Making Serverless Computing More Serverless

Zaid Al-Ali†, Sepideh Goodarzy†, Ethan Hunter†, Sangtae Ha†, Richard Han†, Eric Keller†, Eric Rozner⋆
†University of Colorado Boulder; ⋆IBM Research

***Abstract*—In serverless computing, developers define a function to handle an event, and the serverless framework horizontally scales the application as needed. The downside of this function-based abstraction is it limits the type of application supported and places a bound on the function to be within the physical resource limitations of the server the function executes on. In this paper we propose a new abstraction for serverless computing: a developer supplies a process and the serverless framework seamlessly scales out the process's resource usage across the datacenter. This abstraction enables processing to not only be more general purpose, but also allows a process to break out of the limitations of a single server – making serverless computing more serverless. To realize this abstraction, we propose ServerlessOS, comprised of three key components: (i) a new disaggregation model, which leverages disaggregation for abstraction, but enables resources to move fluidly between servers for performance; (ii) a cloud orchestration layer which manages fine-grained resource allocation and placement throughout the application's lifetime via local and global decision making; and (iii) an isolation capability that enforces data and resource isolation across disaggregation, effectively extending Linux cgroup functionality to span servers.**

I. Introduction

Serverless computing is currently a limited abstraction: the main abstraction is an event-driven system. Developers write functions that serve as event handlers to process a given event and possibly return a response to whatever triggered the event. The serverless framework ensures enough server resources are allocated to handle whatever events occur (at any frequency). In order for the frameworks to scale, the scope of execution is limited to a single function that is assumed to be stateless. Developers can't assume execution on the same server or even the same process among any collection of requests. Any data to persist must be stored through some data store (*e.g.,* Amazon S3). The downside is that not every application is able to be built in such a way, and many times, even highly request-response based applications have a complimentary, more traditional, cloud application coupled with the serverless component.

In this paper, we propose a more general serverless abstraction. Our goal is to present a serverless abstraction that enables the seamless, scale-out features provided by current serverless architectures, while supporting a wide variety of application types in a model that programmers are familiar with – namely, a process, the same abstraction an operating system gives today. Processes can have multiple threads of execution, can access I/O through sockets, and persist state through memory or storage. The challenge, of course, is realizing a process-based serverless framework which can map our serverless process abstraction to an underlying, physically distributed infrastructure. To that end, we propose a new architecture called ServerlessOS to enable our vision and argue three key components are necessary to make our vision a reality:

Fluid Multi-resource Disaggregation: In order to support an abstraction of a process that transparently spans multiple physical servers, we need to break the tight coupling between a process and the underlying physical server resources. That is, we need some form of disaggregation of resources. This has been demonstrated with OS support through works such as distributed shared memory (DSM [11]), which enabled a shared memory address across servers, but was plagued by inefficiencies that resulted in complex cache coherence, which ultimately proved unscalable. More recent work leveraged high-speed networks to separate resources, such as memory from computation, whether transparently through swap space (*e.g.,* nswap [10] or Infiniswap [6]), or explicitly exposed within applications (*e.g.,* RAMCloud [12]). While separation allows for abstraction (and logically decouples a process from any given physical server), it can incur performance overhead when non-local resources are utilized. With ServerlessOS, we argue both disaggregation (to realize the seamless, scale-out process abstraction) and tight coupling (to improve performance) are required in practice. Specifically, ServerlessOS utilizes OS support to make disaggregation work in a practical manner across server resources, and importantly, allows movement (*i.e.,* fluidity) between physical resources to enhance proximity and thus performance.

Fine-grained Live Orchestration: Cloud orchestration today generally focuses on management of virtual machines and containers. APIs can launch virtual instances upon a creation request. Afterward, orchestration is responsible for monitoring and optimizing run-time performance by automatically migrating or scaling workloads. To support ServerlessOS, which has finer-grained management requirements due to disaggregation, orchestration needs to expand to manage the individual resources across the infrastructure. Such management requires the allocation of resources as well as orchestrating the fluidity at run-time on a global level.

