BulkX: Resource-Centric Serverless for Bulky Application

Zhiyuan Guo Zachary Blanco Zerui Wei Junda Chen Jinmou Li
Bili Dong Ishaan Pota Mohammad Shahrad Harry Xu Yiying Zhang

UC San Diego *University of British Columbia $UC Los Angeles

Abstract

Serverless computing, commonly offered as Function-as-a-Service (FaaS), was initially designed for small, lean applications. However, there has been an increasing desire to run larger, more complex applications (what we call bulky applications) in a serverless manner. Function-based serverless systems cause significant resource wastage when executing bulky applications, as a function is the resource allocation and execution unit, yet function size does not capture application needs.

We propose a new resource-centric serverless-computing model for executing bulky applications, and we build the BulkX serverless platform following this model. The core idea of BulkX is adaptive serverless, where resource allocation and execution automatically and dynamically adapt to application behavior and underlying cluster resource availability. Our results show that BulkX reduces resource consumption by up to 90% compared to today’s function-based serverless systems, while improving performance by up to 64%.

1 Introduction

Serverless computing, commonly offered as Function-as-a-Service (FaaS), is a cloud service that allows users to deploy and execute their applications without managing servers. Serverless computing has gained tremendous popularity in the past few years thanks to its benefit of minimal IT burdens, pay-per-use pricing, and automatic scaling (auto-scaling) [20, 31].

Initially, serverless computing was designed for running short, small-scale functions like simple HTML serving. In recent years, there has been an increasing desire to run large, complex applications such as video processing [14, 30], scientific computing [49, 66], machine-learning tasks [13, 19, 44], data analytics [12, 25], and relational databases [11, 34] in a serverless style (i.e., event-triggered, pay-per-use pricing, auto-scaling, and no management of underlying systems). These applications often 1) run longer or consume more memory than typical FaaS function size limits, 2) exhibit resource usage variations across different phases of computation, 3) require different amounts of resources with different inputs. We call serverless applications with one or more of these features bulky serverless applications, or bulky applications for short.

Today’s practice of running bulky applications in a serverless style is function DAGs [3, 4, 7, 30, 36, 76], where users construct a DAG of functions and set the size of each function in the DAG. This approach creates several resource-waste and performance issues. First, a function’s size is fixed across invocations. Yet, under different inputs, there will be resource waste if provisioning for peak input usage or application failure if under-provisioning. Second, different functions in a DAG execute in different environments (e.g., containers on one server or different servers). Communicating between and setting up the environments can become a major performance overhead, especially when breaking a bulky application into many functions. On the other hand, reducing the number of environments by making functions bigger would cause resource waste, as a function has the same resource allocation throughout its execution; resources allocated for the peak execution needs are wasted during non-peak time of the function. Finally, functions in a DAG store intermediate and shared data in a disaggregated layer, usually over a key-value-like interface [7, 38]. Accessing such a layer causes performance overhead due to network communication and data serialization. The root cause of these issues is today’s function-centric serverless model, where the resource size, resource unit (code piece or data objects), and execution method of a function cannot adapt to application workloads or hardware resource availability.

To address these issues, we propose an adaptive, resource-centric serverless model: users deploy monolithic programs; at each invocation, the serverless system adapts the resource placement, scaling, and execution methods to tightly follow the internal resource requirements of that invocation and the current cluster resource availability (Figure 1).

Following this model, we build BulkX, a serverless platform designed for bulky applications. To be adaptive, BulkX supports several execution methods and scopes. First, BulkX supports the execution of a part of or the whole application in one environment, where all computation and memory accesses are native, i.e., unmodified memory instructions and
function calls within a process (e.g., invocations 1 and 2 in Figure 1). Then, for memory usage, in addition to local memory, BulkX supports the placement of memory objects on a different server from computation and transparently converts applications’ memory accesses into network communication (e.g., invocation 4 T3 data in Figure 1). For computation, BulkX supports the parallel execution of a code piece on one or multiple servers and invoking a function in the user program in a local process or on a remote server.

To execute applications using the above mechanisms, BulkX should first choose the appropriate resource unit, resource size, and resource location to meet an application invocation’s resource needs (i.e., resource-centric). To accomplish this BulkX must capture resource features and their changes within an application’s invocation and across invocations. Instead of imposing runtime monitoring overhead and reacting to every resource change in an on-demand way, our solution is to capture as many resource features prior to the execution as possible, with a combination of user hints and coarse-grained profiling history, so that at runtime, BulkX can make proactive scheduling decisions and sets up execution and communication environments in the background. Specifically, users express where resource changes may happen by annotating their source programs. BulkX identifies the smallest execution units based on user-annotated code scopes and transforms user programs into an intermediate representation we proposed called resource graph. Each node in the graph corresponds to a user annotation, as shown in Figure 2. To decide the size (number of cores, amount of memory) of each graph node, BulkX leverages profiled history when making on-demand, auto-scaling decisions. To place a graph node, BulkX uses a locality-based, greedy placement policy. BulkX co-locates computation and memory as much as possible, first in one server, multiple servers within a rack, and finally across racks.

In sum, unlike serverless DAGs, BulkX’s internal resource graph and profiled history capture fine-grained resource features of an application, and BulkX executes adaptively by “merging” graph nodes into local execution, by supporting non-local execution, and by automatically sizing each execution unit. As such, BulkX achieves optimized performance and resource efficiency for each application invocation.

We evaluate BulkX with a set of microbenchmarks and three real-world applications: data analytics queries from the TPC-DS benchmark, video processing pipelines, and a machine-learning task. We compared BulkX to two FaaS frameworks (OpenWhisk [5], AWS Lambda [7]), four serverless DAG frameworks (AWS Step Functions [7], PyWren [36], ExCamera [30], gg [29]), one serverless resource tuning system (Orion [40]), and two VM-based execution methods (remote-memory swapping [10], VM migration [54]). Our machine-learning results show that BulkX reduces resource consumption by 40% to 84% compared to OpenWhisk while only adding 1.3% performance overhead. Our TPC-DS results show that BulkX reduces resource consumption by 73% to 85% and improves performance by 54% to 64% compared to PyWren. Our video process results show that BulkX reduces resource consumption by 33% to 90% and improves performance by 33% to 47% compared to gg on OpenWhisk.

We will make BulkX publicly available upon acceptance.

2 Motivation and Related Works

This section discusses serverless application trends, the limitations of function-centric serverless computing, and various related works.
2.1 Serverless Application Trends

Serverless applications are becoming increasingly more complex and bulkier. According to a 2021 study, 31% of open-source serverless applications have workflow structures [28]. The popularity of serverless DAGs at Azure witnessed a 6x increase from 2019 to 2022 [41]. Meanwhile, large applications that traditionally run in VMs, such as databases and machine learning, are seeing serverless offerings with better resource and cost efficiency [13, 25, 34, 56].

As applications get more complex and larger, their executions internally have different resource requirements. For example, as shown in Figure 3, the TPC-DS [55] data-analytics benchmark’s query 95 includes five internal stages. They demand drastically different amounts of memory and CPU (as seen by the number of parallel workers). Other applications, such as machine learning, video/image processing workflows, and scientific computing, also exhibit resource-consumption variations within one execution.

At the same time, many serverless applications’ resource consumption depends on their inputs [27, 45, 47]. Figure 4 shows the memory usages of the same TPC-DS query 95 when given input sizes ranging from 10 GB to 200 GB. Across different inputs, the variation of memory usage at each stage can be as huge as 12 times for the same worker function. We observe similar input-dependent resource features in other data analytics (33x more resource when the TCP-DS benchmark dataset size grows by 10x), video transcoding (94x differences when input video changes from 240P to 4K), and many others.

