miscellaneous XPU Operations

XPU-Shim further provides some XPU operations to ease a serverless runtime to utilize heterogeneous PUs.

***Cross-PU spawn* (`xSpawn`).** XPU-Shim introduces global spawn (abbr. `xSpawn`), which follows the *spawn* interfaces in Unix systems to allow a process to start a new program on other PUs. As shown in Table 2, `xSpawn` requires *PU_id* field, which indicates the target PU of the spawn. XPU-Shim does not share any implicit permissions between parent and child processes, and relies on the *capv* (an array of capabilities) to allow a parent to grant permissions to the child explicitly.

3.5 Vectorized Sandbox Abstraction

Now, we introduce the second key abstraction, *vectorized sandbox*, which handles the hardware heterogeneity.

Open container initiative (OCI) runtime specification [33] is the most widely used sandbox abstraction in serverless computing, e.g., Docker runc (container-based sandbox), gVisor (process-level virtualization sandbox), and Kata container (VM-based sandbox) all implement the abstractions. The core of OCI runtime is five interfaces, as shown in the upper half of Table 3.

The five interfaces are generic enough to abstract sandboxes in many cases, which is one of the reasons why the specification is widely accepted. Next, we explain how we implement the sandbox interfaces for functions running on FPGAs and then discuss the limitations caused by abstraction and how we extend it to vectorized sandbox abstraction.

FPGA serverless functions. Molecule supports serverless functions running in FPGAs. The programming model is similar to normal serverless functions. Developers need first upload their function codes (written in HLS, OpenCL, or even Verilog) to the platform, which will compile the FPGA function with a wrapper (e.g., shell in AWS F1 [8] or an FPGA OS [65, 69]) into a bitstream (or FPGA image). Then, FPGA functions can be invoked as usual serverless functions.

Specifically, we implemented a new sandbox runtime named *runf*, which is responsible for maintaining sandboxes on FPGAs. It will maintain FPGA serverless instance states, which will be used when *state <sandbox-id>* invoked. To create a sandbox running on FPGA, *runf* will download the corresponding FPGA image and program it into FPGA devices. *start <sandbox-id>* will be invoked when the serverless runtime needs to handle a request for the FPGA function. In this case, *runf* transfers the arguments to the FPGA devices and issues a command to the device to execute the function and waits for the results. Similarly, *runf* can erase the FPGA devices to delete a sandbox.

Challenges. Although the above method can implement the sandbox used for FPGA serverless, it has at least two limitations. First, the OCI interfaces have poor scalability on heterogeneous devices like FPGA. Unlike generic-purpose PUs, FPGA can only flush one image at a time. That means we can only run 1 FPGA instance in machines with 1 FPGA device. Even with techniques like partial re-configuration, one FPGA can only support very limited regions [65, 69, 73]. Besides, it also leads to higher startup latency as we need to re-program FPGA devices for most requests.

Second, explicitly deleting sandboxes incurs non-trivial costs that are unnecessary in these cases. In CPU or DPU cases, a serverless runtime should explicitly delete a sandbox to save hardware resources. However, in FPGA, the flushed functions will not occupy any resources and can be easily replaced when a new FPGA serverless function is created. These two limitations motivate vectorized sandbox abstraction.

Vectorized sandbox. Specifically, we have made the following three extensions, as shown in the bottom half of Table 3. First, sandbox creation is vectorized as *create vector<sandbox, func-id>*. That means *runf* can create a vector of FPGA sandboxes at one time. Specifically, *runf* can put the vector of sandboxes into one FPGA image and flush the image containing all the sandboxes. Then, *runf* can directly invoke the target sandbox (if it is cached in the image) when requests arrive. The design can optimize the startup latency as FPGA functions are more likely to be cached in devices.

Second, similar to creation, the start interface is vectorized as *start vector<sandbox-id>*. The extension enables concurrent execution that is beneficial to achieve auto-scalability with limited FPGA devices. For example, the wrapper in FPGA can partition resources into several regions and allow several sandboxes to handle requests concurrently.

Last, Molecule does not explicitly destroy FPGA sandboxes, i.e., the *delete* command will be empty and directly return (but the *runf* will update sandbox states). The real destroy operations happen in the next *create* operation, which will replace the current hardware implementations with the new one. This effectively improves the *delete* performance. The approach will not add overheads to the next *create* operation as it does not include the erasing operation.

Summary. The vectorized sandbox efficiently enables accelerators for serverless computing. It is worth noting that vectorized sandbox relies on the wrapper in the FPGA to guarantee security and performance isolation among instances, e.g., Coyote [69] implements OS abstractions in FPGA and relies on MMU/TLB for isolation between vFPGAs. XPU-Shim leverages the vectorized sandbox abstraction to abstract hardware heterogeneity with great performance.

4 MOLECULE DESIGN

This paper presents Molecule, a serverless runtime on heterogeneous computers. Compared with prior systems [4, 6], Molecule aims to achieve the following goals that are not explored:

- Molecule can manage worker machines with heterogeneous devices, e.g., DPUs, FPGA, smart I/O devices and others.
- Molecule does not rely on the host OS to provide single-OS abstractions but relies on a distributed shim, XPU-Shim, to manage functions on distributed OSes.
- Molecule should support prior serverless mechanisms (e.g., fork-based startup) on heterogeneous computers.

4.1 Heterogeneous Serverless Computing

Programming models. Molecule has similar programming models of AWS Lambda [6] and others [4], as shown in Figure 5. Developers need to write their functions based on a specific language runtime supported by Molecule, e.g., Python. As shown in Figure 5-a, we provide the same language runtime for CPU (x86 server in

```
# Python language runtime
def matmul(n):
  A = np.random.rand(n, n)
  B = np.random.rand(n, n)

  start = time()
  C = np.matmul(A, B)
  latency = time()-start
  return latency

def handler(event):
  n = int(event['n'])
  result = matmul(n)
  return result
```

(a) CPU/DPU function (*matmul*).

```
// FPGA openCL runtime
Vector madd(matA, matB,
  dim0, dim1) {
  int i = 0;
// Auto-pipeline
  for (; i < dim0 * dim1;
        ++i)
  c[i] = matA[i]+matB[i];
  return c;
}
Vector handler(event) {
  //...
  return madd(...);
}
```

(b) FPGA function (*madd*).

Figure 5: Heterogeneous serverless functions. ***The figure shows pseudocode of simplified functions.***

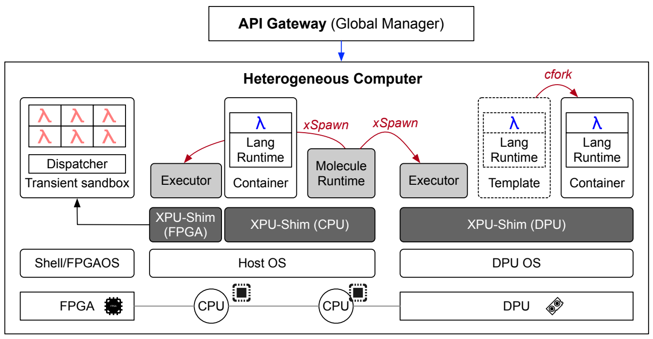


Figure 6: Molecule architecture.

our platform) and DPU (Arm PUs in our platform) as they support general-purpose programming. However, as their computation capabilities are heterogeneous, it is unfair to treat CPU and DPU as the same, especially when serverless promises pay-as-you-go billing models.

Unlike the homogeneous resource model used in existing platforms, e.g., the one-fits-all memory value in AWS lambda [14], Molecule requires end-users to explicitly assign resources to a function, and select the type of PU (i.e., CPU or FPGA) according to their prices and hardware abilities, e.g., DPU has the lowest prices and FPGA has the highest prices. Users can choose multiple settings and let the platform decide how to schedule instances. According to users' configuration, the API Gateway then schedules a function's instance to machines with at least one of the required kinds of PU where the function can execute.

