HiveMind: A Hardware-Software System Stack for Serverless Edge Swarms

Liam Patterson
lgp36@cornell.edu
Cornell University
Ithaca, New York, USA

David Pigorovsky
dap263@cornell.edu
Cornell University
Ithaca, New York, USA

Brian Dempsey
bjd86@cornell.edu
Cornell University
Ithaca, New York, USA

Nikita Lazarev
nl524@cornell.edu
Cornell University
Ithaca, New York, USA

Aditya Shah
as2564@cornell.edu
Cornell University
Ithaca, New York, USA

Clara Steinhoff
cfs232@cornell.edu
Cornell University
Ithaca, New York, USA

Ariana Bruno
amb633@cornell.edu
Cornell University
Ithaca, New York, USA

Justin Hu
jh2625@cornell.edu
Cornell University
Ithaca, New York, USA

Christina Delimitrou
delimitrou@cornell.edu
Cornell University
Ithaca, New York, USA

ABSTRACT

Swarms of autonomous devices are increasing in ubiquity and size, making the need for rethinking their hardware-software system stack critical.

We present HiveMind, the first swarm coordination platform that enables programmable execution of complex task workflows between cloud and edge resources in a performant and scalable manner. HiveMind is a software-hardware platform that includes a domain-specific language to simplify programmability of cloud-edge applications, a program synthesis tool to automatically explore task placement strategies, a centralized controller that leverages serverless computing to elastically scale cloud resources, and a reconfigurable hardware acceleration fabric for network and remote memory accesses.

We design and build the full end-to-end HiveMind system on two real edge swarms comprised of drones and robotic cars. We quantify the opportunities and challenges serverless introduces to edge applications, as well as the trade-offs between centralized and distributed coordination. We show that HiveMind achieves significantly better performance predictability and battery efficiency compared to existing centralized and decentralized platforms, while also incurring lower network traffic. Using both real systems and a validated simulator we show that HiveMind can scale to thousands of edge devices without sacrificing performance or efficiency, demonstrating that centralized platforms can be both scalable and performant.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527407

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Embedded and cyber-physical systems**; • **Hardware** → *Networking hardware*; • **Networks** → *Network components.*

KEYWORDS

serverless, edge swarms, IoT, cloud computing, hardware acceleration, domain specific language

ACM Reference Format:
Liam Patterson, David Pigorovsky, Brian Dempsey, Nikita Lazarev, Aditya Shah, Clara Steinhoff, Ariana Bruno, Justin Hu, and Christina Delimitrou. 2022. HiveMind: A Hardware-Software System Stack for Serverless Edge Swarms. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3470496.3527407

1 INTRODUCTION

Swarms of edge devices are increasing in number and size [13, 17, 26, 31–33, 35, 36, 43, 44, 59, 61, 62, 62, 72, 75, 98, 110, 116, 119, 123, 125]. Swarms enable new applications, often with intermittent activity [58, 70, 71, 90, 91, 96, 116, 119, 127], spanning accounting for people in disaster zones, to monitoring crops, and navigating self-driving vehicles. The devices themselves have low-power, modest resources, and are prone to unreliable network connections.

As swarm sizes increase, designing a hardware-software system stack that enables programmable and performant operation for resources that span cloud and edge devices becomes a pressing need. Prior work has explored both centralized [26, 33, 119] and distributed [36, 46, 58, 90, 123] approaches. In centralized systems all control, i.e., decision making on task allocation, as well as task execution happens in a backend cloud infrastructure. In distributed settings each edge device is mostly-autonomous, i.e., selects tasks to execute, and executes them locally, only transferring its output to a backend system. Neither approach is optimal. Centralized systems, while they enjoy global visibility into the swarm's state and can leverage cloud resources, quickly hit scalability bottlenecks with more edge devices. Distributed systems, on the other hand, scale

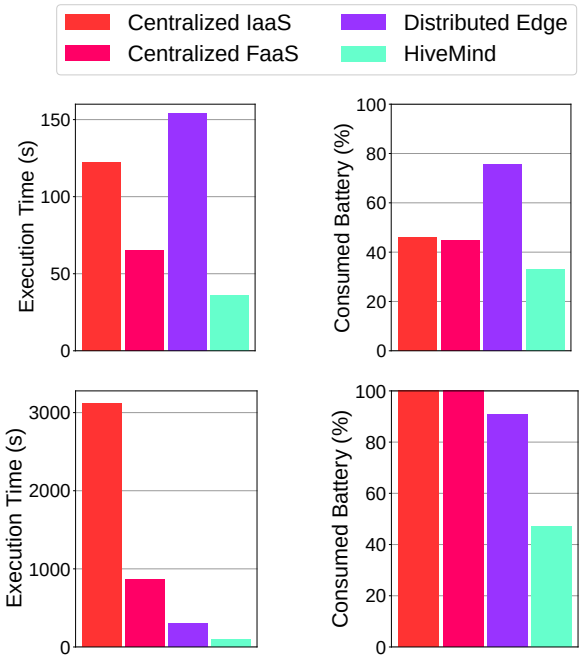

Figure 1: Execution time and consumed battery for a treasure hunt scenario where drones try to locate objects in a field for a (top) real 16- and (bottom) simulated 1000-drone swarms.

better, but are hindered by the lack of coordination between devices, especially when there is redundant computation, or computation that would benefit from swarm-wide learning.

We present *HiveMind*, the first hardware-software system stack for swarm coordination that effectively bridges the gap between centralized and distributed coordination. HiveMind focuses on performance predictability, resource efficiency, and programmability. It relies on three key design components. First, it proposes a high-level declarative programming model for users to express the task graph of their applications, abstracting away the complexity of explicitly managing cloud and edge resources. It then uses program synthesis to explore different task placements strategies between cloud and edge, transparently to the user. Second, HiveMind uses a centralized, cloud-residing controller with global visibility into the swarm's state and available resources. To make centralized coordination scalable, HiveMind leverages serverless computing [6, 7, 12, 14, 19, 20, 77], which ties well with the intermittent activity and fine-grained parallelism of edge tasks, which do not warrant long-term resource reservations. It additionally automates task scheduling and straggler mitigation, lowering the bar for porting new applications to cloud-edge platforms. Third, at the hardware level, HiveMind proposes a reconfigurable, FPGA-based acceleration fabric for remote memory access and networking between edge and cloud resources and within cloud servers. By revisiting the entire system stack for edge swarms, HiveMind achieves the best of centralized and distributed coordination, while removing the burden of managing cloud-edge resources from the user.