2.2 Function-Based Serverless Limitations

Existing serverless computing solutions mainly follow a function-based model, running applications as a single function or a DAG of functions. We now discuss their limitations for bulky applications.

Resource waste in functions. A Nov 2023 serverless characterization study from a public serverless provider reports more than 90% CPU wastage for 60% of invocations and 80% memory waste for around 50% of functions [37]. The resource inefficiency of today’s function-centric serverless computing stems from its fixed function size throughout the execution of a function invocation and across different invocations. Moreover, today’s FaaS providers like AWS Lambda only support fixed, limited sets of function CPU-to-memory ratios [8, 15], but workloads can have arbitrary CPU-to-memory ratios [16].

To help users save cost, research works [40, 50] and tools [6, 9, 27] find cost-optimal function sizes in public clouds. These works are still confined by the function model and all its limitations. For example, Orion selects an optimal size for a function but has to use the same size throughout the function’s entire execution and for all invocations of the function. Moreover, the most optimal CPU-to-memory ratios may not fall into the few options offered by providers. BulkX avoids such limitations altogether by adapting resources according to applications’ execution needs.

Function DAGs. Several serverless frameworks like Durable Functions [18], AWS Step Function [7], and OpenWhisk Composer [4] let users express their applications as a DAG of functions. Kappa [76] is a continuation-based programming model for long-running applications to work beyond function timeouts by splitting programs into fine-grained pieces based on user-specified continuation. These systems’ main goal is to enable applications to run beyond a single function’s size limits; they execute applications according to the statically written DAGs regardless of runtime resource needs or availability. Once deployed, all invocations of an application use the same DAG and the same function sizes, and each function is executed in a separate environment. As a result, it is difficult, if not impossible, for users to find the right DAG and function sizes: a small DAG with coarse-grained functions wastes resources for some invocations, while a complex DAG with many small functions causes environment communication and start-up performance overhead.

Differently, BulkX dynamically, adaptively, and automatically optimizes application execution based on resource features and cluster availability for each invocation, resulting in optimal performance and resource utilization.

Domain-specific serverless. Apart from generic serverless frameworks, there has been a body of domain-specific works for enabling various application domains to run in the serverless style [11–14, 19, 29, 30, 34, 36, 44, 53, 75]. For example, PyWren [36] is a service that offers Python map primitives on functions that powers serverless data analytics [25, 75] and linear algebra [67]. ExCamera [30] and Sprocket [14] are serverless frameworks for video processing. These frameworks all follow the function-based serverless model and inherit all the function-based resource-waste issues. Moreover, building them requires significant manual efforts for each application domain. Differently, BulkX is a generic solution that avoids resource waste caused by function-based serverless.

Besides research works, there are also commercial domain-specific serverless offerings [11–13, 25, 34, 44]. Unfortunately, no public information about their internals is available. From their pricing information, many are still following the function model. Thus, we suspect they also inherit the resource waste issues of function-based serverless.
2.3 Resource Disaggregation and Migration

We now discuss two techniques for reducing resource waste used in non-serverless settings, resource disaggregation and execution migration, and why they are insufficient for bulky serverless computing.

Resource disaggregation. Over the past few years, there have been a host of research works to disaggregate memory from computation for resource saving [10, 32, 33, 39, 52, 61, 65, 72]. Running bulky serverless applications on a disaggregated system seems a feasible approach for resource efficiency, but this approach is insufficient. First, unlike BulkX, existing disaggregation systems do not autoscale memory or compute resources for different application invocations or execution phases, resulting in resource wastage. Second, disaggregation systems assume an architecture where compute nodes have insufficient memory and always have to make remote data accesses for running applications, either explicitly [33, 61] or via swapping [10, 32]. BulkX sits on a regular server cluster and executes an application fully locally as much as possible.

Migration-based approaches. One potential way for adapting to applications’ internal resource phase changes is to migrate a running container [1] or other execution units [21, 42, 48, 51] to another server when the current server runs out of resources. However, live migration of a bulky application is slow, as it usually involves moving large amounts of memory. Moreover, when an application’s resources grow frequently, a migration-based solution would need to either migrate repeatedly or migrate once but set the new allocation to accommodate the potential peak resource usage, causing significant performance overhead or resource wastage. Nu [60] proposes to decompose a process into smaller proclets, each having a non-shared heap space, while shared data is read-only and replicated across servers. The migration of proclets is lightweight, but Nu has two major drawbacks if applied to serverless computing: 1) by replicating shared data on all servers, Nu causes significant resource wastage; 2) proclets cannot write to shared data, which limits the type of applications Nu can support.

3 BulkX Overview

To use BulkX, users deploy source programs with annotations and their triggering events. Internally, BulkX transforms the original code and dynamically chooses the most appropriate way of executing each invocation of a user application. Centering around our idea of adaptive resource-centric serverless, BulkX consists of several vital components in two parts: background offline tasks happening behind the scene and a foreground runtime system, as shown in Figure 2. Connecting the two parts is an intermediate representation we proposed called resource graph. A resource graph consists of virtual computation or data resource components, each with a history-based resource profile.

BulkX’s offline part (§4) consists of a sample-based profiler, a program analyzer, and a compiler. They work together to capture resource features of an application before the BulkX runtime carries out an invocation of it. BulkX analyzes user-annotated programs and decomposes each application into a resource graph. Our compiler generates two versions of transferred programs, one as native execution and one as remote accesses for all the accesses across resource graph nodes. BulkX samples application runs to perform lightweight profiling. BulkX adds profiled historical information such as object lifetime and size to resource graphs and uses the information at the runtime for proactive, informed scheduling.

The foreground system, BulkX runtime (§5), sits at the application-perceived critical paths. Its goals are to automatically and adaptively allocate and scale CPU and memory resources and to execute applications with high resource and performance efficiency. It leverages the resource graph and the accompanying history resource profiles to proactively perform informed scheduling and scaling decisions. We design a two-level scheduling system that uses a locality-based policy to schedule as many components together or close to each other as possible. On each server, a BulkX executor launches and facilitates the execution of compute and data components in containers. Within each container, application runs with the BulkX runtime library, which executes BulkX internal APIs such as remote memory access.

4 Application Deployment and BulkX Offline

This section introduces how to use BulkX and discusses how BulkX prepares efficient execution prior to the runtime, which we will introduce in §5.

4.1 Application Deployment

BulkX targets applications that are bulky, i.e., resource-hungry or long-running applications that have internal resource variations or input-dependent resource variations. Users write monolithic programs instead of functions in BulkX. Unlike serverless DAGs which uses provider APIs to explicitly communicate across functions and to external data stores [3, 7, 66], user programs for BulkX perform normal procedure calls within the same program as defined by the programming languages, and they use memory with native, unmodified memory allocation and accesses.

On top of native programs, BulkX offers two annotations for users to identify parts of their programs that have distinctive CPU needs and memory resource needs. The first annotation, @compute, applied to procedure calls, indicates a call site that is likely to have different parallelism from the caller, e.g., the caller function executed in a single thread, but the callee executes in multiple parallel threads. Thus, the runtime should allocate different amounts of CPU resources for them. We
Based on user annotations, BulkX transforms a user program into internal components that communicate with one another. It is not tied to any cloud providers. Overall, BulkX’s interface is simple, versatile, and generic, as it is not tied to any cloud providers.

4.2 Resource Graph and Its Generation

After users submit their source programs to BulkX and before their first and subsequent invocations, BulkX prepares their execution under the hood by transforming user programs. BulkX’s preparation centers around a new intermediate representation we propose called resource graph. Each graph node is a compute or data component, with the former being a code site with distinctive CPU usage patterns and the latter being a memory object with distinctive memory usage patterns. Each graph node has a resource feature that we record based on profiled history. Edges in a resource graph represent triggering or accessing relationships.