Molecule supports functions running in domain-specific accelerators like FPGA. Developers can write their FPGA functions in our FPGA runtime (Figure 5-b) and upload the functions to the platform to generate an FPGA image. Users can also construct a function chain (or DAG) using different types of functions, e.g., a *frontend* function (on DPU) to pull an image from storage services, and then transfer the image to an FPGA function *gzip* to compress the image. Molecule schedules a function chain in one computer in most cases for better communication performance [43, 61].

System architecture. The architecture of Molecule is shown in Figure 6. Molecule serves serverless requests from the global manager.

It can run on any PU (host CPU in the figure) in a heterogeneous computer and manage functions in other PUs with XPU-Shim. Molecule will launch *executor*s on other PUs through `xSpawn`, which are responsible for managing local function instances using the vectorized sandbox abstraction. The executor receives commands from Molecule (through `nIPC`), executes the commands on the local OS, and returns the results. For accelerators (e.g., FPGA) that cannot launch a generic program, we start a virtual XPU-Shim instance on the neighbor CPU/DPU (e.g., XPU-Shim for FPGA in the figure). This instance is responsible for running the corresponding executor and managing the accelerator.

Next, we introduce how Molecule achieves two important performance goals in serverless: low startup and communication latency.

4.2 Optimizing Startup Latency

A function instance can handle a request with either *cold start* or *warm start*, depending on whether there are available cached sandboxes. The first execution of a function usually begins with a cold start, which needs to prepare function images, create sandboxes, and load function codes. The cold start usually causes long latency, e.g., >1s in complicated Java functions [54, 93]; which are the major costs for most functions.

Fork-based startup [51, 54, 78] and snapshot-based startup [11, 16, 35, 54, 91, 92] are the two most widely adopted optimizations for reducing startup latency. Molecule follows the line of research, with two new contributions. First, prior optimizations utilizing fork to achieve state-of-the-art startup latency requires an additional layer of virtualization, e.g., gVisor's microVM in Catalyzer [54] and unikernel in Seuss [51]. We propose `cfork`, the first container-level fork to achieve <10ms startup latency. `cfork` is designed to support heterogeneous computers.

Second, accelerators like FPGA cannot utilize fork to boost startup latency, bringing new challenges to serverless runtimes. Therefore, Molecule leverages vectorized sandbox abstraction to combine multiple serverless instances into one image, which increases the possibility of hitting a cached instance for incoming requests.

Container fork. Container fork (`cfork`) inherits the idea of sandbox fork in Catalyzer [54] to generate new instances from a pre-prepared *template* container (on CPU and DPU), but overcomes three new challenges to achieve container-level fork on heterogeneous computers.

First, a container may contain multi-threads or even multi-processes that are hard to be cloned correctly and efficiently as Unix `fork` only propagates the forking thread. Catalyzer relies on the underlying hypervisor (i.e., gVisor) to provide the ability to fork a whole sandbox, which is inapplicable for Molecule without the hypervisor layer. To solve this challenge, `cfork` proposes *forkable language runtime*: a wrapper for serverless functions that is responsible for forking multi-thread instances. The language runtime will temporarily merge all the threads into a single thread, save the multi-threaded contexts in memory, and expand it to a multi-threaded one after `cfork`. This lifts the fork mechanism from OS to language runtimes controlled and prepared by serverless platforms. Currently, we have supported forkable Python and Node.js runtime, which accounts for nearly 90% of functions in AWS [36].

Second, Molecule needs to migrate the forked function instances from the template container to a new container for isolation. To satisfy the need, Molecule will prepare a new container (called *function* container) for the forked instance. During `cfork`, the forkable runtime will reconfigure its namespaces and cgroup according to the function's configuration, and then load the function's code (and dependencies if any) and establish a connection to the Molecule. After that, Molecule can assign requests to the child instance, and the child executes them in the *function* container. By default, Molecule prepares generic *template* containers for all functions using the same language, e.g., one Python template for all Python functions. Molecule can launch a dedicated template with code and dependencies for hot functions to further reduce the latency.

Third, `cfork` should support the multi-OS system on heterogeneous computers. In order to fork new instances on multiple PUs, every PU needs to prepare a *template* container for each language runtime. After that, Molecule can utilize `nIPC` to request neighbor PUs to create *function* containers and `cfork` new instances. The new instances will utilize `nIPC` to get requests and return responses.

Caching FPGA function instances. Instead of "fork", Molecule will cache function instances to mitigate the cold-boot costs on FPGA. This is enabled by our vectorized sandbox abstraction. Specifically, Molecule can utilize keep-alive policies (e.g., FaasCache [57] and others [82]) to predict the function instances that should be cached and prepare an FPGA image with those instances. When a request arrives, and the target function instance is cached, Molecule can directly invoke the function (i.e., warm start) without re-programming FPGA.

The number of cached instances highly depends on the design of the wrapper in FPGA, which should provide both isolation and fair-sharing among instances. For example, the state-of-the-art system Coyote [69] can support multiple vFPGA and utilize DRAM stripping to achieve performance isolation, which will be a good choice for Molecule to support serverless computing. Molecule uses a simpler way now to statically partition and protect the resources.

4.3 Optimizing Function DAG Communication

Serverless applications are usually composed of a chain of functions (also named "serverless DAG") [82, 84, 93], making communication latency important. Serverless platforms usually use the network and API gateway for communication in all cases, which can incur unnecessary costs when functions are running in a single machine. State-of-the-art serverless systems [43, 61] adopt local IPC when two function instances are in the same PU and use network mechanisms (e.g., gRPC or HTTP) when they are on different PUs. For example, SAND [43] allows function instances to communicate through a *local bus* with each other when they are on the same PU.

Molecule leverages XPU-Shim to support IPC-based communication in a heterogeneous computer, allowing functions on different PUs to communicate like in a single PU through `nIPC`.

`nIPC`*-based DAG call.* Prior systems [43, 61] usually rely on an intermediate entity, e.g., local bus in SAND [43] and the engine in Nightcore [61], to transfer messages. Molecule implements a "direct connect" method that establishes connections directly between caller and callee function instances, i.e., establishing a full-duplex connection using FIFOs between two functions. To support this, each function will create a *self_fifo* which is named using its UUID (globally unique), and block on the FIFO (i.e., *xfifo_read*). When a request arrives, Molecule injects the UUIDs of caller and callee into each function instance, so functions can communicate with others by writing others' FIFOs (i.e., *xfifo_write*).

Supports for DPU. As nIPC provides the XPU-FIFO abstraction, which is almost the same as local FIFO used in single-PU systems, the supporting efforts for CPU-DPU heterogeneous computers are minor (about 30 LoC changes in Node.js runtime). The major difference is that functions should register their FIFOs to XPU-FIFOs to allow processes on other PUs to access.

Supports for FPGA. FPGA functions can utilize the same FIFO design to transfer data between CPU and FPGA. However, this also leads to two copyings to transfer data between two FPGA functions, i.e., the caller needs to copy the data to host DRAM, and the callee needs to copy the data from the host DRAM back to the FPGA attached DRAM. Molecule implements a zero-copying method by leveraging DRAM data retention [1], an advanced FPGA feature that allows Molecule to load a new FPGA image without erasing the data in the FPGA attached DRAM, i.e., the data is persisted. In this case, the caller FPGA function can leave the data in the FPGA DRAM, and the callee FPGA function can directly use the data without data movement. FPGA wrapper is responsible for clearing sensitive data.

5 IMPLEMENTATION AND OPTIMIZATIONS

We implement XPU-Shim totally in user space for better portability. The core of the XPU-Shim prototype includes 3,542 lines of C/C++ codes (excluding third-party libraries). Besides, we provide an XPU-Shim library (1,460 lines of C codes) that provides the XPUcall interfaces invoked by processes. XPU-Shim will connect and synchronize states among PUs utilizing the underlying interconnect. In our settings, the DPU and CPU communicate through RDMA (which is the only exported PCIe-based communication method), while FPGA and CPU communicate through DMA.

Besides, we implement Molecule based on vectorized sandbox abstraction. In CPU and DPU, the abstraction is implemented based on Docker runc [32] (by always passing one-sized vector). In FPGA, we implement a new runtime, *runf*, which will manage FPGA serverless functions. *runf* relies on an FPGA wrapper to statically assign DRAM banks (or PLRAMs) to an instance; two instances can share a DRAM bank only when they will never execute concurrently (enforced by the wrapper). Molecule supports two mostly used language runtimes, Node.js and Python, accounting for 90% of functions in AWS [36] for CPU and DPU and provides one FPGA language runtime based on OpenCL and Xilinx Vitis [39].