We build the entire end-to-end HiveMind system on a real swarm with 16 drones with a 12-machine backend server cluster. We also show HiveMind's generality in terms of edge devices by also porting it on a swarm of 14 terrestrial robots. We implement a benchmark suite comprised of a wide spectrum of edge applications, such as SLAM, image recognition, and weather analytics, as well as end-to-end multi-phase scenarios, locating stationary items in an area, and identifying the number of unique people in a field. We quantify the implications of serverless for IoT applications, as well as the trade-offs between centralized and distributed execution. Fig. 1 shows the performance and energy efficiency of HiveMind for a representative end-to-end scenario, compared to fully centralized and fully distributed systems. For the centralized system we show a setting that uses serverless (FaaS) and one that uses statically provisioned cloud resources of equal cost (IaaS). The scenario involves the drones trying to locate a set of tennis balls randomly placed in a sports field. We show results for the real 16 drones and a simulated 1000-drone swarm, using a new, validated simulator. In all cases, HiveMind greatly outperforms the other systems, both in terms of performance and energy efficiency. The difference is more dramatic for larger swarms, where centralized systems hit scalability bottlenecks. These results are consistent across all examined applications. Finally, we show that each software-hardware component in HiveMind is essential in achieving these benefits, although its mechanisms can be used independently as well, e.g., in the event where FPGA acceleration is not available in a cloud.

2 SWARM COORDINATION DESIGN TRADE-OFFS

We first quantify the trade-offs between centralized and distributed approaches in swarm coordination, to identify overheads that can be accelerated in hardware, and programmability bottlenecks that can be addressed through better software interfaces. All experiments are done end-to-end, on a real drone swarm, with a server cluster as the cloud backend.

2.1 Methodology

Drones: We use a swarm of 16 programmable Parrot AR. Drones 2.0 [21], each equipped with an ARM 32-bit Cortex A8 1GHz processor running Linux 2.6.32. There are 2GB of on-board RAM, complemented with a 32GB USB flash drive. Each drone has a vertical 720p front-camera for obstacle avoidance, and the following sensors: gyroscope, accelerometer, thermometer, magnetometer, hygrometer, and altitude ultrasound sensor. We additionally fit an 8MP camera to each drone's underside over USB, for high definition photos. Unless otherwise specified, drones collect 8 frames per second, at 2MB per frame. Drones fly at a height of 4-6m and move at 4m/s. Their camera has a 92field of view (FoV), with an approximate coverage of $6.7m \times 8.75m$ per frame.

Cluster: We use a dedicated cluster with 12, 2-socket, 40-core Intel servers with 128-256GB of RAM each, running Ubuntu 18.04. Each server is connected to a 40Gbps ToR switch over 10Gbe NICs. Servers communicate with the swarm with two 867Mbps LinkSys AC2200 MU-MIMO wireless routers [15].

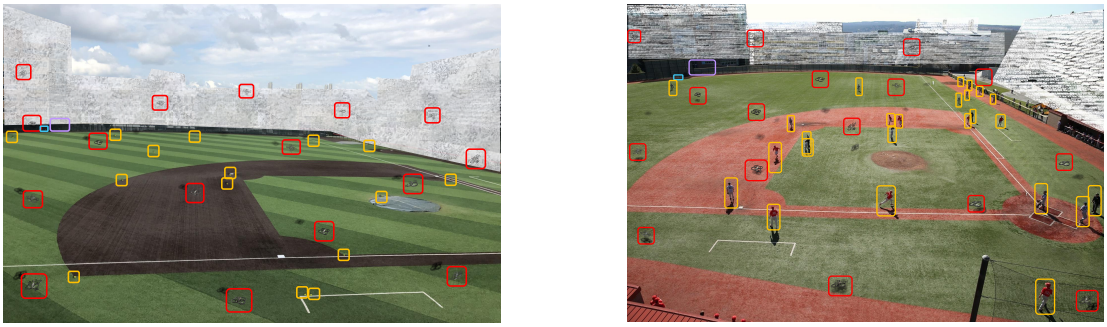

Figure 2: The drone swarm executing (a) the first scenario where the number and location of a set of static items (tennis balls) is determined, and (b) the second scenario, where the number of unique people in a field is determined.

Single-phase applications: We design and implement a benchmark suite of diverse applications, which process sensor data collected on the drones. We select both resource intensive applications better suited for cloud resources, and more lightweight services that edge devices can accommodate. These include *S*1: face recognition (identify human faces using FaceNet [112]), *S*2: tree recognition (identify trees using a CNN from TensorFlow's Model Zoo [25, 29]), *S*3: drone detection (detect other drones using an SVM classifier trained for the orange tag all our drones have [5]), *S*4: obstacle avoidance (detect obstacles in the drone's vicinity and adjusts course to avoid them, using the obstacle detection framework in `ardrone-autonomy` [5]), *S*5: people deduplication (disambiguate between faces using FaceNet [112]), *S*6: maze (navigate through a walled maze using the Wall Follower algorithm [23, 52]), *S*7: weather analytics (weather prediction based on temperature and humidity levels in sensor data), *S*8: soil analytics (estimation of soil hydration from images and humidity sensor), *S*9: text recognition (image to text conversion of signs), and finally *S*10: simultaneous localization and mapping (SLAM, using image and sensor data) [4]. We evaluate one service at a time to eliminate interference, however, the platform supports multi-tenancy.

Multi-phase scenarios: In addition to the previous applications, we also build two end-to-end scenarios, each with multiple phases of computation and I/O, to examine more representative use cases for drone swarms, shown in Fig. 2.

- **Scenario A – Stationary Items:** The swarm is tasked with locating 15 tennis balls placed in a baseball field. At time zero, the field is divided equally among the drones. Routes within each region are derived using A* [48], where each drone tries to minimize the total distance traveled. In addition to collecting images, each drone also runs an on-board obstacle avoidance engine, based on the SVM classifier in `ardrone-autonomy` [5, 10] trained on trees, people, drones, and buildings. Obstacle avoidance always runs on-board to avoid catastrophic failures due to long network delays with the cloud. We have ensured that it does not have a large impact on power consumption.
- **Scenario B – Moving People:** The swarm needs to recognize and count a total of 25 people on a baseball field; the number of people is not known to the system. People are allowed to move within the field, therefore the same person may be photographed by multiple drones, requiring disambiguation. We implement two versions of the person recognition model based on the Tensorflow Detection Model Zoo [8, 25, 29], one that can run on a serverless framework, and a native implementation for the edge. People disambiguation is based on the FaceNet [112] face recognition framework, which uses a CNN to learn a mapping between faces and a compact Euclidean space, where distances correspond to an indication of face similarity.

2.2 Network Overheads

We first examine a setting where all computation happens in the cloud, leading to network congestion. Fig. 3a shows the fraction of end-to-end latency corresponding to network processing, task execution, and management operations (task instantiation, scheduling, etc.) across the ten single-tier jobs and two end-to-end scenarios. Execution includes computation and data sharing between functions. Non-shaded bars show median latency, and shaded bars tail latency (99^{th} pctl). Across all jobs, networking accounts for at least 22% of median latency (33% on average), and a higher fraction of tail latency. This is even more pronounced for the end-to-end scenarios, which involve multiple phases of computation and communication between the drones and cluster. Services are not running at max load here, so the network links are not oversubscribed. We now examine the system's scalability as network load increases. Fig. 3b shows the bandwidth and tail latency for Face Recognition, as the number of drones collecting images of higher resolution increases. Tail latency remains low for fewer than 4 drones, even for max resolution (8MP). As the number of drones increases, the network saturates, and latency increases dramatically. While the max resolution is not always required, offloading all data to the cloud limits the number of devices the framework can reliably support, or requires acceleration of networking processing to accommodate larger swarms.