**Resource graph generation.** Based on user annotations, BulkX generates an initial resource graph where each @data object becomes a compute component, and each @compute code site becomes a data component. BulkX determines triggering or accessing edges by analyzing the program’s control flow and data-accessing relationship. In addition to user-annotated data components, we perform program analysis similar to Mira [33] to identify the shared objects across compute components, which we separate as data components.

BulkX samples an application’s runs to capture the resource usage of each resource graph node (CPU usage for compute components, allocation size and lifetime for data components). It stores a histogram of all captured statistics with decaying weights at each resource graph node.

**Compiling.** BulkX transforms a user program into internal forms for execution using the BulkX compiler. Compilation happens primarily at the offline time, although the BulkX runtime can also trigger the compilation of certain parts of the user program if needed.

Offline, the BulkX compiler transfers each directed edge (one compute triggering another compute component) as an internal API that will be intercepted by BulkX library at the runtime — at this time, the BulkX runtime can decide whether to continue the triggered component in the same execution environment or to launch new one(s).

Offline, BulkX compiles two versions of each compute component that communicates with others (e.g., the top three computes in the graph in Figure 6). The first version assumes that data components are local to all their accessing compute components, and we compile the accesses as native memory instructions. The second version assumes data components are placed on a different server from all their accessing compute components. For this version, we identify all memory accesses to data components and convert them into BulkX’s internal data-access APIs. As frequent API calls into the BulkX library incur high performance overhead, we optimize the generated code by batching accesses to multiple fields in a data structure as one API call to access a larger range.

As will be discussed in §5.1.1, BulkX dynamically determines the execution method for components in a resource graph. There can be cases where some compute components...
co-locate with some of their accessing data components, but remote to the others. We don’t try to precompile for all cases, as enumerating all combinations of local/remote accesses would result in explosive amounts of compilations. Instead, we only pre-compile two versions (all local and all remote) and leave the compilation of the remaining cases to the runtime. To improve performance, once the runtime compiles a version for one invocation, it is cached and reused for future invocations with the same component layouts.

5 BulkX Runtime System

This section discusses how the BulkX runtime system schedules and executes an application’s invocations with diverging resources requirements in an adaptive, proactive, scalable, and reliable way.

5.1 Adaptive Scheduling and Execution

To be adaptive, BulkX innovates on scheduling policy and execution/auto-scaling mechanisms, with the overarching goal of achieving the best performance and resource efficiency for each invocation of bulky applications.

5.1.1 Adaptive Scheduling Policy

Overall, our scheduling policy tries to co-locate compute components and their accessed data components, as doing so avoids remote data accesses. It also tries to co-locate triggered and triggering compute components to avoid environment startup overhead and output/input messaging overhead.

Specifically, when scheduling an application’s invocation, the scheduler first tries to find servers that can fit the entire application in the hope that all components can run in one environment. It chooses the server with the smallest available resources among them to leave more spacious servers for future larger invocations (e.g., invocation 1 in Figure 6). The scheduler marks the chosen server’s with the potential need for the amount of resources of the whole application as estimated from profiled history. We do not allocate resources for future use in a resource graph. Instead, the marked resources are given low priority when the scheduler allocates resources for other applications. The chosen server starts the application in a container whose size is set according to the first compute component’s profiled resource needs. If this compute component accesses data components, the server also allocates the data components locally if possible. Afterward, if this server still has enough resources for the next components, BulkX chooses the same server and continues the computation in the same container process but changes the container size to what the new components need.

If there is not enough resources to co-locate components on any servers, BulkX finds a server with the smallest available resources to launch each of them. For instance, in Figure 6, at invocation 2’s T2, Server 1 runs out of resources, and BulkX launches the blue component on Server 2. BulkX then launches the remote-access compilation version or generates a new compilation on the fly if none exists.

When scaling up resources, BulkX also follows the same locality policy. For example, BulkX first tries to expand a data component on its current server. If that is not possible, BulkX picks another server and prioritizes servers already running compute components that access the data component.

Like BulkX, some prior works also try to co-locate functions. A recent work, XFaaS [62], proposes running multiple functions concurrently within one process and re-using runtime environments across invocations for serverless computing in the Meta private cloud. Differently, BulkX does not co-run different applications, as our target environment is multi-tenant public clouds. Moreover, unlike XFaaS, BulkX is resource-centric instead of function-centric.

5.1.2 Adaptive Execution and Autoscaling

We now discuss how BulkX adaptively executes and scales data and compute components. Our high-level idea is to adapt the statically generated resource graph into physical execution units based on cluster availability and resource feature profiles. As such, multiple components in the original graph can be materialized as one physical component, while one original component can turn into multiple physical components. The former can happen for two reasons: BulkX groups neighboring compute/memory components into one when they have similar lifetime and scaling patterns over most history executions, and BulkX co-locates components in one execution environment. The latter happens when resource needs grow, and BulkX automatically scales out a compute/memory component.
Compute component execution. We execute compute components in containers (Docker in our current implementation) by initially sizing them according to profiling results and loading appropriate binaries (local or remote-access versions). For the local-access version, compute components run directly on Docker. BulkX only assists in mmap-ing local data components and forwarding the compute-component execution output to the scheduler. In the non-local mode, the BulkX runtime executes remote-memory access APIs (which are generated by the BulkX compiler) on top of an RDMA or TCP network stack.

BulkX allows a compute component to use arbitrary amounts of memory or CPU as long as they do not exceed the user-specified limits. Thus, BulkX may need to auto-scale compute components beyond the profiled configurations. BulkX detects when a compute component is under memory pressure and asks the scheduler to allocate additional memory space. The scheduler may schedule the additional space on the same or different server. As such space is added after the launching of a compute component, BulkX uses it as (local or remote) memory swap space and transparently performs swapping at the user level.

CPU resource autoscaling. BulkX supports both scaling up and scaling out. For scaling up (and down), BulkX simply changes a compute component’s assigned vCPU. BulkX periodically measures CPU utilization. If the utilization is 100%, we increase vCPU by 1. BulkX’s scaling out (and down) applies to compute components that have parallelism (such as group and sample in Figure 5). BulkX determines the vCPUs to use based on request load and use thresholds of adding/reducing vCPUs determined by history profiles. For example, when an earlier invocation uses 10 vCPUs for 10 parallel execution of a compute component and has 50% CPU utilization on all the vCPUs, a future invocation of 10 parallel execution would only use 5 vCPUs.

Data component launching and autoscaling. BulkX automatically starts a data component when the first compute component accessing it in the resource graph starts. The memory controller at the server chosen by the scheduler allocates a memory space with the size indicated in the resource graph. The data component ends when the last compute component accessing it ends or explicitly releases it.

When a data component co-locates with a compute component accessing it on one server, the memory controller allocates a virtual memory space for the data component and mmaps it to the compute-component container. When a data component is not co-located, the memory controller manages it as a virtual memory space in a reserved BulkX process for a TCP cluster or an RDMA memory region for an RDMA cluster, both of which are accessed by compute components via the network. We explain how BulkX isolates memory regions in Appendix.

A data component can grow beyond its initially allocated size because of application’s additional allocation. In this case, the BulkX memory controller on the server picked by the scheduler allocates an additional memory space. When a compute component is currently accessing a data component natively, BulkX mmaps the additional memory space on the same node or performs swapping if the additional memory space is on a different node (T3 in Figure 6). When a compute component is currently accessing the data component via BulkX APIs, the BulkX library will transparently issue network requests to multiple servers for accesses spanning separated memory spaces.