Coordinated Isolation: Isolation is a critical component of modern, shared cloud infrastructures that enables appli-

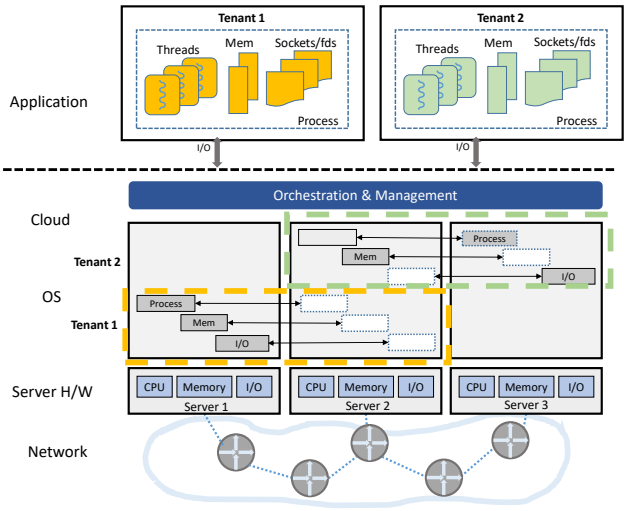

Figure 1. High-level overview of ServerlessOS.

cations from different tenants to co-exist. Isolation allows administrators to ensure quality of service and allocate resources efficiently. Current technology isolates within a single physical machine, consisting of two main capabilities: data privacy (achieved through namespaces) and isolation of resource use (achieved through defining allowable resource consumption via virtual machines or containers). In ServerlessOS, isolation needs to be extended across physical resources (as the process spans across multiple servers), which is more complex than isolating a single function execution replicated across servers (as in today's serverless frameworks).

The rest of this paper outlines the high-level components that integrate to form ServerlessOS.

II. ServerlessOS

We propose moving serverless computing beyond the current limited scope of an event driven framework to support a wider range of programming paradigms. Below, we overview ServerlessOS while also listing its benefits.

Overview. We propose the serverless abstraction should be a process, as illustrated in Figure 1. At a high-level a process consists of an address space, threads of execution, and mechanisms for I/O (whether to the network, via sockets and the OS network stack, or disk, via the file descriptors and OS file system/storage abstractions). Below the process abstraction lies the physical infrastructure abstracted by the serverless framework. The process can actually be running across multiple servers – *e.g.,* process 1 (for tenant 1) is shown across physical servers 1 and 2. To achieve our abstraction through OS level support, three main components are needed to build ServerlessOS. We elaborate each in the following subsection.

Benefits of ServerlessOS. The main benefit of ServerlessOS is it supports general, rather than only event-driven, compute. ServerlessOS provides support for generalized compute by maintaining the same process abstraction developers are familiar with today. This has many positive side effects. For example, with ServerlessOS, developers can immediately redeploy legacy or existing code in serverless environments without having to significantly refactor their application. As the serverless ecosystem is rapidly evolving with many different customized solutions, a process-based serverless model can more easily allow customers to avoid vendor lock-in. As process resource usage grows, ServerlessOS can utilize its fluid multi-resource disaggregation properties to satisfy growing demands, instead of relying on large resource reservations that can decrease workload densities. In other words, a key advantage is ServerlessOS allows cloud providers to treat the datacenter just like application developers treat it: a pool of CPU, I/O, and memory. Finally, note that a process-based serverless model is actually complementary to many current serverless schemes: ServerlessOS can be used to scale resources required by individual functions invoked by serverless handlers.

A. Fluid Multi-resource Disaggregation

Disaggregation is a concept where the tight coupling between different resources traditionally associated with computing (*i.e.,* memory, CPU, and I/O) is broken. Breaking this tight coupling enables greater flexibility – *e.g.,* disaggregating memory [6], [10], [12] has been shown to enable a process to scale the amount of memory beyond the practical limits of a physical server. This is a key component to ServerlessOS as logically the abstraction of a process can be realized across a collection of physical servers. However, disaggregation can cause performance issues without a careful design. To make ServerlessOS practical, we need to retain the performance of tight coupling while gaining the flexibility of disaggregation. We call this *fluid multi-resource disaggregation*, as we can fall back on disaggregation as needed, but can also move resources around (fluidly) to enhance proximity (*e.g.,* co-locate a thread of execution with the portion of data it accesses). Technology such as virtual machine [4] or process [8] migration is a step toward fluidity, but these designs support whole processes or machines, and not disaggregated resources. As such, in this section we discuss fluid disaggregation of each of the resources, and provide insight as to why this is practical in ServerlessOS.