XPUcall optimizations. As the XPU-Shim is another process in the system, we need an IPC approach to communicate between a user-process and XPU-Shim. In a naive implementation, we use FIFO and shared memory to implement XPUcalls. The XPUcall uses FIFO to pass small arguments and uses shared memory to transfer bulk data. However, the two IPC round trips, shown in Figure 7-a, can lead to significantly high costs in DPUs, e.g., 100us in our Bluefield-1 DPU, while the costs in host CPU is about 20us. Therefore, we propose two optimizations.

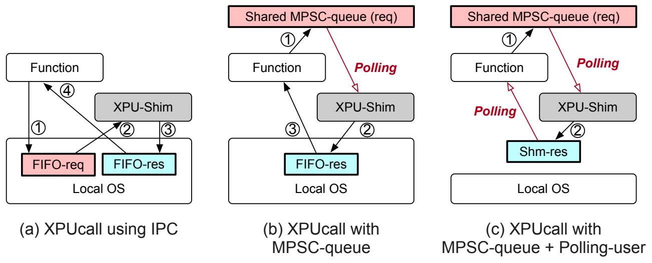


Figure 7: XPUcall optimizations. *MPSC is short for Multi-Producer Single-Consumer.*

Figure 7-b shows the case that XPU-Shim polls on an MPSC (Multi-Producer Single-Consumer) queue and relies on IPC to notify processes for XPUcall responses. This can reduce the IPC round trips from two to one. The optimization is reasonable in devices as XPU-Shim can be pinned to dedicated cores for better performance. Besides, we can further let the processes poll on shared memory for responses, eliminating IPC costs, as shown in Figure 7-c. In our evaluation, we choose the second method as the default one.

To enhance security, the MPSC queue is only used for notifying XPU-Shim about which process has issued XPUcalls. All invocation information is recorded in a per-process shared memory, so a malicious process may only perform DoS attacks by corrupting the queue but cannot learn information of other processes. XPU-Shim also supports multi-threaded handling for XPUcall-intensive scenarios, in which each XPU-Shim thread will handle a dedicated MPSC queue. An alternative implementation is to use the Multi-Producer Multi-Consumer queue to allow work-stealing [49, 53].

Inter-PU synchronization. XPU-Shim follows the ideas of prior multi-kernel [46–48] and partitioned kernel [83] designs to synchronize global states among different PUs through message passing explicitly. We have the following strategies to manage global states.

- *No synchronization with statically partitioned global states*: creating and destroying of processes (i.e., CAP_Group) can (mostly) be processed by XPU-Shim on the local PU because the global PID is encoded with PU ID and local UUID.
- *Immediate synchronization*: states like XPU-FIFO's global UUIDs should be globally unique. Therefore, *xfifo_init* XPUcall needs to synchronize with other PUs to ensure that a global UUID is valid. Besides, we synchronize all capability update operations immediately to ensure permission checking can always finish locally, i.e., no runtime synchronization costs.
- *Lazy synchronization*: XPU-Shim allows harmless stale states. For example, when an XPU-FIFO UUID's reference counter turns to 0 and could be deleted, XPU-Shim revokes the XPU-FIFO's resources but synchronizes the UUID information lazily to other PUs. This enables optimizations like batching to avoid frequent synchronization.

Profile selections. Molecule allows developers to choose multiple settings (or profiles) for their serverless applications, e.g., a function can be scheduled on both CPU and DPU. When a request for a multi-setting application comes, the control plane of Molecule will choose a specific PU based on platform-specific policies. For now, Molecule uses a policy that considers function-chain by locating functions in one chain to the same PU. It is feasible to extend Molecule to use other policies, e.g., machine-learning model-based ones [96].

Keep-alive policies. Molecule inherits the approach used in existing systems [57, 82] for the keep-alive policies. The decision is made by the control plane of Molecule. Molecule now will tend to cache functions in a chain in the same image. We will consider incorporating advanced policies like FaaSCache [57] in our future work.

Limitations. Currently, our prototype does not implement the direct communication between DPU and FPGA; instead, we rely on the host CPU to forward the data between DPU and FPGA, i.e., *CPU-intercepted* communication.

6 EVALUATION

In the evaluation, we answer the following questions:

- *Question-1*: How does XPU-Shim reduce the cross-PU communication latency? (§6.1)
- *Question-2*: Can Molecule achieve better scalability and performance on heterogeneous computers? (§6.2)
- *Question-3*: Can Molecule achieve better performance than commercial serverless systems? (§6.3)
- *Question-4*: Can Molecule achieve low startup latency for serverless on heterogeneous computers? (§6.4)
- *Question-5*: Can Molecule achieve low communication latency for serverless on heterogeneous computers? (§6.5)
- *Question-6*: How does Molecule reduce the end-to-end latency of serverless applications? (§6.6)
- *Question-7*: How is Molecule compared with state-of-the-art serverless systems? (§6.7)
- *Question-8*: How easy is it to support a new accelerator? (§6.8)

We use two settings for evaluation. First, we use one server with Intel Xeon Platinum 8160 CPU (2.10GHz 96 cores in total), connected with two 100Gbps Mellanox Bluefield-1 DPUs, each with 16 ARM cores (800Mhz), as the CPU-DPU heterogeneous computer. Second, we use an AWS F1.x16large EC2 instance (64vCPU) with eight UltraScale Plus FPGAs as the CPU-FPGA heterogeneous computer.

We compare Molecule with Molecule-homo (the homogeneous version of Molecule) and commercial serverless systems like AWS Lambda [6] and OpenWhisk [4]. Molecule-homo does not utilize XPU-Shim; therefore, it can only run on either CPU or DPU (but not both) and cannot utilize accelerators like FPGA. Molecule-homo uses Node.js Express [5] and Python Flask [3] as the baseline DAG methods, which are also used in OpenWhisk. It also does not use optimizations like `cfork`. However, our good implementation of Molecule-homo achieves much better performance than AWS Lambda and OpenWhisk (§6.3). This makes Molecule-homo a more reasonable baseline to illustrate the benefits of hardware heterogeneity and optimizations of Molecule.

6.1 Neighbor IPC Performance

`nIPC`'s performance is at the core of Molecule performance. Thus, we evaluate it first. We compare the performance of `nIPC` with Linux FIFO as shown in Figure 8. We evaluate three `nIPC` cases based on different XPUcall implementations (Figure 7). In all three cases, a caller in DPU will issue a *xfifo_write* and measure the

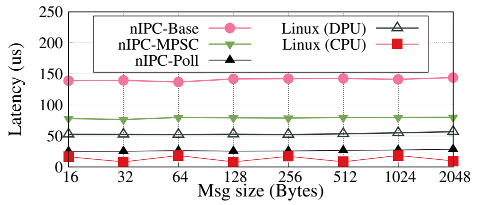


Figure 8: `nIPC` latency. ***The figure shows local Linux FIFO and `nIPC` performances using three XPUcall implementations (i.e., base, MPSC, and MPSC with polling). According to XPUcall implementations, `nIPC`'s latency ranges from 25us to 144us.***

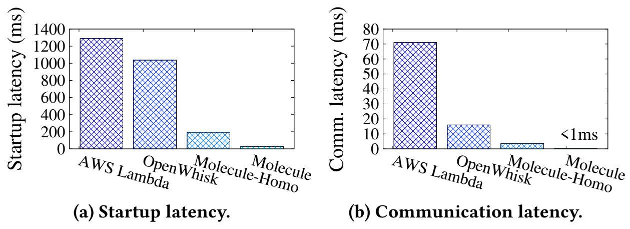


Figure 9: Comparison with commercial serverless systems.

latency as the `nIPC` results. Linux FIFO is the ideal communication mechanism used in state-of-the-art serverless systems for internal calls.