5.2 Proactive Scheduling and Execution

Different from existing serverless systems that schedule and auto-scale resources on demand, BulkX proactively schedules resources and sets up environments before resource change happens by leveraging profiled history, as discussed below.

5.2.1 Proactive Scheduling

Starting up the execution environment for a serverless application is a slow process that usually involves the start-up of a container/lightweight VM, language runtimes, and various libraries [64]. A single start-up time is not significant for long-running bulky applications, however, such an application can have a large resource graph and involve many rounds of start-up overhead. To avoid this overhead, we pre-launch a subsequent component in a resource graph when the current component is running. This pre-launching approach is also adopted by Orion [40], but Orion’s pre-launching is based on user-supplied static serverless DAG, while BulkX’s resource graph is an intermediate representation that adapts dynamically. Similar to traditional serverless frameworks [59, 64, 69], we also pre-warm the first component in a resource graph based on historical invocation patterns.

5.2.2 Asynchronous Communication Setup

As communicating or accessing components in a resource graph can sit on different servers, especially in a highly utilized rack, it is crucial to ensure that both the actual communication
We describe the detailed tuning method in Appendix.

We let the scheduler send one component’s physical location (in the form of executor ID) to its communicating component’s (data plane) and the connection setup (control plane) are fast. To improve the overall performance of executing applications, we support both RDMA and TCP communication, with several optimizations on the control path and the data path. Similar to existing remote-memory works [33], we optimize the data path by batching small network requests and by caching fetched data locally. For RDMA, we perform one-sided, zero-copy communication.

The communication control-path optimization is unique to serverless computing, as we need to minimize network connection set up time, while allowing the direct communication between components in the data plane. The general challenge in serverless communication is that a function cannot easily know where other functions are. Prior serverless communication systems rely on an overlay network to build connections [71]) or pre-establishes connections between all pairs of endpoints [24, 46, 73, 74]. The former is slow, while the latter does not meet our resource elasticity goal. Both try to establish connections via direct channels between two nodes.

Our idea is to leverage non-direct but already established connections to assist the initial location discovery. Our observation is that a component has always established a connection with the scheduler by the time it starts up. Moreover, the scheduler is the one who decides and thus knows the physical locations of the two components that will be communicating. We let the scheduler send one component’s physical location (in the form of executor ID) to its communicating component’s network virtualization controller when initiating the two components. With the location information, the two components can easily establish a datapath connection (e.g., RDMA QP). To further optimize the startup performance, we initiate the location exchange process as soon as the execution environment is ready while the user code is loaded in parallel. Doing so hides the location-exchange overhead behind the performance-critical path (Figure 23).

5.2.3 History-Based Resource Adjustment

To improve the overall performance of executing applications in a BulkX cluster, we dynamically set the initial allocation size and the adjustment amount. Rather than adjusting components based solely on current metrics, we incorporate historical usage to avoid over-adjusting for one-time input changes.

BulkX decides how to scale each component with two parameters, initial size and incremental size. BulkX tunes these values to strike a balance between performance and resource efficiency. The initial size of a component decides the resource to allocate at the start-up time. During runtime, BulkX autoscales the resource amount one incremental size at a time to avoid frequent small resource adjustments. BulkX re-adjusts these two sizes periodically after $K$ (e.g., 1000) executions of an application. For compute components, we have initial and incremental sizes for both vCPU amount and memory amount. For memory components, we have sizes for memory amount. We describe the detailed tuning method in Appendix.

5.3 Scalability, Reliability, and Consistency

We end the design section by discussing how BulkX scales its scheduler, handles failure, and supports consistency.

5.3.1 Scalable two-level scheduler

As a bulky application often has multiple scheduling entities (components in a resource graph), as well as frequent runtime scale to adapt to inputs. BulkX needs a scalable scheduling mechanism. We propose a two-level scheduler architecture where a cluster has one global scheduler, and each rack within has a rack-level scheduler. When a user event triggers an application, the request is sent to the global scheduler (possibly after a data-center load balancer directs requests across multiple clusters). The global scheduler maintains the rough amount of available resources in each rack. It uses this information to direct the application request to a rack by balancing loads across racks. It then looks up the application’s compilations in the compilation database and sends the compilation and corresponding resource graph to the rack-level scheduler.

The rack-level scheduler handles allocation requests for each component in a resource graph. It chooses a server to initially run the application based on policies in §5.1.1. Upon completing a compute component, its result is sent to the rack-level scheduler via reliable messages (§5.3.2). The release of a data component or the need to scale it up are also sent to the rack-level scheduler. The rack-level scheduler increases the server’s available resources accordingly and decides where to schedule the subsequent component or the scaled-up part of a component. It then updates the resource counting of the destination server and notifies that server’s executor to start the component. Thus, the rack-level scheduler always has an accurate view of available resources in all the servers in the rack and what components currently run on each of them. If a rack runs out of resources for a request, the rack-level scheduler sends the request back to the global scheduler to find another available rack.

5.3.2 Failure Handling

Traditional FaaS-based serverless re-executes the entire function when failure happens. This works well when applications are small, but bulky applications can be time- and resource-consuming to re-execute. To avoid re-executing an entire bulky application after a failure, we send the result of every compute component to the rack-level scheduler via reliable messaging (e.g., Kafka [35] in our implementation). When a failure happens, BulkX discards the crashed component and all data components it accesses. BulkX locates the latest cut of a resource graph where all the edges crossing the cut have been persistently recorded. BulkX re-executes from this cut using the persistently recorded input messages.

We apply the same failure-handling principle to data components. BulkX does not make data components durable or
replicate them, as they only store intermediate state [38]. If any memory region in a data component crashes, BulkX discards all the compute components that access it and all other memory regions of the same data component. BulkX then finds the latest reliable resource graph cut to restart.

Resuming from the reliably recorded messages in the above manner is sufficient to guarantee at-least-once semantics, which is the same as today’s serverless systems [22, 68]. Our failure handling design balances foreground performance and failure recovery performance compared to mechanisms that take more checkpoints [17] or those that restart the entire function chain after a failure [4, 7].

5.3.3 Consistency and Synchronization
Multiple compute components can access the same data component as shared memory. BulkX provides basic distributed synchronization primitives instead of a particular consistency scheme. Users can use the BulkX synchronization primitives like distributed locks to enforce critical sections. As BulkX runs on generic RDMA or TCP network, all communication is through messaging, and we do not provide automatic coherence across components. New coherent media like CXL could potentially be used with BulkX to provide hardware-level coherence, which we leave for future work.

6 Evaluation

Implementation. We implemented BulkX’s compiler and program analysis on top of Mira [33]. We implemented BulkX’s runtime system on top of Apache OpenWhisk [5]. In total, BulkX includes ~19.6K SLOC: 3.2K lines of change on Mira, 10K lines of Scala to extend OpenWhisk’s scheduler and executor, and 4.6K lines of C together with 1.8K lines of Python for runtime and libraries. Currently, BulkX supports programs written in Python, the second most commonly used programming language in serverless [26], and in C++, a common language used for bulky applications.

Environment. We evaluated BulkX on a local rack of a Dell PowerEdge R740 servers each equipped with two 16 core Intel Xeon Gold 5128 CPU, 64 GB of memory, and a 100 Gbps Mellanox Connect-X5 NIC. We use eight servers for running BulkX and systems in comparison and four additional servers to run Redis. All the servers are connected to a 100 Gbps FS N8560-32C 32-port switch. All servers run Ubuntu v20.04 with Linux v5.4.0. We use Docker v24.0.7 for containers.