Memory. Memory disaggregation allows a process's actual memory content to not be tied to a physical machine. Memory disaggregation has been demonstrated previously, where execution is pinned to a machine, but the memory of the program is accessed across some network. This model has been demonstrated implicitly at the virtual memory abstraction through the swap interface [6], [10], transparently at the

physical memory level via hotplug and userspace page fault handling [5], and explicitly at the application layer via an explicit API to access remote memory [12]. For performance reasons, it is advantageous to move memory (*e.g.,* a page) to be co-resident with the process currently accessing it. While the disaggregated abstraction implies a rigid physical separation, in each of the above works, fluidity is already present. Their implementations dictate that when a thread's execution accesses some memory that is not co-resident, it pulls (whether implicitly or explicitly) the memory to the thread's machine, where the thread then works on it locally, before putting it back into remote memory.

CPU. CPU disaggregation allows processing to become decoupled from a physical processor. With this, a thread of execution can transition between physical machines. Doing so can improve performance by exploiting locality [7]. Rather than pulling data to the processing, processing can move to where the data is. This tradeoff is dependent on the application: if there is a lot of memory to be pulled in, it makes sense for the processing to jump to that machine rather than pull it all onto the machine where the processing currently is.

There are some initial signs that achieving CPU disaggregation is practical. With multi-core systems where the OS scheduler can schedule a thread on any core, steps toward CPU disaggregation have already been taken. Further, prior work can create a shell of a process on a separate machine (*i.e.,* where the data structures are created in the OS) and can then transfer execution between machines [1]. This work leverages memory disaggregation so transferring execution effectively only requires notifying the receiving machine where the current execution is in the process. This work showed 2-3x speedups over memory disaggregation only, and jumping latency on the order of less than a millisecond (which could probably be improved to 10-30 microseconds based on technologies that optimize inter-server communication [12]).

I/O. I/O disaggregation allows the device which physically captures an I/O event to become decoupled from the physical machine that will service or originate the I/O. For example, with network I/O the network interface card (NIC) might reside on one machine, and the processing of a received packet can reside on a different machine. Network disaggregation can be achieved through proxying (where a device driver on the NIC's machine serves as a proxy and sends to the machine which will process it), or by leveraging memory disaggregation (where the packet is placed in memory and will be pulled to another machine to be processed). In each case, I/O disaggregation only works when the inter-server communication is orders of magnitude faster than the I/O itself (*e.g.,* via a rack backplane). This constraint can be lessened because CPU disaggregation can move processing to the I/O.

I/O disaggregation also includes the case where the I/O occurs on one machine and the processing resides on a set of machines (effectively splitting the I/O). This is what load balancers do today, but for the ServerlessOS framework, we need the abstraction as seen by a process (not by an application designed as a distributed system). That is, we need to be able to dynamically split and merge network sockets (or other I/O channels). Finally, we need to handle the case of being able to move I/O. This might occur if bandwidth needs increase to the point where the physical machine currently performing the I/O can't handle it. In this case, we can move the I/O (move the socket) to a different machine with a higher performing interface by extending socket migration [3] and software-defined networking [9] techniques. Additionally, CPU disaggregation can move processing with a socket.

B. Fine-grained Live Orchestration

We envision an orchestration and management layer will be needed to dynamically apportion resources to disaggregated processes that are distributed across multiple hosts in the datacenter. The structure of such a layer would involve both global and local decision making in terms of which resources (*e.g.,* memory, CPU, I/O, and storage) are devoted to which distributed processes. A global policy maker would determine which subset of nodes and their resources are available to a process for expansion or contraction. The datacenter will need a mechanism for dynamic resource discovery as nodes advertise their availability in terms of how much much memory, CPU and networking they are willing to provide to the serverless infrastructure.

Once the scope of nodes has been defined to which a process may expand or contract, then local decision-making policies will determine when and under what conditions to expand/contract processes. We believe the local decision-making should take into account the state of the local machine and the potential expansion nodes. For example, if a process is under local memory pressure, it may not be beneficial to expand to a remote machine that is also under memory pressure or suffers from overloaded network or CPU because expanding may hurt performance. The state of the remote machine plays as important a role as the state of the local machine on the decision to expand a process.