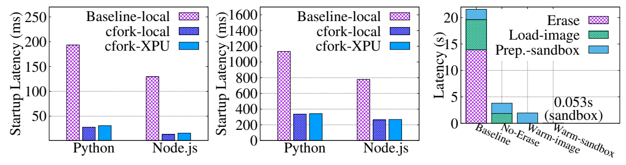
As a result, `nIPC`-Base and `nIPC`-MPSC (i.e., multi-producer single-consumer) have 1.6x–2.8x times higher latency than Linux IPC (on DPU) because of the XPUcall costs. `nIPC`-Polling can achieve about 25us latency, which is even better than Linux IPC (on DPU) because it bypasses the slow kernel on the device; it is still 1.5x–3.1x slower than Linux IPC (on CPU). The results confirm the efficiency of our XPUcall optimizations on devices. We do not apply these optimizations on the CPU because XPUcall causes much fewer costs in the CPU (about 20us) in our settings.

6.2 Benefits of Heterogeneous Serverless

This section explains how the results in Figure 2 are achieved.

Higher function density on CPU-DPU computers. We take one Python image processing function as a case and evaluate the maximum function density in one worker machine. As shown in Figure 2-a, the resources of the CPU in our server only support up to 1000 concurrent instances in the baseline. When we configure the function to run on both CPU and DPU, Molecule can create new instances on DPU that can achieve more than 50% density with 2 Bluefield DPUs. Notably, Molecule does not improve the per-PU density, while the density per computer is improved by utilizing DPUs with Molecule.

Lower computation latency on CPU-FPGA computers. Matrix operations are at the heart of deep learning. Compared with general-purpose CPUs, FPGA is a better choice to perform these computations. We implement three basic operations, *matrix scaling*, *matrix addition*, and *vector multiplication*, into CPU functions and FPGA functions. As shown in Figure 2-b, all the three functions achieve lower latency (2.15x–2.82x) using FPGA functions.



(a) Startup at CPU. **(b) Startup at DPU.** **(c) Startup at FPGA.**

Figure 10: Serverless startup latency. ***Molecule's*** **`cfork`** ***can significantly outperform baseline cold boot, and a remote*** **`cfork`** ***only adds negligible costs with the help of XPU-Shim.***

Table 4: FPGA resource utilization.

	# LUTs	**# REGs**	**# BRAMs**	**# DSPs**
AWS F1 Total	1,181,768	2,364,480	2,160	6,840
Wrapper (12 func.)	119,517 (10.1%)	196,996 (8.3%)	486 (22.5%)	787 (11.5%)

6.3 Commercial Serverless Systems

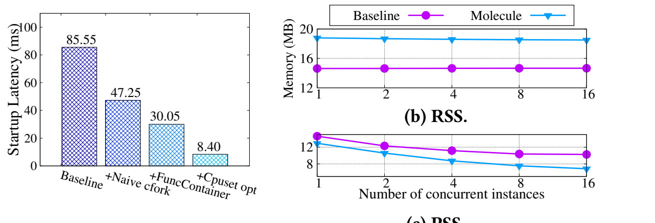
Before evaluating other metrics and applications, we compare Molecule and Molecule-homo with existing serverless systems, including OpenWhisk and AWS Lambda. Specifically, we evaluate two important performance metrics: startup latency and communication latency. We use a helloworld function for startup latency evaluation and an image processing function for communication evaluation (the transferred size <1KB). In AWS, we use the step function as the communication method. As shown in Figure 9, Molecule achieves 37–46x better startup latency and 68–300x better communication latency compared to OpenWhisk and AWS Lambda. Even our homogeneous version, Molecule-homo, achieves 5–6x better startup latency and 4-19x better communication latency.

Next, we elaborate on the details of how our optimizations on Molecule achieve this good performance.

6.4 Function Startup Latency

Startup latency on CPU and DPU. We take Python image processing as a case to evaluate function startup latency in Molecule, as shown in Figure 10-a and b. We compare three cases, the Molecule-homo baseline, the Molecule's `cfork` issued by local PU, and the Molecule's `cfork` issued by a neighbor PU (cfork-XPU in the figure). The results show that Molecule's `cfork` can significantly outperform baseline cold boot. Besides, with XPU-Shim supports, we can fork a remote template with negligible costs (about 1–3 ms).

Startup latency on FPGA. We take vector multiplication as an example to break down the startup latency on FPGA, as shown in Figure 10-c. We consider three stages: erasing the old image, loading the target image, and preparing a software sandbox to invoke functions. The most naive approach (i.e., "Baseline") takes more than 20s to finish the whole process. The major costs come from erasing. As discussed, erasing is unnecessary (for most cases) in FPGA serverless, and Molecule can achieve about 3.8s by skipping the step (i.e., "No Erase"). Molecule can achieve better performance (1.9s) when the function is cached because of vectorized sandbox design. In the best case, if *runf* still maintains the warmed sandbox, we only need 53ms to invoke the function. It is worth noting that optimizing FPGA startup latency is not the major goal of this paper, e.g., Molecule can achieve 20ms startup latency when the FPGA deploys systems like Coyote [69].



(a) `cfork` breakdown. **(b) RSS.** **(c) PSS.**

Figure 11: `cfork` breakdown and memory usages. ***(a) Breakdown of*** **`cfork`** ***optimizations. (b) and (c) present the memory usage.*** **`cfork`** ***saves more memory.***

We analyze the potential to cache instances. Table 4 presents the resource utilization in F1 with vectorized image. Our wrapper containing 12 instances (four instances for madd, mmult, and mscale) only takes 10.1% and 22.5% of the total LUT and BRAM resources. With 8 FPGAs, Molecule can cache 96 FPGA function instances in one computer. Moreover, the FPGA wrapper required by vectorized sandbox introduces space overheads, i.e., 5% lookup tables (logic resources on the FPGA) in F1 for the case in Table 4. Other advanced wrappers take similar costs, e.g., Coyote [69] requires 2-4% base overhead and up to 14% for a full-featured wrapper with four vFPGAs.

`cfork` ***breakdown.*** We break down the `cfork` performance to illustrate how different optimizations work[2], as shown in Figure 11-a. "Baseline" indicates the startup latency using Molecule-homo. "Naive cfork" initializes a *function* container, uses `cfork` to propagate a new process, and assigns the child process to the *function* container's cgroup and namespaces. What's more, "FuncContainer" eliminates the overhead to start a new *function* container, but uses a pre-initialized one to settle the child process. "Cpuset opt" does the same as "FuncContainer" but applies a patch to the Linux kernel, replacing semaphore locks in "kernel/cgroup/cpuset.c" to mutex locks, and thus reduces the overhead to change the cgroup. As we can see in the figure, simply using `cfork` can achieve about 2x better startup performance compared with baseline. When we apply all the optimizations, the function can initialize more than 10x faster than the baseline.

Memory saving. Figure 11-b and Figure 11-c compare memory usages of an image resizing function in Molecule-homo and Molecule under concurrent running functions (from 1 instance to 16). We use the resident set size (RSS) and the proportional set size (PSS) to represent the memory usage. RSS is the total memory used by a process, while PSS is composed of the private memory of that process plus the proportion of shared memory with other processes. Each point in the figure shows the average value of memory usage over all running instances. In Molecule, RSS and PSS also contain *template* container's resources. A comparison of the two figures shows that Molecule achieves lower PSS comparing to the baseline (34% lower for 16 instances). The major reason for the improvement is that `cfork` shares more states among instances. Molecule

[2]The results are evaluated on a desktop machine with Intel Core i7-9700 CPU (8 cores, 3.00GHz), 16GB memory and Linux 5.8.10 kernel.