6.1 End-to-End Application Results
We first present the high-level results of running three bulky applications on BulkX: data analytics with Pandas [43] running the TPC-DS benchmark [55], video processing pipelines, and simple machine-learning training using logistic regression.

6.1.1 Distributed Query with TPC-DS
The first application we evaluate is distributed data analytics on Pandas running the TPC-DS benchmark [55]. We compare BulkX with PyWren [36] running on OpenWhisk on our local cluster. PyWren splits each TPC-DS query into several stages, each containing some DataFrame operators. For each stage, PyWren generates different number of workers to perform the computation, and we use Orion [40] to set the resource for each worker. Intermediate data storage for PyWren uses Redis servers in the cluster. We implement each query as a single Python program and annotations.

We use multiple input dataset, whose size ranging from 2 GB to 1 TB to evaluate three TPC-DS workloads with different performance characteristics and varying resource requirements: queries 1, 16, and 95. They read 2.5 GB, 20 GB, and 19 GB of data respectively and have peak memory usage of 240 GB and peak CPU of 120 vCPUs.

Overall cost and performance. Figure 8 shows BulkX’s and PyWren’s total memory consumption. PyWren failed to scale efficiently with multiple isolated functions. BulkX reduces memory consumption by 72.5% to 84.8%. PyWren provisions each stage for its peak memory usage. Moreover, each worker in PyWren fetches all the data it will access from Redis before the computation starts and stores the data back after all its computation finishes. Essentially, PyWren pays for the same amount of memory consumption twice. In contrast, BulkX right-sizes each component in a single stage by auto-scaling on demand. Moreover, BulkX does not incur the double-memory-consumption problem. Both systems achieve max CPU utilization of 120 cores. In average, BulkX utilize 91.2% of CPU time, much higher compared to PyWren’s 63.8%, as function-centric scale causes resource stranding. Figure 9 plots the total run time of the three queries. BulkX outperforms PyWren by...
To understand the effect of different BulkX we perform a similar ablation study on video transcoding using the encoding/decoding operators developed by ExCamera [30]. ExCamera divides video frames into batches and uses one parallel worker to encode a batch. Our version is a single program with 11 annotations, where input video is sliced into parallel segments and each segment are processed with up to 16 parallel compute units. We use ExCamera’s set up of six frames as a single encoding unit and a batch size of 16 units. BulkX generates a resource graph with 37 compute and 33 data components.

We compare BulkX to ExCamera and ExCamera running on the gg [29] serverless frameworks. Original ExCamera uses a fixed VM to schedule and merge results, while running decoding steps parallel in serverless workers, while gg is a pure serverless implementation. The gg implementation represents the encoding of each frame batch with 80 functions, each reading from and writing to a Redis intermediate storage pool. We also compare with a local vpxenc encoder [70] that runs everything natively on one server in our cluster.

**Ablation study.** To understand the effect of different BulkX techniques, we add one technique at a time and show the resulting memory consumption and application performance in Figure 10 using the TPCDS-16 query. We start with the baseline of a static function DAG as used by PyWren. We then use our static resource graph instead of the original function DAG and execute each graph node in a separate environment. Thanks to a resource-oriented decomposition, by scaling on resources instead of functions, it greatly reduces resource consumption both in used and unused memory. It slightly improves application performance, but still has relative high remote access and scheduling overhead. Afterward, we add the adaptive scheduling and execution support (§5.1), which co-locates many graph components and use local communication interfaces, thus greatly improves application performance. Lastly, we add the support of proactive scheduling/execution and history-based resource adjustment (§5.2). The former improves performance because of environment/connection pre-setup. The latter improves resource efficiency because of more accurate resource sizing and less runtime remote scales.

### 6.1.2 Video Processing

Our second application is video processing, where we perform video transcoding using the encoding/decoding operators developed by ExCamera [30]. ExCamera divides video frames into batches and uses one parallel worker to encode a batch. Our version is a single program with 11 annotations, where input video is sliced into parallel segments and each segment are processed with up to 16 parallel compute units. We use ExCamera’s set up of six frames as a single encoding unit and a batch size of 16 units. BulkX generates a resource graph with 37 compute and 33 data components.

We compare BulkX to ExCamera and ExCamera running on the gg [29] serverless frameworks. Original ExCamera uses a fixed VM to schedule and merge results, while running decoding steps parallel in serverless workers, while gg is a pure serverless implementation. The gg implementation represents the encoding of each frame batch with 80 functions, each reading from and writing to a Redis intermediate storage pool. We also compare with a local vpxenc encoder [70] that runs everything natively on one server in our cluster.

**Overall cost and performance.** Figure 11 plots the total execution time of the three systems processing a 1-minute slice of the movie “Sintel” [58] in three resolutions: 240P, 720P, and 4K. The peak resource usage hits the cluster resource limit of 120 CPUs and 174GB of peak memory in both Openwhisk and BulkX. BulkX achieves the best performance for all three resolutions. BulkX performs adaptive materialization and launches 81.3% of components in same batch on the same server. In contrast, even when servers used for the gg experiments have the same amount of free resources, gg’s functions share data that always lives on separate Redis servers. vpxenc runs in one server, but it is limited by the amount of parallelism it can achieve: only 18 out of 32 allocated cores and 14 out of 16GB allocated memory are actually used. This difference is more apparent with larger videos that benefit from larger parallelism.

Figure 12 and Figure 13 plot the total memory and CPU consumption. BulkX consumes the least amount of memory. As function-based frameworks, ExCamera and gg set the same function size for all inputs. Thus, it has to make provisions for the largest anticipated inputs, causing significant unused resources for the smaller 240P and 720P videos. vpxenc has similar or lower used memory as BulkX, but its unused memory amount is much larger. As a single-unit execution, its size needs to be set to the peak memory size and cannot be adapted to requirement change over time.

Similar to memory, ExCamera, gg, and vpxenc also incurs high CPU resource wastage because of their peak-based static provisioning. Unlike memory, their used CPU resource is also much higher than BulkX with 720P and 4K resolutions, mainly because of their longer execution time.

**Ablation study.** We perform a similar ablation study on video processing, as shown in Figure 14. Similar to TPC-DS, each added technique reduces memory consumption for video processing. Differently, video processing incurs high overhead in scaling memory objects with small increments, which causes performance downgrade when simply changing from function DAG to resource graph. Performance is greatly improved.
As discussed in §2.3, Memory Consumption (GB * s) Execution Time (s) Execution Time (s) 0.0 0.2 0.4 20 0.6 30 0.8 200 1.0 40 300 1.2 50 400 1.4 60 500 1.4 70 600 1.6 80 700 1.8 90 800 2.0 100 When adaptive and proactive optimizations are added.

6.1.3 Logistic Regression

The third application is machine learning training using logistic regression (LR) ported from Cirrus [19]. Here, we focus on comparing with a variety of serverless baselines. It first loads the input data set and separates it into a training set and a validation set. Then, it performs the regression on the training set. Finally, it validates the trained model with a validation set. BulkX’s resource graph for LR includes four compute components corresponding to the above steps and three data components (training set, validation set, and learned weights).

We compare BulkX (RDMA and TCP) with executing the original program on vanilla OpenWhisk and on FastSwap [10], a system optimized for remote-memory swapping. OpenWhisk executes the application as one function sized to the peak memory size. FastSwap uses the same amount of local memory as BulkX’s compute component and remote memory of the peak memory size. We run these systems on our local cluster to have controlled network environments. We also ran a set of configurations on AWS: 1) running the entire program as one Lambda function, 2) running the same four code pieces as BulkX compute components in four Lambdas orchestrated by AWS Step Functions [7] and storing data components in S3 or Redis [57]. For Step Functions, we use two settings: SF-CO (cost optimized): size Lambda to achieve the lowest cost [6, 9, 27]; and SF-Orion: size each function using Orion [40].