The decision to move computation between nodes also depends on both local and remote state. For example, if a thread is pulling too much data from a remote machine, then we can move the processing to the node the data resides on. However, if the remote node's CPU is oversubscribed, or its networking is congested, then the decision to move computation may have to be deferred. Similarly, if the local networking cards are congested, then the decision may be to move the computation to a remote machine with available network bandwidth. However, if the remote machine is

under memory pressure, then again the decision to move computation may have to be deferred.

In this sense, remote machines provide backpressure on the jumping/migration algorithm that balances the repulsion due to local oversubscription. These two forces contend with each other, but must resolve so that a clear decision is made about where to execute each thread of a process. This balancing act should be stable on a per-thread basis such that we don't get flapping or oscillatory behavior for any thread.

C. Coordinated Isolation

Serverless is often run in multi-tenant environments, and hence administrators need tools to provide isolation between workloads. Here, we briefly overview how two key elements of isolation, data privacy and resource management, can be supported in ServerlessOS.

Data privacy. Serverless providers need to ensure a serverless application cannot read or write state from another application. Current serverless architectures utilize container namespaces to ensure isolation amongst tenants, but provide no built-in mechanism for functions to access state across machines. Because ServerlessOS uses disaggregated resources, container namespaces must be supported over multiple servers. Extending a common namespace over all of a thread's distributed execution environment enables data privacy in a seamless way.

Resource management. In multi-tenant environments, application developers as well as administrators require mechanisms to control the amount of resources an application consumes. Specifically, it must be easy to place bounds on CPU, memory, storage, and network usage in order to ensure scale-out serverless workloads do not consume an inappropriate amount of datacenter resources. In current single machine environments, Linux control groups (cgroups) already provide necessary bounds on resource usage. With ServerlessOS, the concept of cgroup needs to be extended over multiple machines to support fluid disaggregation. Scaling vertically within a machine can easily be handled by previous techniques [2]. Scaling horizontally poses a new challenge. This challenge, however, is lessened to a certain degree because current cgroup implementations already scale to multiple cores within a single machine. For example, the cgroup CPU scheduler keeps the amount of allowable remaining runtime as global state. To decrease overhead, local cores obtain chunks of processing time from the global allowance and track processing locally [13]. In ServerlessOS, machines servicing an application can similarly acquire chunks of processing time from an orchestration layer. This allows the orchestration layer to coordinate cgroup functionality across disaggregated resources, which ensures aggregated resource limits are adhered to, while providing fluidity in the distribution of resource usage over disaggregated environments.

Last, the cgroup framework needs to become more flexible in ServerlessOS. For example, just as default cgroups can pin a container to a specific set of cores, cgroups in ServerlessOS should be able to pin an application running in a container to a specific set of the disaggregated resources. Such mechanisms can be utilized by the orchestration layer to ensure ACLs, SLAs, or other policies are upheld, or to simply limit the fluidity of a certain application. In ServerlessOS, the configuration of cgroups can be extended to explicitly support the notion of disaggregated resources.

III. Conclusions

This paper presents our vision for serverless computing that provides an abstraction of a process, instead of a function, to enable a broader class of applications to benefit from the simplicity enabled by serverless architectures. Our ServerlessOS design introduces three key components: a mechanism to disaggregate and move process resources across the physical infrastructure, a run-time orchestration layer, and a distributed isolation technique.

References

[1] E. N. Ababneh. *Automatic Scaling of Cloud Applications via Transparently Elasticizing Virtual Memory*. PhD thesis, University of Colorado Boulder, 2017.

[2] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 472–479, June 2017.

[3] M. Bernaschi, F. Casadei, and P. Tassotti. Sockmi: a solution for migrating tcp/ip connections. In *Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on*, pages 221–228. IEEE, 2007.

[4] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[5] B. Caldwell, Y. Im, S. Ha, R. Han, and E. Keller. Fluidmem: Memory as a service for the datacenter. *arXiv preprint arXiv:1707.07780*, 2017.

[6] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[7] A. Gupta, E. Ababneh, R. Han, and E. Keller. Towards elastic operating systems. In *HotOS*, 2013.

[8] Kerrighed. Kerrighed. http://www.kerrighed.org/wiki/index.php/Main_Page/. [Online; accessed 6-Feb-2018].

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[10] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. In *European Conference on Parallel Processing*, pages 1160–1169. Springer, 2003.

[11] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[12] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[13] P. Turner, B. B. Rao, and N. Rao. Cpu bandwidth control for cfs. In *Proceedings of the Linux Symposium*, pages 245–254, 2010.