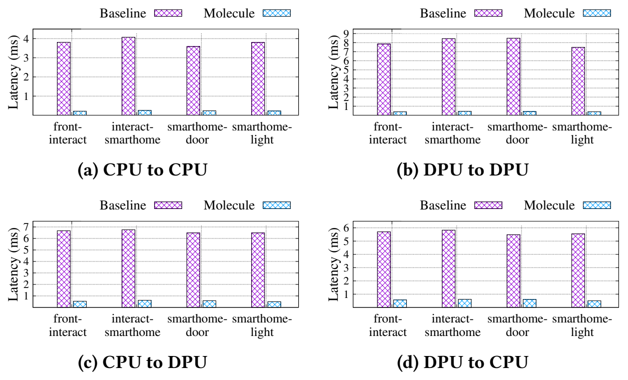


Figure 12: Serverless DAG communication latency. *Molecule achieves 10–18x lower latency.*

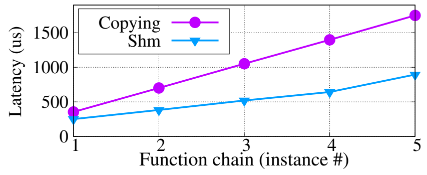


Figure 13: FPGA function chain (end-to-end) latency.

requires higher RSS because of the additional resources required by the *template* container.

6.5 Function Communication Latency

CPU and DPU. We take Alexa skills as a case to evaluate DAG communication latency in Molecule, as shown in Figure 12. We compare Molecule (using IPC and `nIPC`) and the baseline (Molecule-homo using Node.js Express). We consider different cases, including CPU only (Baseline-CPU), DPU only (Baseline-DPU), and cross CPU and DPU. In all the cases, IPC-based DAG optimizations can achieve 15–18x better latency than the Baseline. Although `nIPC` can incur additional costs than local IPC, it still significantly outperforms the baseline method (10–13x).

FPGA. Molecule's `nIPC` utilizes DMA to transfer data between CPU and FPGA functions, which only incurs 50–100us costs to transfer 4KB data. We further compare the basic approach with our shared memory optimization (based on data retention), using an FPGA function chain with five functions to perform vector computation. As shown in Figure 13, the optimizations can effectively improve end-to-end performance (i.e., 1.95x) by mitigating unnecessary data movement.

6.6 Real Applications and Benchmarks

We evaluate Molecule with applications from ServerlessBench [93] and FunctionBench [66]. We compare the end-to-end latency of baseline (Molecule-homo) and Molecule on the CPU, DPU, or both (if the application is a DAG chain). Instances on DPU with Molecule are booted remotely with `cfork` and XPU-Shim's supports. Besides, we evaluate both the cold-boot and warm-boot (instances cache hit) results, shown in Figure 14. We also evaluate three serverless applications with FPGA functions [38], *GZip*, *Matrix-Comput*, and *Anti-MoneyL*, to illustrate the performance benefits of Molecule. We normalize results and label concrete numbers in the figure.

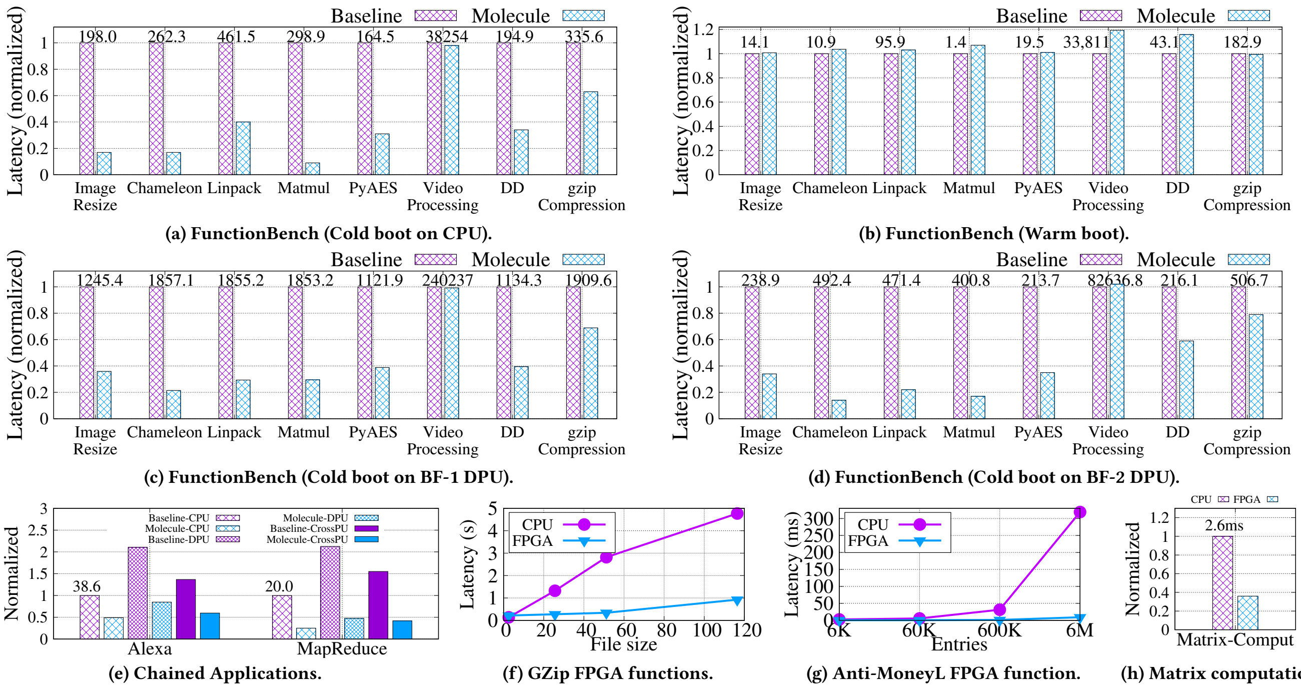
Performance with cold-boot. Figure 14-a and c show the end-to-end latency of eight applications with cold-boot on CPU and Bluefield-1 DPU (or BF-1 DPU). Molecule outperforms the baseline in all cases, achieving 1.01x-11.12x less latency. The major factor for the improvement depends on the execution latency, e.g., Molecule only achieves 1.01x better latency in Video Processing because it takes about 37 seconds for processing a video, which dominates the costs. Instead, we can achieve 11x in Matmul as it is a short function that only takes 3–12ms for processing.

BF-1 DPU requires longer latencies than CPU (4x–7x) because of its low frequencies (i.e., 800MHz ARM cores). We further support Molecule on BF-2 DPU, the state-of-the-art DPU with up to 2.75GHz cores, and present the performance in Figure 14-d. As a result, DPU functions achieve 3x–4x better (compared with BF-1) latencies on BF-2, and are very close to the CPU functions' performance. This indicates that it is reasonable to utilize DPU for serverless computing, which has comparable performance and promises better energy efficiency.

Performance with warm-boot. *Warm boot* means reusing beforehand booted instances to execute a function, which is usually much faster than a cold boot. Consequently, some serverless systems [57] prepare several function instances in advance so that the invocations of these functions can avoid cold boot (i.e., cache hit) and thus optimize startup latency. Figure 14-b shows the warm-boot latency on the CPU, which means each instance is created and cached before the first invocation request's arrival. Both the baseline and Molecule achieve better results as states are warmed, and achieve almost the same results in most cases. Molecule leads to additional costs in some cases because `cfork` will lead to more page faults to handle copy-on-write. This indicates that it is reasonable to cache warm instances through common booting rather than fork when the cache hit rate is high.

Performance with chained functions. We present the end-to-end latency of two chained functions, Node.js Alexa (with five functions) and Python MapReduce (with three functions), as shown in Figure 14-e. Besides the baseline and Molecule results on CPU and DPU, the figure includes latency when distributing functions to different devices (i.e., CrossPU in the figure). Specifically, we ensure that all inter-function calls are cross PU, e.g., in Alexa, the 1st, 3rd, and 5th functions are in the host CPU, and the 2nd and 4th functions are in the DPU. These instances are pre-booted to lower startup overhead's influences on the end-to-end latencies. As shown in the figure, Molecule can achieve 2.04–2.47x less end-to-end latency in Alexa with our IPC/`nIPC`-based optimizations, and 3.70–4.47x less latency in MapReduce.

FPGA serverless applications. We evaluate three serverless applications that FPGA can accelerate. GZip is the application from FunctioBench, which will compress a passed File. We implement the function utilizing Molecule's FPGA runtime and compare their performance with different sized files (from 1KB data to 112MB Linux code). As shown in Figure 14-f, FPGA accelerated Gzip significantly outperforms CPU Gzip when file size is larger than 25MB,



(a) FunctionBench (Cold boot on CPU).