2.3 Centralized versus Distributed Execution

Finally, we examine the trade-offs between fully centralized execution, where all computation and data aggregation happens in the server cluster, compared to a fully decentralized environment,

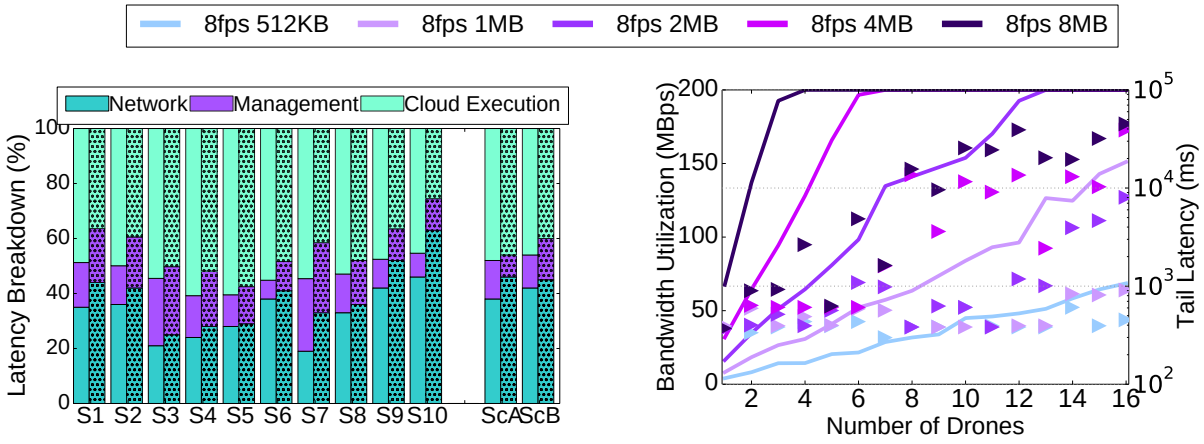

Figure 3: (a) Latency breakdown to network, management operations (scheduling, instantiation, etc.), and cloud execution across all single-tier tasks and multi-tier scenarios. Non-shaded bars show the median latency, and shaded bars the tail latency (99^{th} percentile). Fig. (b) network bandwidth usage for the face recognition tasks ($S1$).

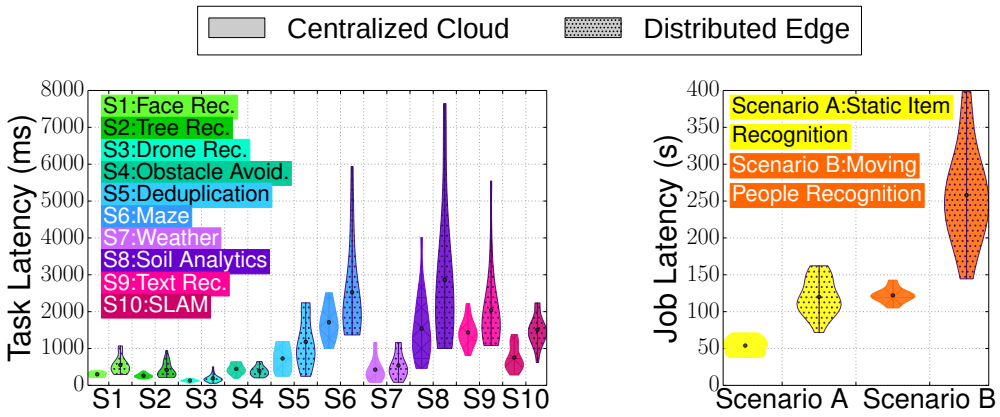

Figure 4: Distribution of task latency execution across the (a) ten single-tier jobs and the (b) two end-to-end scenarios for centralized cloud execution (non shaded plots) and distributed edge execution (shaded plots). Each violin plot shows the PDF of task latencies for that application.

where computation happens on the edge devices, and only the final outputs are transmitted to the cloud.

To control the placement and instantiation of short-lived tasks in the backend cloud we use Apache OpenWhisk [20], a widely-used open-source serverless framework, and the backbone of IBM Cloud Functions [68, 70, 71, 82, 114]. OpenWhisk instantiates functions in Docker containers, via an HTTP request to an NGINX front-end, which triggers the OpenWhisk Controller to check a database [2] for authentication, and select an Invoker to instantiate the function. Applications are in a language OpenWhisk supports (Python, node.js, Ruby, PHP, Scala, Java, Go). While the trade-offs between centralized and distributed execution have been explored for traditional cloud environments [63, 90, 91, 115], the opportunities and challenges serverless introduces impact prior findings. We discuss these implications in more detail in Section 3. Each of the ten jobs runs for 120s, repeated 10 times, and each end-to-end scenario runs to completion, repeated 50 times.

Fig. 4a shows the task latency distribution across the ten single-tier jobs, and Fig. 4b across the two end-to-end scenarios. For most jobs, centralized execution achieves better and more predictable performance than on-board execution, despite the increased overheads from offloading data to the cloud. The difference stems both from the higher compute and memory capabilities of server nodes, and from the higher concurrency serverless can exploit compared to on-board execution.

The three exceptions are drone detection ($S3$) and weather analytics ($S7$), which behave comparably on the cloud and edge due to their modest resource needs, and obstacle avoidance ($S4$), which achieves better performance at the edge, by avoiding data transfers, and by adjusting its route in-place, instead of waiting for the cluster to update its route if there is an obstacle. The results are similar for the two scenarios, and more pronounced for the more computationally-intensive Scenario B. Despite the performance advantage of the cloud, it also greatly increases network traffic, hurting scalability.

Beyond the performance comparison, on-board execution quickly drains the drones' battery, leaving the second scenario incomplete due to several drones running out of power.

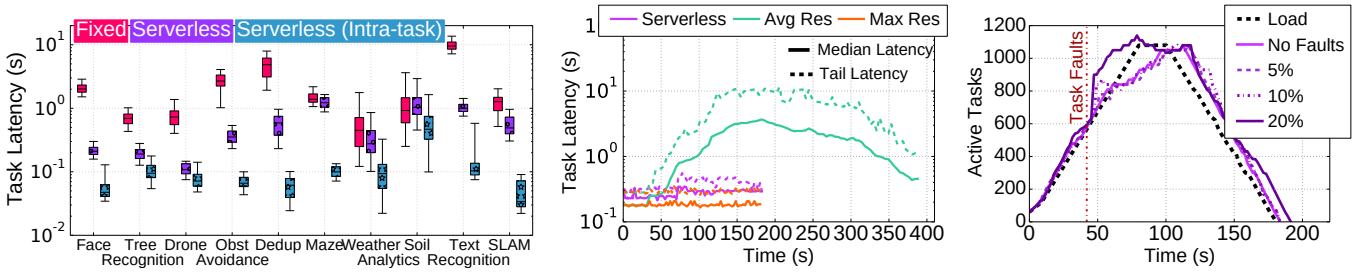

Figure 5: The opportunities of serverless for edge jobs: (a) Task latency with fixed and serverless deployments, (b) Latency for Face Recognition with a fluctuating load, and (c) Number of active tasks when a fraction of functions fail.