Figures 15 and 16 plot the total memory consumption of these schemes when processing two input dataset sizes, 12 MB and 44 MB, which result in peak memory usage of 0.78 GB and 2.4 GB respectively. As all schemes use the same number of vCPUs in this experiment, CPU consumption comparison directly translates to comparing execution time, which we show in Figure 17 for the 44 MB input. Overall, BulkX consistently achieves the lowest resource consumption, highest resource utilization, and best performance, and the improvement is higher with the small input. Even when running on TCP, BulkX still largely reduces resource consumption with small performance overhead.

Among local baselines, FastSwap has the highest resource consumption and performance overhead among the three, because it wastes a significant amount of remote memory by allocating for the peak (no autoscaling) and its swap-based coarse-grained remote memory access overhead. Comparing the AWS baselines, running the same resource graph on Step Functions only reduces 2%-5% memory resource consumption with SF-CO and SF-Orion compared to single Lambda, far from how much BulkX reduces over OpenWhisk. Digging deeper, we found that splitting a program into Lambda functions introduces both performance and resource overhead, as seen in Figure 17. Each function needs extra time to read/write data from S3 or Redis and serialize/deserialize the data. Serialization and deserialization also requires extra memory space. These results demonstrate that simply writing an application as function DAGs on today’s serverless platforms is not enough. Additionally, all AWS baselines have huge resource wastage, especially with the small input, for several reasons. FaaS services only allow one function size for all invocations and throughout an invocation’s execution. Moreover, AWS fixes the CPU-to-memory size ratio, causing waste of CPU or memory. Finally, the long-running Redis instance is provisioned for peak resource usage and wastes huge amounts of resources at non-peak times.

6.2 Closer Looks

To further understand BulkX, we performed a set of additional evaluations to focus on various features of BulkX.

Alternative runtime scaling methods. As discussed in §2.3, resource disaggregation and migration are two possible ways of adjusting resources and avoiding wastage. We compare BulkX with 1) swap-based disaggregation (to be fair, we run all BulkX’s systems except that we perform swapping for all accesses beyond local memory), 2) a best case for live migration by only counting pure data movement time using full 100 Gbps network bandwidth, 3) migration with MigrOS [54], and 4) vanilla OpenWhisk. For a fair comparison, we let all systems use the same amount of total resources (not counting unused resources).

As shown in Figure 18, for different inputs, the size of Join stage varies from 267 MB to 14.7 GB. with BulkX’s adaptive materialization, all components run locally with the scaling factor 100, and 34% of data components’ memory regions are scheduled at remote servers with the scaling factor 1000. Disaggregation solutions like the swap results we show always place data in remote servers, causing high application-execution performance overhead because of network com-
munication. With migration-based solutions, the application execution runs natively and has no overhead. However, migrations cause significant overhead even with an optimal best case, because bulky applications’ memory consumption is high (SF 1000). Finally, OpenWhisk incurs significant performance overhead for the reasons covered before.

**Adaptation to different inputs.** To understand how BulkX adapts to different inputs, we use the TPC-DS Query 1 with five input sizes (5, 10, 20, 100, and 200 GB) to compare BulkX with PyWren. Figure 19 shows the memory consumption comparison in relative terms, with absolute numbers reported on the top (CPU consumption is similar and not included). BulkX’s memory consumption is consistently lower than PyWren. BulkX adapts its component sizes at the run time according to inputs, thus resulting in negligible resource wastage. PyWren uses one function size across inputs, resulting in significant resource wastage, especially when inputs are small. Its used memory is also higher than BulkX because of the reasons discussed in §6.1.1.

Figure 20 plots the corresponding execution time. BulkX achieves good performance, especially with small input sizes, thanks to its adaptive-materialization. BulkX is able to schedule everything locally for scale factors 5 and 10, 96% local for SF 20, and 95% of parallel compute components distributed evenly across four servers. BulkX is able to pack most non-shared data components local to the compute components, with the shared data components accessed remotely.

**Adaptive placement.** To compare different adaptive placements of BulkX, we run the `ReduceBy` operator within TPC-DS Q16. It involves a set (3 to 120) of parallel compute components passing their result to a single compute component (fan-in), with one data component used for data sharing per sending compute component (total data size from 730 MB to 113 GB). Figure 21 shows three cases: local runs everything in one machine, remote-scale scales up data components to a remote machine (with 46% being local), and disagg runs all components on different machines. Application execution time increases with more components being remote, and most of the overhead comes from pure I/O movement. The system overhead of setting up execution environments and network connections is small.

**Proactive resource adjustment.** Figure 22 plots the performance and memory utilization with three schemes: fixing initial and incremental sizes to 256 MB and 64 MB, provision data components to the highest usage (peak-provision), and BulkX’s history-based size setting. We use real-world serverless application memory usage profiles from Azure [64]. In addition to the average of all the datasets, we choose a few representative applications to highlight. Small: most invocations’ memory usages are small, Large: most invocations used a lot of memory, and Varying: different invocations’ memory usages vary largely. Detailed invocation resource usage distributions could be found in Appendix. Overall, BulkX achieves high utilization and good performance. Peak provisioning achieves the best performance since no auto-scaling or swapping is needed, but has low memory utilization. A fixed size configuration can lead to both poor resource utilization (when real usage is below the configuration) and poor performance. (when frequently adding many physical memory components).

**Scheduler scalability.** Our evaluation shows that BulkX’s global scheduler can process 50k application invocations per second, 2.5 times faster than OpenWhisk’s cluster scheduler. The rack-level scheduler can schedule 20k components per second. We further analyze the component processing rate a rack can execute. Our evaluated applications have an average of around 1 second duration per component; each (compute) component consumes one core on average. A rack typically has less than 1,000 cores. This means a rack’s computation rate is 1,000 components per second. Applying the same type of calculation using today’s serverless function duration and resource consumption [64], we get 9,804 functions per second to be a rack’s computation rate. Both computation rates are much smaller than our scheduler’s 20,000/s request rate. This analysis is robust regardless of the number of components in an application. Overall, BulkX can sustain a cluster of around 100 to 1000 racks, which is more scalable than OpenWhisk. As enable resource sharing between clusters already achieves high resource utilization, future implementations could deploy multiple global level scheduler to achieve better scalability and performance.

**Proactive execution and network setup.** Figure 23 shows the impact of BulkX’s communication techniques on a workload with one compute component accessing one data component in warm started environments with no established connections. The first bar represents OpenWhisk without an overlay network, while the second bar adds the overlay network to enable direct component-to-component communication. The third bar replaces the TCP stack in the second bar.
with BulkX’s RDMA stack, improving component execution time. The fourth bar removes the overlay network and uses BulkX’s network virtualization module, reducing initialization time. Finally, the fifth bar includes BulkX’s asynchronous messaging optimization, hiding network connection setup time.

7 Conclusion

We presented BulkX, a resource-centric serverless framework that executes bulky applications adaptively. Our thorough evaluation results demonstrate BulkX’s performance and resource efficiency over existing function-based serverless frameworks. We hope BulkX can allow more types of applications to port to and benefit from serverless computing. Future works can extend BulkX by adding more features like more versatile consistency support, more comprehensive compiler features, and more programming language support.

References

[1] Criu. https://criu.org/Main_Page.

[2] Or-tools - google optimization tools. https://github.com/google/or-tools, 2023.