(b) FunctionBench (Warm boot).

(c) FunctionBench (Cold boot on BF-1 DPU).

(d) FunctionBench (Cold boot on BF-2 DPU).

(e) Chained Applications.

(f) GZip FPGA functions.

(g) Anti-MoneyL FPGA function.

(h) Matrix computation.

Figure 14: Serverless applications. ***We present end-to-end latency. The unit is milliseconds unless explicitly declared.***

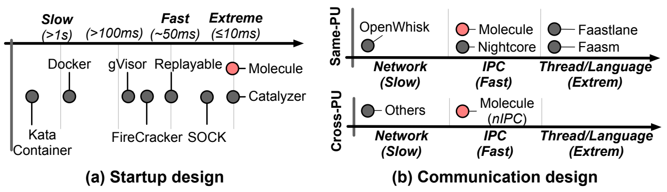


Figure 15: Serverless system design. ***Compared with state-of-the-art serverless system designs (i.e., Kata Container [26], Docker [15], gVisor [22], FireCracker [42], Replayable [92], SOCK [78], Catalyzer [54], OpenWhisk [4], Nightcore [61], Faastlane [70], Faasm [84]), Molecule achieves both low startup and low communication latencies.***

i.e., 4.8–8.3x better latency. Anti-MoneyL is an application for anti-money laundering checking. It will process a set of files with transactions. Figure 14-g presents the latency to process transactions with different entries (6 thousand to 6 million). FPGA accelerated Anti-MoneyL outperforms CPU Anti-MoneyL by 4.7–34.6x. Last, Figure 14-h presents the results of a matrix computation application in CPU and FPGA. FPGA functions achieve 2.8x lower latency.

6.7 Comparison with State-of-the-Art Systems

This section compares Molecule with state-of-the-art serverless systems [22, 26, 42, 54, 61, 70, 78, 84, 92] on optimizations.

Startup optimizations. As shown in Figure 15 (a), snapshot (or checkpoint and restore) and *fork* are the two most widely adopted optimizations for reducing startup latency. For example, Replayable Execution [92] and FireCracker [91] leverage prepared snapshots to mitigate the application initialization cost. Catalyzer leverages

Table 5: Supporting different PUs. ***VSandbox is short for vectorized sandbox. Implementations can be reused by different PUs.***

PUs	VSandbox	XPU-Shim	Programming model
DPU	Modified *runc*	Communicate with DPU through RDMA	Multi-languages
FPGA	*runF* (on OpenCL)	Communicate with FPGA through DMA	OpenCL (can be extended to others)
GPU	*runG* (on CUDA)	Communicate with GPU through DMA	CUDA C++ (can be extended to others)

`sfork` to reuse the state of running instances and achieve sub-millisecond startup latencies. Compared with prior optimizations, Molecule's `cfork` is the first to fork a *container-based serverless function* and overcomes the challenges of forking multi-threaded processes. Besides, `cfork` is the first to support the cross-PU fork, which is necessary for heterogeneous serverless computing.

Communication optimizations. State-of-the-art serverless systems [61, 70, 84] utilize IPC or shared memory for better communication performance, as shown in the upper half of Figure 15 (b). For example, Nightcore [61] proposes *internal call* abstraction, that utilizes Linux FIFO for communication. Faastlane [70] executes functions of a chain as threads within a single process to achieve extreme communication performance (but having worse isolation). Molecule follows the line to utilize IPC for communication. Moreover, Molecule utilizes neighbor IPC to achieve the fastest cross-PU communication latencies with the same IPC abstraction for functions, as shown in the bottom half of Figure 15 (b).

6.8 Generality

Molecule eases the efforts to support new heterogeneous PUs (or devices) for serverless computing. Specifically, with the proposed two general abstractions, developers only need to implement three components to support a new PU, including vectorized sandbox runtime, XPU-Shim and programming models. Molecule is responsible for the rest, including hardware management, functions scheduling, states/data synchronization, and others. Besides the DPU and FPGA presented in the paper, we have preliminarily supported GPU and smartNIC on Molecule, and our experiences indicate that the efforts are small.

Table 5 shows the required components for different PUs. We take GPU as an example. Specifically, we should do the following things to implement abstractions for GPU required by Molecule.

(1) **Implement a vectorized sandbox runtime.** It is called *runG* now for GPU. runG needs to implement the five interfaces defined by the runtime, i.e., create, start, kill, delete, and state GPU kernels/functions. runG is implemented based on CUDA API. Besides, GPU is nature to support vectorized abstraction as a single GPU wrapper (with Nvidia Multi-Process Service) can easily support multiple functions (either using multiple contexts or a single context).
(2) **Implement XPU-Shim interfaces.** XPU-Shim provides a shared view for functions. We implement it for GPU in the same way for FPGA. The XPU-Shim will listen for XPUcalls from GPU functions, handle them, and synchronize data (when necessary) with other PUs.
(3) **Have a programming model of serverless functions for the new device/PU.** In our prototype, a GPU serverless function is a CUDA C++ kernel function and implements a specific interface (the *entry_point*). A wrapper manages the GPU functions implemented based on CUDA API. It is possible to use other models which are orthogonal to Molecule.

With Molecule, these steps are sufficient to enable GPU for serverless computing, and GPU functions can seamlessly cooperate with CPU, DPU and FPGA functions. There are certainly further works and optimizations to do to better utilize GPU for serverless computing, e.g., some developers prefer to use high-level frameworks (e.g., Tensorflow) instead of CUDA. These works are orthogonal to Molecule's goals, and we believe future works can solve them.

7 RELATED WORK

System supports for heterogeneous computers. There is a long line of researches on the system supports for heterogeneous devices [41, 44, 46, 47, 50, 58, 59, 67, 77, 79–81, 83, 85, 86, 90]. Some systems use smart devices to offload computation-intensive tasks [59, 67, 80, 81, 86]. Distributed OSes, e.g., Multikernel [47], Omnix [85], and Popcorn [46], are proposed to run a single OS (with multiple kernels) to manage heterogeneous hardware resources; however, it is non-trivial to implement such an OS with all required primitives. Floem [80] proposes programming abstractions for NIC-accelerated applications. However, the abstractions are too specific for serverless functions to use. Flick [52] utilizes advanced hardware features to support applications running on a heterogeneous ISA computer; however, it requires hardware modifications and is hard to use in existing environment. E3 [72] is a microservice execution platform for SmartNIC-accelerated servers, which can significantly improve the energy-efficiency. However, E3 does not consider serverless-specific requirements, e.g., low startup and communication latencies.

Compared with existing heterogeneous frameworks, we propose XPU-Shim and vectorized sandbox, which are two basic abstractions for heterogeneous serverless computing. We carefully design (only) essential XPUcalls. Molecule is built upon the abstractions and can enable serverless computing on heterogeneous computers with great performance.

System optimizations for serverless computing. Prior works propose many optimizations for serverless computing, including snapshot and fork-based startup [51, 54, 54, 78, 91, 92], IPC-based communication [43, 61], and others [57, 63, 70, 74, 88, 94, 95]. Molecule follows the line of research and proposes `cfork` and `nIPC`-based communication to achieve low startup and communication latency on heterogeneous computers. Other works, e.g., keep-alive policies [57], core scheduling [63], fault tolerance [88, 94], long-running programming models [95], are orthogonal to Molecule, which will be explored and integrated in our future work.

8 CONCLUSION

This paper proposes Molecule, the first serverless system supporting heterogeneous computers. Molecule is carefully designed to abstract away the hardware distribution and heterogeneity with a shim layer (XPU-Shim) and vectorized sandbox abstraction. Our results show Molecule brings significant benefits on scalability and performance. We believe Molecule will motivate many future works for heterogeneous serverless computing.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Amirhossein Mirhosseini and the anonymous ASPLOS'22 reviewers for their insightful suggestions. We thank Tianxia Wang and Xingda Wei for their helps on providing the initial implementations of Python `cfork`. We thank Xuhong Peng and Jun Lu for their helps on providing smartNICs and help us on debugging. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), the National Natural Science Foundation of China (No. 61925206, U19A2060, 61972244), STCSM (No. 21511101502). Corresponding author: Yubin Xia (xiayubin@sjtu.edu.cn).