3 THE ROLE OF SERVERLESS IN EDGE SWARMS

From the previous study, it is clear that centralized control is superior in terms of performance and resource efficiency, however, it suffers from high network overheads and scalability bottlenecks. To address in particular the second of these challenges, we leverage serverless computing, a new event-driven programming framework for services with fine-grained parallelism and intermittent activity. We first quantify the opportunities serverless offers as the programming model of the cloud backend, and then discuss its challenges.

Since in this section we are exclusively interested in the trade-offs between serverless and traditional IaaS/PaaS deployments for edge applications, we consider performance metrics within the bounds of the cloud system, i.e., latency is measured from the moment sensor data arrive to the cloud until before the response is sent to the drones. This isolates any overheads that serverless may add from the impact of network congestion between edge and cloud.

3.1 Serverless Background

Serverless or Function-as-a-Service (FaaS) computing has recently emerged as a cost-efficient alternative for applications with high data-level parallelism, intermittent activity, fluctuating load, and mostly stateless operation, for which maintaining long-running reserved resources is inefficient [6, 7, 14, 20, 30, 38, 67, 73, 76, 77, 82, 85, 93, 95, 101, 114, 122]. Serverless functions are instantiated in short-lived containers or VMs, last up to a few minutes, and containers are terminated shortly after the process completes, freeing up resources. Users are charged on a per-request basis, depending on the amount of CPU and memory time allocated.

Serverless gives cloud providers better visibility into application characteristics, and reduces cloud overprovisioning [39, 40, 53, 55, 56, 94, 109], as users are no longer responsible for explicitly reserving allocated resources.

Functions typically take less than a second to spawn, and most providers allow the user to spawn thousands of concurrent functions. Functions are instantiated in containers or VMs but are not guaranteed a specific machine type, and are scheduled by the provider to improve utilization. Serverless lowers the entry bar for new applications, but also opens them up to unpredictable performance due to function interference. Current providers only offer service level agreements (SLAs) in terms of availability, not performance, as the provider does not have visibility into application-level performance metrics.

Internet of Things (IoT) applications are particularly well suited for serverless, given their intermittent nature, and the modest amount of on-board resources, which motivates cloud offloading [6, 7, 12, 14, 19, 20, 74, 77, 103, 104, 108, 111, 120]. Serverless also introduces caveats for edge applications, especially when they have to meet quality of service (QoS) requirements, the violation of which can cause catastrophic failures of edge equipment. While prior work has studied the architectural implications of serverless for cloud jobs [114], its implications for edge services are not clear.

3.2 The Opportunities of Serverless

Concurrency: Fig. 5a shows the task latency distribution across all ten applications under constant load, when using a fixed amount of cloud resources in a containerized (non-serverless) datacenter, and when using serverless without and with intra-task parallelism. An example of a task is the process of recognizing a human face in a frame batch of one second. In serverless, each task can instantiate a single function, or leverage intra-task parallelism to further improve performance. For fairness, the total amount of CPU time is the same for all deployments. The boundaries of the box plots show the 25^{th} and 75^{th} percentiles, the horizontal line shows the median latency, and the whiskers show 5^{th} and 95^{th} percentiles.

Even without intra-task parallelism, serverless is almost always an order of magnitude faster than the fixed allocation. This is not surprising, given that serverless takes advantage of parallelism across tasks without being limited in the number of cores it can occupy (except if the user limit is reached; 1,000 functions by default on AWS Lambda [6]). The maze traversal, and the weather and soil analytics do not significantly benefit from fine-grained parallelism. For the maze traversal, the tasks per second are lower than for the other jobs, as drones move slowly in the maze, hence the benefit from task concurrency is diminished, and for weather analytics, the amount of sensor data is modest and the computation is lightweight, so even the constrained fixed resources do not become oversubscribed.

Enabling intra-task parallelism further improves performance. For jobs like image-to-text recognition and SLAM, the improvement is dramatic, as they have ample parallelism, and are CPU- and memory-intensive.

This shows that as long as the edge application has ample parallelism (data-, task-, or request-level), serverless benefits performance. There are two caveats to this: first, the latency variability is typically higher in serverless, due to interference between functions

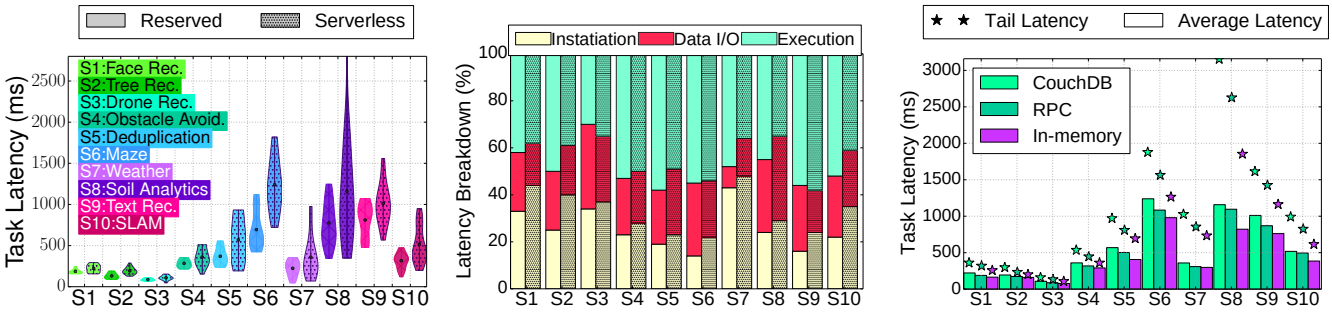

Figure 6: The challenges of serverless for edge applications. (a) Performance variability across task executions, (b) Impact of instantiation and data sharing on task latency, and (c) Impact of data sharing protocol on task latency. *S1-S10* correspond to the ten applications, and *ScA*,*ScB* correspond to the two end-to-end scenarios, as shown in Fig. 6b.

sharing a physical node, and due to the instantiation and management overheads introduced by OpenWhisk. Second, parallelism does not come for free, as the user or cloud provider need to determine the available parallelism of a given job; distributing work and aggregating results also incurs overheads from data sharing and synchronization. Despite OpenWhisk introducing millisecond-level overheads for instantiating a container, traditional PaaS/IaaS clouds introduce several seconds of overheads to spin up new instances.

Elasticity: Fig. 5b shows the task latency for Face Recognition (*S1*), under fluctuating load. First only one drone sends images at low rate, and progressively more drones transfer images of higher frames-per-second (fps) to the cloud. Eventually, the load decreases down to a single drone. We compare serverless to two fixed deployments; one provisioned for the average and one for the worst-case load. While serverless closely follows the load, the average-load deployment quickly becomes saturated, hurting latency. The max-load deployment also follows the load, but greatly underutilizes resources [53–57, 94, 109]. Given the intermittent activity of edge devices, statically provisioning cloud resources is inefficient.