[3] Azure Durable Functions. Accessed 2022-04-16. https://azure.microsoft.com/en-us/pricing/details/functions/.

[4] Apache OpenWhisk Composer, Accessed 2023-04-17. https://github.com/apache/openwhisk-composer.

[5] Apache OpenWhisk: Open source serverless cloud platform, Accessed 2023-04-17. https://openwhisk.apache.org/.

[6] AWS Lambda Power Tuning, Accessed 2023-04-17. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[7] AWS Step Functions, Accessed 2023-04-17. https://aws.amazon.com/step-functions/.

[8] Google Cloud Functions Pricing, Accessed 2023-04-17. https://cloud.google.com/functions/pricing.

[9] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. COSE: Configuring Serverless Functions using Statistical Learning. In IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, 2020.

[10] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys ’20), New York, NY, April 2020.

[11] Amazon Web Services. Amazon Aurora Serverless v2 is generally available. https://aws.amazon.com/about-aws/whats-new/2022/04/amazon-aurora-serverless-v2/, April 2022.

[12] Amazon Web Services. Amazon Redshift Serverless is now generally available. https://www.databricks.com/blog/announcing-general-availability-databricks-sql-serverless, July 2022.

[13] Amazon Web Services. Amazon SageMaker Serverless Inference is now generally available. https://aws.amazon.com/about-aws/whats-new/2022/04/amazon-sagemaker-serverless-inference/, April 2022.

[14] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A serverless video processing framework. In Proceedings of the ACM Symposium on Cloud Computing, pages 263–274, 2018.

[15] AWS. Configuring Lambda function memory, Accessed 2021-12-13. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html.

[16] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys ’23, page 381–397. ACM, 2023.

[17] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. In Proceedings of the ACM on Programming Languages (OOPSLA ’21), New York, NY, October 2021.

[18] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Serverless workloads with durable functions and netherite, 2021.

[19] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ML workflows. In Proceedings of the ACM Symposium on Cloud Computing (SoCC ’19), November 2019.

[20] CB Insights. Why Serverless Computing Is The Fastest-Growing Cloud Services Segment. https://www.cbinsights.com/research/serverless-cloud-computing/, September 2018.

[21] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In 2nd Symposium on Networked Systems Design & Implementation (NSDI 05), Boston, MA, May 2005.
[22] Google Cloud. Cloud Functions execution environment - execution guarantees, Accessed 2023-04-17. https://cloud.google.com/functions/docs/concepts/exec.

[23] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michał Podstawski, and Torsten Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference (Middleware ’21)*, Québec city, Canada, December 2021.

[24] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. rFaaS: RDMA-enabled FaaS platform for serverless high-performance computing. *arXiv preprint arXiv:2106.13859*, 2021.

[25] Databricks Inc. Announcing the General Availability of Databricks SQL Serverless. https://aws.amazon.com/about-aws/whats-new/2022/07/amazon-redshift-serverless-generally-available/, May 2023.

[26] Datadog. The state of serverless, August 2023. https://www.datadoghq.com/state-of-serverless/.

[27] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference (Middleware ’21)*, New York, NY, May 2021. ACM.

[28] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 2021.

[29] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winston. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, 2019.

[30] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winston. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI’17)*, Boston, MA, USA, 2017.

[31] Global Market Insights. Serverless Architecture Market size to cross $90 Bn by 2032. https://www.gminsights.com/pressrelease/serverless-architecture-market, December 2022.

[32] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, March 2017.

[33] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP ’23, Koblenz, Germany, 2023.

[34] InfoQ. Microsoft Announces the Preview of Serverless for Hyperscale in Azure SQL Database. https://www.infoq.com/news/2023/02/azure-sql-hyperscale-serverless/, February 2023.

[35] Jun Rao Jay Kreps, Neha Narkhede. Kafka: A distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB ’11)*, Athens, Greece, June 2011.

[36] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC ’17)*, Santa Clara, CA, September 2017.

[37] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC ’23*, page 443–458. ACM, 2023.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’18)*, Carlsbad, CA, October 2018.

[39] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms, 2022.

[40] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the Three Rights: Sizing, Bundling, and Pre-warming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’22)*, Carlsbad, CA, July 2022.
[41] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnietky, Saurabh Bagchi, and Somali Chaterji. WiseFuse: Workload characterization and DAG transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

[42] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. XvMotion: Unified Virtual Machine Migration over Long Distance. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 97–108, Philadelphia, PA, June 2014. USENIX Association.

[43] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O’Reilly Media, Inc., 2012.

[44] Microsoft Azure. Model training on serverless compute. https://learn.microsoft.com/en-us/azure/machine-learning/how-to-use-serverless-compute?view=azureml-api-2&tabs=python, November 2023.

[45] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. Parrotfish: Parametric regression for optimizing serverless functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC ’23, page 177–192. ACM, 2023.

[46] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’19)*, Renton, WA, July 2019.

[47] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys ’21)*, Virtual, April 2021.

[48] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX ’05)*, Anaheim, California, April 2005.

[49] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Leveraging serverless computing to improve performance for sequence comparison. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB ’19, page 683–687, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Joe H. Novak, Sneha Kumar Kasera, and Ryan Stutsman. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems and Networks (COMSNETS)*, 2019.

[51] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002. USENIX Association.

[52] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ram-cloud storage system. *ACM Transactions Computer System*, 33(3):7:1–7:55, August 2015.

[53] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 131–141, 2020.

[54] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. MigROS: Transparent Live-Migration support for containerised RDMA applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 47–63. USENIX Association, July 2021.

[55] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD ’02)*, Madison, Wisconsin, June 2002.

[56] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’19)*, Boston, MA, February 2019.

[57] redislabs. Redis. https://redis.io/, 2009.

[58] Ton Roosendaal. Sintel. In *ACM SIGGRAPH 2011 Computer Animation Festival*, SIGGRAPH ’11, page 71, New York, NY, USA, 2011. Association for Computing Machinery.

[59] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. IceBreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’22)*, Lausanne, Switzerland, April 2022.
Appendix

8.2 Adaptive Compilation and Generated Memory Interface

When compiling into resource graph, BulkX identifies the dataflow dependency on annotated function invocations, through tracing the usage of their return values. Further, all accesses to annotated data objects and variables in context are identified. Functions including instructions that uses the return value of @compute calls will be divided at the location that uses the results from @compute calls. Recursive call to @compute is not supported in BulkX.

When generate compiled code, For remote invocations, extra code is generated to request the mmaped location of remote objects and get/set variables in context.

BulkX adaptively generate different interfaces when materialization, if BulkX knows the source and target memory types. If an access is mainly on local memory, a local interface is generated, otherwise, remote accessing interface adapt to access pattern will be generated. For example, for accessing memory over RDMA, BulkX will consider the remote memory latency and access pattern using [33]. For synchronization, it will be generated using local sync or remote sync implemented by BulkX. Certain patterns are required for BulkX to generate efficient access interface.

It’s possible that a single memory component is launched as multiple physical components to different locations, for example, a compute component is accessing both local and remote memory objects. This could happen before or after the accessing compute component is scheduled, that is, before or after the memory access interface is generated. Both local and remote accessing interface could handle the case: raw memory interfaces accesses remote memory through swapping, and remote interface could access local memory through local cache section described in Mira [33].

This interface also includes a set of BulkX low-level APIs as shown in Figure 24 that a compute component code can use to access other memory components or communicate with other compute components.

9 Additional Design Details

9.1 Memory Component Isolation

On memory side, different memory regions must not share same page and are isolated through the virtual memory system or the paging.