A ARTIFACT APPENDIX

A.1 Abstract

Molecule is a sandbox runtime for heterogeneous serverless computing. This artifact includes the prototype implementation of Molecule, FunctionBench and ServerlessBench ported to Molecule, and the experiment workflow to run these workloads.

A.2 Artifact Check-List (Meta-Information)

- **Program:** Molecule
- **Data set:** Open-source workloads from FunctionBench, ServerlessBench, and AWS.
- **Hardware:** FPGA is needed for FPGA function tests (available at AWS EC2 F1), and DPU is needed for CPU-DPU tests.

- **Metrics:** Latency.
- **Output:** Performance reports for the test cases.
- **Experiments:** The artificat includes all the scripts and workloads (benchmarks) necessary to reproduce results.
- **How much disk space required (approximately)?:** 10GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 3 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MulanPSL v2.

A.3 Description

A.3.1 How to Access. The source code and benchmarks are hosted on Github: https://github.com/Molecule-Serverless/molecule-artifact.

A.3.2 Hardware Dependencies.

- **CPU**: Servers with x86 CPU are recommended. Users can also use ARM servers for experiments.
- **FPGA**: Users need a computer that is equipped with (at least) one FPGA card for FPGA-based functions. AWS EC2 F1 instance (using AMI: ami-02155c6289e76719a) is perfectly for this case.
- **DPU**: Users need a computer (with x86 CPU cores) that is equipped with Nvidia Bluefield DPUs. As DPU is a generic processor (using ARM CPU cores) and is abstracted by XPU-Shim, the experiments for CPU-DPU settings are almost the same as CPU-only settings. The artifact will use the CPU for the tests. The instructions to configure and use DPU are also provided and can be used for users with DPUs.

A.3.3 Software Dependencies. The following is a list of software dependencies for Molecule and workloads (the listed versions have been tested, and other versions might work but not be guaranteed):

- OS: Ubuntu 18.04/20.04 or Centos-7, Linux kernel 5.4.106.
- Compiler: g++/gcc-7.5.0, go1.15.5.
- Dependent libraries: build-essential, pkgconf, libtool, libsystemd-dev, libprotobuf-c-dev, libcap-dev, libseccomp-dev, libyajl-dev, go-md2man, libtool, autoconf, automake.
- Other dependeicies: Docker

The README on Github provides instructions to install the dependencies.

A.3.4 Data Sets. The evaluation workloads are as follows:

- Eight workloads (Chameleon, Linpack, Matmul, PyAES, Video Processing, DD, gzip Compression, Map-Reduce) from FunctionBench.
- Two workloads (Image Resize, Alexa) from ServerlessBench.
- Three FPGA workloads (GZip FPGA, Anti-MoneyL, Matrix) ported from AWS/Xilinx demos.

A.4 Installation

Molecule is open-sourced at https://github.com/Molecule-Serverless. As the project contains many components, e.g., container runtime supporting fork and language runtimes like Python/Node.JS for functions, we use git-modules to manage them all in the artifact.

The code repository is organized as follows:

- molecule-benchmarks/: the benchmark suite (including source code of serverlessBench) and scripts.
- functionBench/: the source code of functionBench.
- forkable-python-runtime/ and molecule-js-env/: the language runtime (Python and Node.js).
- xpu-shim/: the XPU-Shim source code.
- pychain/: the IPC-based DAG chain for python functions.
- runc/: the runc supporting cfork.
- vsandbox-runtime/: the vectorized sandbox used to manage FPGA functions (based on crun).
- moleculeruntimeclient/: a client to invoke commands on remote PUs through XPU-Shim.
- docs/: figures and documents about the artifact.

To download and build Molecule and the test workloads, users can run the scripts:

```
## Enter to the molecule artifact
cd molecule-artifact
## Update all submodules:
git submodule update --init --recursive
## Build all componets (on x86 CPU):
./build_all.sh
```

If you have Nvidia DPU (with ARM cores), or you are using an ARM server, use the following command to build:

```
## On the Nvidia DPU node
./build_all_arm.sh
```

In addition, each sub-system and the README in Github include detailed instructions to build systems and workloads separately.

A.5 Experiment Workflow

Molecule utilizes docker container to build function images and run the functions. The artifact provides a set of scripts to run the experiments used in the paper, including hello-world demos (for functionalities), benchmark tests (for results reproducibility), and a set of microbenchmarks that help users to understand the details of techniques and custom the artifact for their own usages. All the scripts are well documented, and the results are formatted and explained.

A.6 Evaluation and Expected Results

The evaluation includes both benchmarks and microbench measurement (mostly on latencies).

A.6.1 Benchmarks and Applications. The artifact includes scripts to run benchmarks and applications. The detailed steps to run them are as follows. The scripts will run both the baseline system and Molecule and print the comparison data.

Functionbench. To get the end-to-end latencies of Molecule, users can use the following instructions:

```
cd TOP_DIR/molecule-benchmarks/function-bench
./func_bench.sh
```

Chained applications. To run chained applications (e.g., Alexa) on Molecule:

```
cd TOP_DIR/molecule-benchmarks
# Build runtime and functions
./chained-func/docker_build.sh
# This script will run Alexa chained applications
./chained-func/docker_run.sh
```

FPGA applications. To run FPGA applications on Molecule:

```
cd TOP_DIR/molecule-benchmarks/fpga-apps
./run_bench.sh
```

Output. The artifact has formated the outputs. For example, the results of Functionbench would be like:

```
Function-bench Tests

        Test-Case: LinPack (taking minutes)
=============== fork-startup result ==============
latency (ms):
avg     50%     75%     90%     95%     99%
6.40    5       8       9       9       9
=============== fork-end2end result ==============
latency (ms):
avg     50%     75%     90%     95%     99%
56.00   52      59      70      70      70
=============== baseline-startup result ==========
latency (ms):
avg     50%     75%     90%     95%     99%
177.60  172     177     186     186     186
=============== baseline-end2end result ==========
latency (ms):
avg     50%     75%     90%     95%     99%
203.70  198     203     212     212     212

        Test-Case: Chameleon (taking minutes)
...
```

The above results show that the scripts will explicitly highlight the time required for each test case and the well-formatted results of baseline and Molecule under the same cases. The results are directly matched to the data in the paper.

A.6.2 Microbenchmarks. For users who care about each detailed technique, we provide a set of test cases to evaluate the different techniques proposed by Molecule. For example, the IPC-based DAG case is as the following.

IPC-based DAG. To get the communication latencies of Molecule, using:

```
# Enter molecule-benchmarks
cd TOP_DIR/molecule-benchmarks
./staged-func/docker_build.sh
# This script will run all cases (Figure-12)
./staged-func/docker_run.sh
```

The above commands will run the test cases and dump the communication latencies between different caller and callee, which are the data used in the paper. Please refer to the README in the artifact's Github for more cases.