Fault tolerance: Fig. 5c shows the number of tasks over time for the same fluctuating load scenario, when a number of functions fails during execution. Even for 20% failed tasks, OpenWhisk is able to hide the increased workload, by quickly respawing tasks on new cores before they degrade the end-to-end execution time. This is important for edge services, as faulty or missing sensor data can cause tasks to fail.

3.3 The Challenges of Serverless

Despite its benefits, serverless introduces several challenges.

Performance predictability: Fig. 6a highlights the higher performance variability serverless exhibits compared to reserved cloud resources. For all ten jobs, we show violin plots of the task latency distribution on reserved and serverless deployments. Each application runs at modest load to avoid overloading the reserved resources. Latency variability is consistently higher with serverless; a similar difference is less pronounced in the reserved resources. This is due to the instantiation overheads of serverless, which are more evident under low load when OpenWhisk terminates unused containers, the impact of OpenWhisk's scheduler when determining task placement, and the overhead of data sharing between dependent functions. OpenWhisk – and most commercial serverless platforms – does not permit direct communication between functions, using instead a database (CouchDB) or persistent storage for intermediate data.

Instantiation overheads: Fig. 6b shows the latency breakdown in serverless in terms of instantiation overheads, data sharing between functions, and useful computation. Measurements are obtained by instrumenting the OpenWhisk controller and Docker containers. Non-shaded bars show median, and shaded bars tail latency. Instantiating containers takes on average 22% of median and 29% of tail latency. For the short-running tasks of weather analytics that fraction exceeds 40%, while for the more computationally-intensive maze traversal, it falls below 20%. In all cases, it is a substantial factor to task latency, and a major contributor to tail latency. Given that edge tasks are usually short-lived, a system should minimize instantiation overheads without sacrificing resource efficiency.

Function communication: Fig. 6c delves deeper into the overheads associated with data exchange between dependent functions. By default in OpenWhisk, and other serverless frameworks [6, 12, 18, 19], functions do not directly communicate with each other, i.e., a child function does not know the location of its parent. In OpenWhisk the "third party" used for communication is a CouchDB instance. In AWS Lambda, functions communicate through S3, AWS's remote persistent storage fabric. The reason for this communication model is that, under serverless, a function should be able to run anywhere in the datacenter, transparently to the user. On the other hand, disallowing direct function communication introduces non-negligible overheads, especially in latency-critical services. Fig. 6d compares task latency with the default CouchDB system in OpenWhisk, to direct RPC communication, and in-memory computation, where a child function is always placed on the same live container as its parent. CouchDB experiences the highest latency, especially for tail latency. This is not surprising, given that for two functions to exchange data they have to go through the OpenWhisk controller to get a handle to a database object. Direct RPC communication is considerably faster, but it is still outperformed by in-memory communication which does not involve any data exchange.

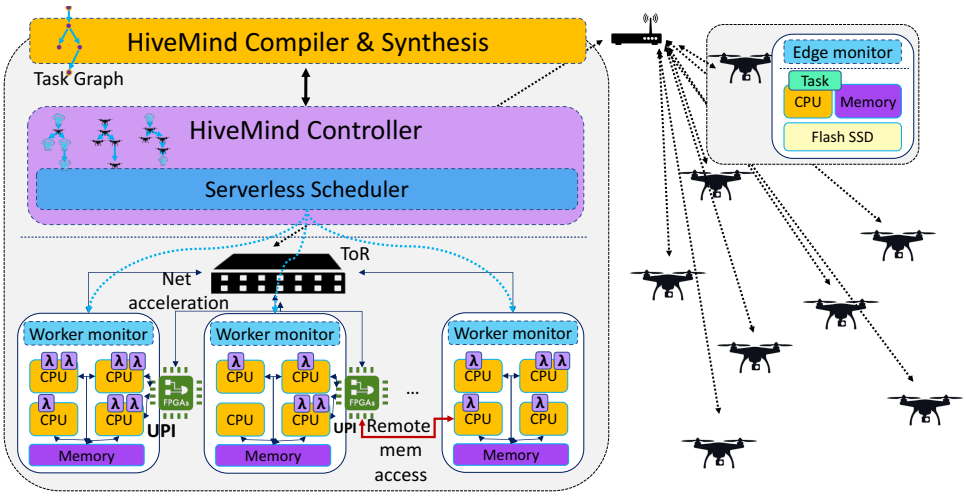

Figure 7: HiveMind platform overview. The compiler and synthesis toolchain in HiveMind start from a user's high-level specification of a job's task graph to generate the code for the individual tasks and cross-task APIs, and determine whether tasks should execute at the edge or backend cloud. HiveMind's scheduler instantiates serverless functions on shared nodes and allocates resources appropriately, and the hardware acceleration fabrics in HiveMind improve performance for RPC processing and remote memory access.

This analysis shows that while serverless has the potential to enable scalable backend computation for swarm coordination, several challenges need to be addressed, including providing a high-level programming interface for users to express their computation without needing to manage low level deployment details, like task scheduling and placement, concurrency configuration, function instantiation, and data sharing.

4 HIVEMIND DESIGN

Based on the analysis discussed above we design HiveMind, a scalable software-hardware system stack for the coordination of edge swarms that is programmable, performant, and resource-efficient. HiveMind follows a vertical design that includes a domain specific language and code synthesis toolchain, a centralized controller for a serverless cloud, and a reconfigurable acceleration fabric for remote memory access and networking. HiveMind abstracts away as much of the complexity of operating a cloud-edge system as possible from the user, without sacrificing performance or efficiency.

To this end, HiveMind makes the following five key contributions: (1) a declarative programming model that allows users to express the high-level structure of their computation without being exposed to the complexities of deployment; (2) a centralized controller that automatically determines what computation should be placed in the cloud versus edge resources; (3) a scheduler for the serverless cloud that handles task placement and function instantiation; (4) a fast remote memory access accelerator that enables data exchange between dependent functions; and finally (5) a reconfigurable networking acceleration fabric that reduces the communication overheads between cloud and edge.

Fig. 7 shows an overview of the HiveMind platform. Below we discuss each key system component in more detail.

4.1 HiveMind DSL and Programming Model

Exposing all complexity associated with composing and deploying an application on a cloud-edge system to the user hinders the adoption of these platforms, or introduces performance and/or efficiency losses due to programmer faults [1, 9, 11, 16]. Instead, HiveMind follows the approach of domain specific languages (DSL) [86, 87, 106], which abstract away most of the system complexity, allowing the user to focus on the high level objectives of their computation, instead of the low-level system details needed to achieve them.

While prior work has explored task-level programming frameworks for the edge [24, 126], these systems primarily focus on querying data collected on edge devices, and/or launching single-phase, specialized computation that consumes sensor data. HiveMind instead implements an Domain Specific Language (DSL) that targets complex, multi-phase jobs, and automates the synthesis of cross-task APIs, data sharing, and task placement. This aids programmability, as developers do not have to handle the complexity of defining APIs between tasks, which is a major source of bugs in such systems [69, 71, 88], especially since the supported APIs vary across edge devices, and the devices themselves can have different ISAs, which complicates work partitioning. Public bug reports in multi-tier cloud and edge applications show that incorrect or inefficient API definition is one of the primary sources behind unpredictable performance, failures, unreachable services, or resource inefficiency [1, 9, 11, 16, 69].