On accessing memory regions, as discussed in §5.2.2, we support both one-sided remote memory accesses using RDMA and two-sided communication using TCP. For RDMA, we assign each physical memory component its own memory region...
(MR) and own protection domain (PD) for proper isolation. When a memory controller receives a request from the scheduler to start a new physical memory component, it launches one process to allocate this physical memory component’s size and registers it with RDMA with a new MR and a new PD. Afterwards, accesses to the physical memory components are all one-sided operations that do not involve the memory controller (§5.2.2). For two-sided TCP, we use the memory controller to allocate memory in a global virtual memory space at physical memory component launching time. It then receives/responses memory-accessing messages and copies message data to/from the allocated memory space. As all accesses go through the trusted BulkX memory controller, physical memory components are properly protected even in one global memory space.

9.2 Swap System for Compute Components

Our remote-memory swapping happens entirely in the user space using Linux userfaultfd and is transparent to user applications. Specifically, the BulkX runtime uses a background thread to monitor page faults caused by the user application threads. When a fault happens, if there is not enough swap space, the runtime asks the scheduler to create and launch a new physical memory component. The runtime’s background thread swaps out pages whenever it detects memory pressure. Since the user-space fault handler cannot access the page table and would not know the page access pattern, we use an NRU (not-recently-used) policy by swapping out a page that has not recently been swapped in.

9.3 Resource adjustment algorithm

With a goal of minimizing resource waste and maximize the performance at the same time, we model the proactive allocation process as an linear optimization problem. The goal of the algorithm is for each component to select the best initial resource amount \(init\) and increment scaling size \(step\).

\[
\min_{\text{step,init}} \quad init + \sum_{h \in \text{History}} \text{step} \times k_h \times \text{cost_factor} \\
\text{s.t.} \quad h \in \text{History} \mid k_h \times \text{step} + \text{init} > h \\
\sum_{h \in \text{History}} \max(\text{init} - h, 0) \times \text{exec_time}_h \times h < \text{Thres} \\
\]

\(\text{cost_factor}\) is a factor that models the scaling cost. We use ortools [2] to solve the optimization problem.

9.4 RDMA-based Communication Control Path

When establishing an RDMA connection (i.e., RDMA QP) between two nodes, they need to first exchange a set of metadata describing their own identities. However, two components cannot easily reach each other or establish a connection to perform this initial metadata exchange due to the dynamic and isolated nature of serverless computing. Prior solutions either use a costly overlay network layer (which accounts for nearly 40% of startup time in our experiments or require container runtime changes for performance improvement [71]) or pre-establish all connections [24, 46, 73, 74] (which does not fit our need to dynamically launch memory components). Both approaches try to establish connections via direct channels between two nodes.

Our idea is to leverage non-direct but already established connections between executors and schedulers to exchange the initial metadata message. Our observation is that a component has always established a connection with its rack-level scheduler by the time it starts up. Moreover, the scheduler is the one that decides and thus knows the physical locations of the two components that will be communicating. We let the scheduler send one side’s (e.g., \(B\)’s) physical location (in the form of executor ID) to the other side’s (e.g., \(A\)’s) network virtualization module when initiating the components. Afterwards, \(A\)’s network virtualization module sets up and sends the necessary RDMA connection metadata of \(A\) together with the destination component’s executor ID (\(B\)) to the scheduler. The scheduler routes the message containing \(A\)’s metadata to the target executor, which then gets sent to the destination component (\(B\)) by its network module. After both sides acquire the other’s metadata message, they establish an RDMA QP. The entire QP establishment takes only 34 ms.

To further optimize the startup performance, the QP establishment process starts as soon as the execution environment is ready, while user code is loaded in parallel. Doing so hides the QP establishment overhead behind the performance-critical path.

Furthermore, we reuse an RDMA QP when a component tries to establish the communication with a physical memory component located on a server that already has another physical memory component communicating with the component. Since the new and the existing physical memory components are both accessed by the same component, there is no need to isolate them and one QP is enough for both components.

9.5 RDMA-based Communication Data Path

BulkX’s RDMA communication data path bypasses kernel. When a compute component accesses another component (compiled as an internal communication API), the BulkX runtime looks up which QP it corresponds to. It then issues a one-sided RDMA operation for \text{read}/\text{write} for memory components or a two-sided RDMA operation for \text{send}/\text{recv} for compute components. As one virtual component can be mapped to multiple physical components, if the internal API call touches a memory region that is on two physical memory components, the BulkX runtime dispatches two one-sided

\(^1\)References are in the main submission file.
We evaluate the maximum throughput BulkX’s resource savings. We implemented the solver using the high efficient Python Mixed-Integer Linear Program (MIP) package. Finding the optimal solution for 10000 disaggregation candidates each with 32 components takes 10ms-15ms. Performance/cost on a fixed cluster. BulkX’s resource saving reflects at the cluster level. We evaluate BulkX and OpenWhisk with the same total cluster resources as shown in Figure 30. First, BulkX could allocate more resources compared to resource-tuned serverless, because of resource-level scaling reduces resource stranding and utilizes free resources across server; Second, most of allocated resources are utilized in BulkX, due to the runtime resource scaling adapts to resource size. As a result, given same amount of resources, BulkX could utilize more resources in real computing, results in a 33% to 90% performance gain compared to Openwhisk.

10 Additional Results

We provide four sets of additional evaluation results. Auto-scaled compute components with memory swapping. To measure our swap system performance, we use a simple microbenchmark of sequentially or randomly reading an array of different sizes. Figure 25 plots the total run time of the microbenchmark with increasing array size. We test two local cache sizes: 200 MB and 400 MB, and compare the swap performance against the case where the local memory is larger than the whole array size (i.e., no swapping and the ideal performance but impossible with the compute pool configuration). Overall, swapping only adds 1% to 26% performance overhead. The overhead is higher when the array size is bigger and when the local memory size is smaller.

Scheduler Scalability. We evaluate the maximum throughput of BulkX’s rack-level scheduler and find that it can handle around 20K scheduling requests per second. This scheduling rate is similar to OpenWhisk’s scheduler. However, thanks to our locality based design with a scheduler-per-rack, the number of total messages that the system can handle scales linearly with the number of racks. We also find the request rate to the top-level scheduler to be light (around 50K requests per second) and is never the bottleneck in the system.

Per-input component adjustment. We present more details of the workloads used for evaluating per-input component adjustment (§6.2.3) Figure 26 shows the four selected applications’ memory utilization distribution across invocations. Large and Small features a large or small average memory, which is far different from our default setup. Varying features a relative high variance, which can be representative for applications with varying resource requirements. Stable in contrast, features almost the same resource utilization across all invocations. For comparison, we also plot a straight line of 256 MB, which is the default initial memory allocation size in BulkX.

Small Application Performance. Figures 27 and 28 showcase the execution time and resource consumption of five small serverless functions from SeBS [23] and FaaSProfiler [63] benchmarks. These single-function applications have an execution time of less than one second and memory usage smaller than 128 MB. Although these applications do not benefit from BulkX’s resource-centric scaling, BulkX still outperforms OpenWhisk in terms of resource consumption while delivering similar performance. This is attributed to BulkX’s ability to flexibly allocate resources.

Solver Performance. We implemented the solver using the highly efficient Python Mixed-Integer Linear Program (MIP) package. Finding the optimal solution for 10000 disaggregation candidates each with 32 components takes 10ms-15ms.

Figure 25: BulkX Performance with Swap Component. Figure 26: Cold and warm start up time. Figure 27: BulkX Execution Time on Small Applications. Figure 28: BulkX Resource Consumption on Small Applications. Figure 29: Memory Usage Distribution of Azure Dataset. Figure 30: Cluster Level Memory Utilization.