REFERENCES

[1] 2021. Amazon EC2 F1 Instance Expands to More Regions, Adds New Features, and Improves Development Tools. https://aws.amazon.com/about-aws/whats-new/2018/10/amazon-ec2-f1-instance-expands-to-more-regions-adds-new-features-and-improves-development-tools/. Referenced Aug. 2021.
[2] 2021. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1. Referenced July 2021.
[3] 2021. Apache OpenWhisk docker Runtime. https://github.com/apache/openwhisk-runtime-docker.
[4] 2021. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Referenced 2021.
[5] 2021. Apache OpenWhisk NodeJS Runtime. https://github.com/apache/openwhisk-runtime-nodejs.
[6] 2021. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/. Referenced Jan. 2021.
[7] 2021. AWS LambdaEdge changes duration billing granularity from 50ms down to 1ms. https://aws.amazon.com/about-aws/whats-new/2021/03/cloudfront-lambda-at-edge-billing-granularity/. Referenced July 2021.
[8] 2021. AWS Shell Interface Specification. https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md. Referenced Aug. 2021.
[9] 2021. Azure Functions Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/. Referenced Jan. 2021.
[10] 2021. BlueField-3, the most powerful software-defined, hardware-accelerated data center infrastructure on a chip. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Referenced April 2021.
[11] 2021. Checkpoint/Restore in gVisor. https://gvisor.dev/docs/user_guide/checkpoint_restore/. Referenced April 2021.
[12] 2021. Cloud Functions - Overview | IBM. https://www.ibm.com/cloud/functions. Referenced Jan. 2021.
[13] 2021. Compute Express Link. https://www.computeexpresslink.org/. Referenced Aug. 2021.
[14] 2021. Configuring Lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html. Referenced April 2021.
[15] 2021. The Docker Containerization Platform. https://www.docker.com/. Referenced December 2021.
[16] 2021. documentation/Limitations.md at master - kata-containers/documentation. https://github.com/kata-containers/documentation/blob/master/Limitations.md. Referenced April 2021.
[17] 2021. Firecracker. https://firecracker-microvm.github.io/. Referenced December 2021.
[18] 2021. Fn Project - The Container Native Serverless Framework. https://fnproject.io. Referenced December 2021.
[19] 2021. FPGA Accelerated Cloud Server-HUAWEI CLOUD. https://www.huaweicloud.com/en-us/product/fcs.html. Referenced Nov. 2021.
[20] 2021. FPGA Cloud Compute. https://cloud.baidu.com/product/fpga.html. Referenced Aug. 2021.
[21] 2021. Google Cloud Function. https://cloud.google.com/functions/. Referenced Jan. 2021.
[22] 2021. Google gVisor: Container Runtime Sandbox. https://github.com/google/gvisor. Referenced December 2021.
[23] 2021. Hardware Acceleration over NFV in China Mobile. https://wiki.opnfv.org/download/attachments/20745096/opnfv_Acc.pdf. Referenced Nov. 2021.
[24] 2021. How DNAnexus and Edico Genome are Powering Precision Medicine on Amazon Web Services (AWS). https://aws.amazon.com/blogs/apn/how-dnanexus-and-edico-genome-are-powering-precision-medicine-on-amazon-web-services-aws/. Referenced Aug. 2021.
[25] 2021. Instance families - Instance| Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/doc-detail/25378.htm. Referenced Aug. 2021.
[26] 2021. Kata Containers. https://github.com/kata-containers. Referenced December 2021.
[27] 2021. Marvell OCTEON SDK. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-tx2-sdk-solutions-brief.pdf. Referenced April 2021.
[28] 2021. Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/products/smartnics/innova-2-flex. Referenced 2021.
[29] 2021. Multi-Core Processors - LiquidIO Smart NICs | Network adapter - Marvell. https://www.marvell.com/products/infrastructure-processors/multi-core-processors/liquidio-smart-nics.html. Referenced 2021.
[30] 2021. NetFPGA. https://netfpga.org. Referenced 2021.
[31] 2021. NVIDIA Mellanox BlueField DPU. https://www.mellanox.com/products/bluefield-overview. Referenced 2021.
[32] 2021. opencontainers/runc. https://github.com/opencontainers/runc.git. Referenced April 2021.
[33] 2021. opencontainers/runtime-spec: OCI Runtime Specification. https://github.com/opencontainers/runtime-spec. Referenced July 2021.

[34] 2021. Shim (computing) from Wikipedia. https://en.wikipedia.org/wiki/Shim_(computing). Referenced Dec 2021.
[35] 2021. [Snaps] Full snapshot + restore, firecracker-microvm/firecracker. https://github.com/firecracker-microvm/firecracker/issues/1184. Referenced April 2021.
[36] 2021. The State of Serverless. https://www.datadoghq.com/state-of-serverless/. Referenced Aug. 2021.
[37] 2021. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/smartnic. Referenced 2021.
[38] 2021. Vitis Accel Examples Documentation. https://xilinx.github.io/Vitis_Accel_Examples/2021.1/html/. Referenced Aug. 2021.
[39] 2021. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis.html. Referenced Aug. 2021.
[40] 2021. Xilinx Powers Huawei FPGA Accelerated Cloud Server. https://www.xilinx.com/news/press/2017/xilinx-powers-huawei-fpga-accelerated-cloud-server.html. Referenced Aug. 2021.
[41] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. 2021. Mmapx: Uniform Memory Protection in a Heterogeneous World. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/3458336.3465273
[42] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache
[43] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. $SAND$: Towards High-Performance Serverless Computing. In *2018 $USENIX$ Annual Technical Conference ($USENIX$ ATC 18)*. 923–935.
[44] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 189–203. https://doi.org/10.1145/2872362.2872371
[45] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738
[46] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962
[47] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579
[48] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association.*
[49] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (sep 1999), 720–748. https://doi.org/10.1145/324133.324234
[50] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. $GAIA$: An OS Page Cache for Heterogeneous Systems. In *2019 $USENIX$ Annual Technical Conference ($USENIX$ ATC 19)*. 661–674.
[51] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698
[52] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. Flick: Fast and Lightweight ISA-Crossing Call for Heterogeneous-ISA Environments. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 187–198. https://doi.org/10.1109/ISCA45697.2020.00026
[53] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. 2012. BWS: Balanced Work Stealing for Time-Sharing Multicores. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/2168836.2168873
[54] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512
[55] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108
[56] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.
[57] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757
[58] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, Vol. 99. 87–100.
[59] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 153–165. https://doi.org/10.1109/ISCA.2016.23
[60] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 947–960.
[61] Zhipeng Jia and Emmett Witchel. 2021. 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* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701
[62] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).
[63] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 158–164. https://doi.org/10.1145/3357223.3362709
[64] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. 2018. Iron: Isolating network-based CPU in container environments. In *15th $USENIX$ Symposium on Networked Systems Design and Implementation ($NSDI$ 18)*. 313–328.
[65] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 18)*. 107–127.
[66] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 477. https://doi.org/10.1145/3357223.3365439
[67] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 14)*. 201–216.
[68] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 219–231.

[69] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 20)*. 991–1010.

[70] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[71] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 591–605. https://doi.org/10.1145/3373376.3378531

[72] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[73] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 827–844. https://doi.org/10.1145/3373376.3378482

[74] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[75] Mellanox Technologies. 2021. BlueField Multicore System on Chip. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_BlueField_Storage_Controller_Card.pdf.

[76] Netronome. 2021. Netronome Agilio SmartNICs. https://www.netronome.com/media/documents/PB_NFP-4000.pdf.

[77] Joel Nider and Alexandra (Sasha) Fedorova. 2021. The Last CPU. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3458336.3465291

[78] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. $SOCK$: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 $USENIX$ Annual Technical Conference ($USENIX$ ATC 18)*. 57–70.

[79] Nathan Pemberton, Johann Schleier-Smith, and Joseph E. Gonzalez. 2021. The RESTless Cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 49–57. https://doi.org/10.1145/3458336.3465280

[80] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: a programming system for NIC-accelerated network applications. In *13th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 18)*. 663–679.

[81] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 14)*. 67–80.

[82] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 $USENIX$ Annual Technical Conference ($USENIX$ ATC 20)*. 205–218.

[83] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 18)*. 69–87.

[84] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 $USENIX$ Annual Technical Conference ($USENIX$ ATC 20)*. 419–433.

[85] Mark Silberstein. 2017. OmniX: An Accelerator-Centric OS for Omni-Programmable Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3102980.3102992

[86] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 485–498. https://doi.org/10.1145/2451116.2451169

[87] Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He, Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, et al. 2019. Deep learning inference service at microsoft. In *2019 $USENIX$ Conference on Operational Machine Learning (OpML 19)*. 15–17.

[88] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[89] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 21)*. 495–514.

[90] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. *Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers*. Association for Computing Machinery, New York, NY, USA, 117–131. https://doi.org/10.1145/3373376.3378528

[91] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[92] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 39, 16 pages. https://doi.org/10.1145/3302424.3303978

[93] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery. https://doi.org/10.1145/3419111.3421280

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

[95] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277

[96] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 22, 15 pages. https://doi.org/10.1145/3458817.3476215