The HiveMind DSL exposes a declarative programming interface in Python for users to express a high-level description of their task graph, and import the logic associated with different execution steps, similar to PyFlow [22], but significantly augmented to support the cross-task dependencies of edge applications. Listing 1 shows a subset of the DSL's operations to define tasks and task

```
Task(name,dataIn,dataOut,code,taskArgs)
Definition of task, with
i/o data, link to code
path & optional arguments

TaskGraph(edgeList,constraints)
List of tasks in the
application's control flow
and perf/cost constraints

Parallel(taski,taskj)
List of tasks a task can
execute in parallel to

Overlap(taski,taskj)
List of tasks a task can
overlap with

Serial(taski,taskj)
List of tasks that cannot
overlap

Synchronize(task,condition)
Synchronization point
across tasks
```

Listing 1: Subset of HiveMind's DSL operations.

```
Schedule(task)
Sched. constraints &
task priorities

Isolate(task)
The tasks requiring
dedicated containers

Place(task)
Fix task placement
(edge or cloud)

Restore(task)
Fault tolerance policy

Learn(task)
On/Off online learning
(one device vs. swarm)

Persist(task)
Store task's output
in persistent storage
```

Listing 2: Optional management directives.

graphs, as well as to denote timing and execution dependencies between tasks. Users can specify whether tasks are allowed to run in parallel, can partially overlap, or need to execute serially. Listing 2 also shows some of HiveMind's optional management directives, which users can leverage to specify scheduling constraints for specific tasks, fault tolerance policies in the event of a device failure, data persistence requirements for the output for certain tasks, as well as to enable or disable the retraining of any ML models during application execution. The two listings only show a subset of HiveMind's language; we omit data structures in the interest of space; there is support for both individual objects and data streams. In addition to expressing the control flow of their application, users also specify the performance metrics their application must meet, in terms of execution time, latency, and/or throughput. An upper limit in terms of cost for cloud resources can also be expressed. These metrics are used by HiveMind to determine how to partition computation between cloud and edge resources.

Listing 3 shows a simplified version of the task definitions for Scenario B, where unique people are counted in a field.

Once a user expresses their application's task graph, the HiveMind compiler and program synthesis tool compose the end-to-end application, including automatically synthesizing the required APIs for data communication between computational steps (hereafter referred to as tiers). HiveMind generates two types of cross-tier APIs; one based on RPCs using Apache Thrift [3] for computation that may run at the edge, and another using OpenWhisk's function interface for tasks running on the serverless cluster [2, 20]. The former generates code in C++ for the cross-task APIs, while the latter uses CouchDB's communication protocol. In Sec. 4.4 we replace OpenWhisk's default protocol with a hardware acceleration fabric for remote memory access. As the number of phases in a job increases, the number of APIs HiveMind needs to generate also increases. This process is entirely automated, and only happens once, at initialization. Once APIs are composed, HiveMind selects appropriate task mappings at runtime. To reduce the number of generated scenarios, HiveMind accepts optional hints from users regarding tasks that *need* to run either at the cloud or edge, due to hardware specs, security reasons.

```
TaskGraph(list=['createRoute','collectImage',
'obstacleAvoid','faceRecognition',
'deduplication'],constraint=[execTime='10s'])

Task(createRoute,inputMap,outputRoute,
'filepath/to/task/code',
load_balancer='round_robin',
parentTask=None,childTask=['collectImage'])

Task(collectImage,None,sensorData,
'filepath/to/task/code',
speed='4',resolution='1024p',
colorFormat='color',
parentTask=['createRoute'],childTask=
['obstacleAvoidance','faceRecognition'])

Task(obstacleAvoidance,sensorData,adjustRoute,
'filepath/to/task/code',
algorithm='slam',
parentTask=['collectImage']],childTask=[])

Task(faceRecognition,sensorData,recognitionStats,
'filepath/to/task/code',
trainingData='zoo',
algorithm='tensorflow_zoo',
parentTask=['collectImage']],
childTask=['deduplication'])
Task(deduplication,recognitionStats,dedupList,
'filepath/to/task/code',
sync='all',
parentTask=['faceRecognition']],
childTask=[])

Parallel(obstacleAvoidance,faceRecognition)
Serial(faceRecognition,deduplication)
Learn(faceRecognition,'Global')
Place(obstacleAvoidance,'Edge:all')
Persist(faceRecognition)
Persist(deduplication)
```

Listing 3: Example HiveMind application for People Recognition and Deduplication.

4.2 Hybrid Execution Model

Sec. 2.3 showed that offloading all computation to the cloud causes network congestion, limiting scalability. On the other hand, running all tasks at the edge quickly depletes the device's battery, and is prone to unpredictable performance.

Partitioning work between cloud and edge resources is a challenging, long-standing problem [46, 74, 90, 91, 115, 119]. There has been extensive work on offloading computation to a backend cloud, either by manually tagging tasks to run in the cloud, or by automating the offloading process, especially for mobile devices [46, 49, 107, 119]. In the context of an edge swarm, partitioning work manually is problematic for several reasons; first, users do not have a good assessment of the performance and power consumption of different partitioning strategies, unless they can profile the application in detail. Even then, edge applications are prone to load fluctuations and failures, which can affect previous findings. Second, changing where computation runs affects the software infrastructure needed; for example, in our drone swarm, edge devices communicate with the cloud using RPCs, over TCP/IP, while cloud serverless functions communicate over OpenWhisk's CouchDB interface. Third, the use of serverless in the backend cloud changes previous trade-offs, as its instantiation overheads are higher compared to traditional cloud resources, and conversely, the ability of serverless to leverage fine-grained parallelism is more pronounced compared to long-term resource reservations. Additionally, prior work on automating cloud offloads [46, 49, 107] primarily focuses on mobile devices, e.g., phones, which have more powerful resources than typical UAVs, do not have to also manage flight autonomy,

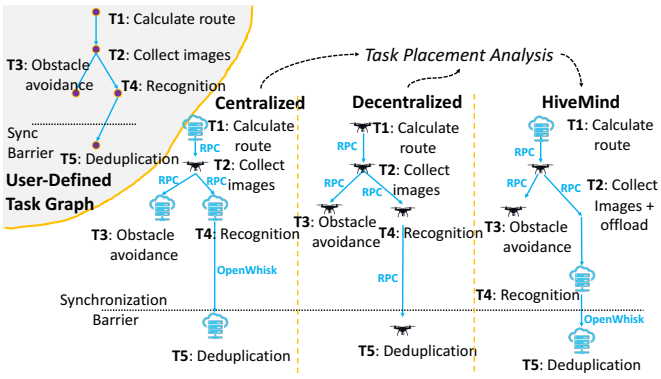

Figure 8: Hivemind's exploration process for task placement.

and mostly optimize for power efficiency instead of performance predictability for multi-tier jobs. Similarly, prior work on offloading work to serverless is mostly limited to using serverless for overflow load [73], and still needs to address the challenges of Sec. 3.3.

Fig. 8 shows an example of this exploration for the second end-to-end scenario of Sec. 2.1. HiveMind's program synthesis then creates all – meaningful – execution models, where part or all of the computation is placed on the edge devices, following any constraints provided by the user about where specific tasks should run. For a simple, 2-tier task graph ($A \rightarrow B$), HiveMind would compose the APIs for a total of 4 end-to-end scenarios ($A_{cloud} \rightarrow B_{cloud}$, $A_{edge} \rightarrow B_{cloud}$, $A_{cloud} \rightarrow B_{edge}$, $A_{edge} \rightarrow B_{edge}$).

Requiring the scenario to be meaningful reduces the search space by discarding execution models that would not make sense practically, e.g., collecting sensor data in the cloud. For each remaining execution model, HiveMind creates the required communication APIs, either using an RPC framework for cloud-edge communication, or using OpenWhisk's API for intra-cloud communication, and profiles the application on the target swarm. The performance and power results are presented to the user, who selects the initial work partitioning scheme that satisfies their performance and/or efficiency constraints. A user can specify constraints in terms of performance, power, cost, or a combination of these metrics. At runtime, HiveMind can change its task mapping if the user-provided goals are not met. Changes to task placement currently only happen at task granularity, i.e., HiveMind would not migrate a single, partially-completed task between cloud and edge at runtime.

To maintain global visibility into all resources, HiveMind uses a centralized Controller, residing in the cluster. The controller consists of a load balancer, which partitions the available work across all devices, an interface to the scheduler responsible for serverless function placement, an interface to communicate to the edge devices, and a monitoring system that collects tracing information from the cloud and edge resources.

4.3 Serverless Cloud Scheduler

We implement our scheduler directly in OpenWhisk's centralized controller, which is responsible for finding an appropriate Invoker to launch a function, and passing it the function information via Kafka's publish-subscribe messaging model. The Invoker then instantiates the function in a container [20]. While prior work has improved on native serverless schedulers to either improve resource efficiency or performance and fairness [66, 67, 73, 81, 101], this work primarily targets cloud-only applications, and applications with a single or a few computation and I/O stages.

Users can specify scheduling constraints through HiveMind's DSL, as discussed in Sec. 4.1. If scheduling constraints are specified, HiveMind's scheduler follows them, assuming they do not conflict with each other, and there are available resources in the cluster. If no scheduling constraints are specified, HiveMind places functions to optimize performance. For each server in the cluster, HiveMind deploys a *worker monitor*; a lightweight process that periodically monitors the performance of active functions, and the server's utilization. Using these monitors, the scheduler identifies nodes with sufficient resources to host new functions.

The scheduler performs two optimizations: first, in multi-tier jobs where consecutive tiers are hosted on serverless, the scheduler tries to place child functions in the same container as their parent, to avoid the costly data exchange through CouchDB. Prior work has explored co-scheduling dependent functions on the same physical node to leverage fast memory-based communication [81]. This is not always possible, either because a server is overloaded, or because the child requires different software dependencies than the parent. The child may also be spawned with some delay, at which point the parent's container has been terminated. For such cases, HiveMind implements a new remote memory protocol that allows fast in-memory data exchange between functions (Sec. 4.4). In general, in serverless, once a parent task invokes a child, it terminates its own function, and OpenWhisk destroys the parent's container. In cases where descendant tasks can be placed on the container their parent used, HiveMind also places the parent's output data in a virtual memory region visible by the child, similarly to prior work [60, 81, 117].

The scheduler's second optimization is to reduce instantiation overheads. In line with similar optimizations on public clouds [6, 7, 14], HiveMind does not immediately terminate an idling container, for the event where a new function arrives in the near future that can use it. If a new function does arrive, HiveMind schedules it in that container. If not, it terminates it. The amount of time an idling container remains alive is empirically set; it ranges between 10 and 30 seconds, given that serverless containers are instantiated much faster than traditional IaaS and PaaS containers.

Finally, to avoid interference between functions, two containers can share a physical server, but never share a logical core. HiveMind pins containers to cores to avoid interference from the OS scheduler's decisions. For our examined applications memory contention was never an issue; however, cache partitioning and memory bandwidth partitioning can also be integrated in HiveMind for performance and security isolation.

In Sec. 5.6 we study the centralized scheduler's scalability. While the system scales to many edge devices, as with any centralized system, it can become a bottleneck. In that case, HiveMind uses multiple schedulers, each responsible for a subset of tasks, but *with global visibility into all cloud and edge resources*. Such shared state cluster managers have demonstrated scalability without hurting decision quality in cloud environments [57, 113].

4.4 Fast Remote Memory Access

As discussed above, placing a child function in the same container as its parent is not always possible. In those cases, OpenWhisk's default data exchange involves requests to the Controller and to CouchDB, where the results of previous functions are stored. This is expensive, especially when many functions try to access data concurrently [42, 83, 84, 100, 118].

Instead, HiveMind uses a hardware-accelerated platform based on an Intel Broadwell FPGA-enabled architecture. The host is an Intel Xeon E5-2600 v4 CPU, integrated with an Arria 10 GX1150 FPGA over a UPI bus (memory interconnect). Incoming requests are processed directly by the FPGA, following an RDMA over Converged Ethernet (RoCE)-style protocol, eliminating the need to copy data between application memory and the data buffers in the OS, and are transferred to the CPU running the function's container via the memory interconnect. The physical placement of parent and child functions is known by the centralized controller. While RDMA protocols have been previously used for disaggregating resources like memory [47, 121, 124], HiveMind is the first implementation of an FPGA-based remote memory acceleration fabric for fast communication between serverless functions.

The FPGA-based implementation significantly improves performance by bypassing the host's network stack, and the tight integration between the host and FPGA avoids the overheads of the PCIe interfaces [41, 65, 79, 83, 89, 99, 100]. HiveMind directly leverages the processor's cache coherence protocol to handle dirty data tracking and demand paging with no software involvement, thus reducing the remote memory access overhead. While remote memory access deviates from serverless's default policy of disallowing direct communication between dependent functions, it does not break the serverless abstraction that a function can run anywhere in the cluster transparently to the user. A child function need not know the physical location of its parent (and vice versa). The child simply sees a virtualized object location for the parent's output, with address mapping handled by the FPGA.

4.5 Hardware-Based Networking Acceleration

Sec. 2 showed that congested networks can have a dramatic impact on performance and power efficiency; similar observations have been made for cloud-based services [34, 42, 45, 64, 65, 79, 80, 97, 118]. The remote memory access framework above reduces the overhead of function communication, however, accelerating traditional RPC-based networking is still required, since the edge devices use RPCs to transfer data to/from the cloud [50, 89, 92, 100, 105, 118]. For network acceleration, we piggyback on the same hardware platform used above. Fig. 9 shows how the tightly-coupled FPGA is connected to the host servers and the network, and partitioned between network acceleration and remote memory acceleration. We offload the entire RPC stack on the FPGA, and use the memory interconnect (UPI bus) to view the FPGA as another NUMA node, and quickly transfer data to/from the host CPU, using zero copy. Our FPGA-based implementation supports multiple threads of asynchronous (non-blocking) RPCs. The FPGA's area is large enough to support both remote memory accesses (18% of LUTs) and RPC offloading (24% of LUTs). We statically partition the FPGA

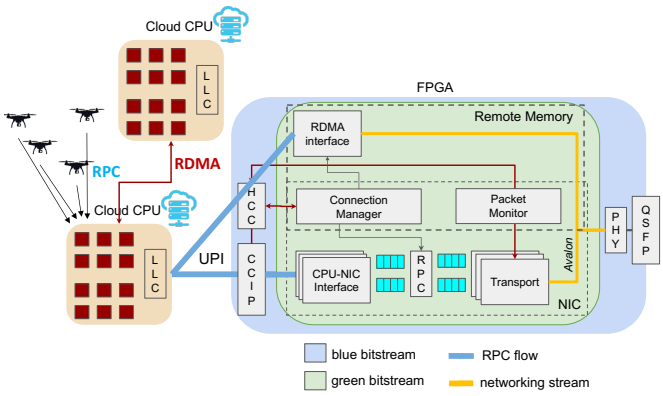

Figure 9: Overview of FPGA-based acceleration fabrics in HiveMind for remote memory access between serverless functions, and for RPC acceleration between the backend cloud and edge devices.

between the two processes, although dynamic partitioning could be supported if needed.

HiveMind's network acceleration supports a multi-connection setup between clients and servers. The networking API defines two classes: the *RPCServer* and the *RPCClient* for each client-server pair. The RPCClient encapsulates a pool of RPC caller threads that concurrently call remote procedures registered in the RPCServer. The software stack sets up the connections and implements a zero-copying API, to directly place incoming RPC requests and responses to dedicated buffers (queues) accessible by the hardware. Buffer sizes are configured on a per-application basis, online, through partial reconfiguration. The rest of the processing is handled by the FPGA NIC. When processing for a packet completes, the FPGA places the RPC payload in a shared memory buffer. Packets are processed to completion by a single thread.

In contrast to existing programmable NICs, which leverage PCIe-based interfaces [37, 41, 64, 78, 79], we again use a NUMA memory interconnect to interface the host CPU with the FPGA, to optimize the transfer of small RPCs, which are common in edge devices. The NUMA memory interconnect is encapsulated into the CCI-P protocol stack [28].

The intuition behind the use of an FPGA for network acceleration is that it allows the networking fabric to be reconfigurable, which suits the diverse needs of edge applications. Reconfiguration is split in *hard* and *soft* reconfiguration. The former is only used for coarse-grained control decisions, such as selecting the CPU-NIC interface protocol or the transport layer (TCP or UDP). Soft reconfiguration, on the other hand, is based on soft register files accessible by the host CPU via PCIe, and their corresponding control logic. It is used for configuring the batch size of CCI-P transfers, provisioning the transmit and receive queues, configuring the queue number and size, configuring the number of active RPC flows, and selecting a load balancing scheme. Soft reconfiguration incurs some small overheads, however, it enables fine-tuning the acceleration fabric to the needs of different applications.

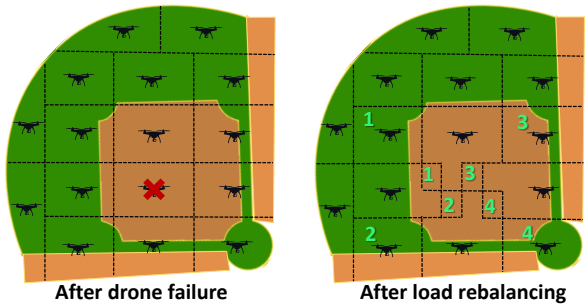

Figure 10: Load repartitioning to handle a drone failure.

The server's NIC simply forwards packets to the FPGA without processing them in the host CPU. HiveMind's network acceleration achieves 2.1us round trip latencies between cloud servers connected to the same ToR switch, and a max throughput with a single CPU core of 12.4Mrps for 64B RPCs. This improves the system's performance predictability, and frees up a lot of CPU resources, which can be used for function execution. In Sec. 5 we evaluate how these benefits factor into end-to-end performance.

4.6 Other Features

Continuous learning: A benefit of centralized coordination is that data from all devices can be collectively used to improve the learning ability of the swarm. A user can enable or disable continuous learning in their application description. If enabled, instead of only using one device's decisions to retrain it, HiveMind leverages the entire swarm's decisions to retrain all devices jointly, which significantly accelerates their decision quality.

Fault tolerance: Edge devices are prone to failures. All devices send a periodic heartbeat to HiveMind (once per second). If the controller does not receive a heartbeat for more than 3s, it assumes that the device has failed. HiveMind handles such failures by repartitioning the load among the remaining devices. Fig. 10 shows such an example for our application scenarios.

Immediately after HiveMind realizes that the red-marked drone has failed, it repartitions its assigned area equally among its neighboring drones assuming they have sufficient battery, and updates their routing information. Depending on which device has failed, this involves reassigning work to a variable number of devices. Users can also specify fault tolerance policies.

Straggler mitigation: HiveMind has a monitoring system that tracks function progress, and flags potential *stragglers*. If a function takes longer than the 90^{th} percentile of that job's functions, OpenWhisk respawns it on new servers, and uses the results of whichever task finishes first [51, 102]. The exact percentile that signals a straggler can be tuned depending on the importance of a job. If several underperforming tasks all come from the same physical node, that server is put on probation for a few minutes until its behavior recovers. OpenWhisk also respawns any failed tasks by default.

4.7 Implementation

The HiveMind compiler and program synthesis frameworks are written is `28,000` lines of C++ and Python. The controller is written in `18,000` lines of C++, and supports Ubuntu 18.04 and newer versions. The controller is implemented as a centralized process, with two hot standby copies that can take over, in case of a failure. We have also implemented a monitoring system that tracks application progress and device status, and verified that it has no meaningful impact on performance; less than 0.1% on tail latency, and less than 0.15% on throughput. Both hardware acceleration processes (remote memory and networking) are implemented using Verilog, System Verilog, and Vivado HLS. HiveMind supports applications in Python, C++, Scala, and node.js.

4.8 Discussion

HiveMind by default assumes that the platform has full control over cloud and edge resources to appropriately place functions on physical machines. However, the techniques in HiveMind are modular, and could be used separately if full system control is not available, as is the case in a public cloud. In such a setting, HiveMind can generate the low-level code of an application from its high level specification, and identify the best mapping between cloud and edge resources. If the cloud provider additionally has support for network-connected FPGAs, HiveMind could harness benefits from network and remote memory acceleration. It would, however, lose the advantages of controlling the physical task placement. Alternatively, if FPGA support is not available, HiveMind would still offer programmability and task placement benefits, but applications would be prone to high network overheads and overheads from the serverless framework's default data exchange protocol (CouchDB for OpenWhisk).

5 EVALUATION

5.1 Performance Analysis

We first examine HiveMind's performance predictability. Fig. 11 shows the performance of all applications with HiveMind compared to the centralized and decentralized platforms. HiveMind's performance is consistently better and less variable compared to both other systems. The applications that benefit the most from HiveMind's design are compute- and memory-intensive services, like maze traversal, image-to-text recognition, and the second end-to-end scenario (moving people recognition) for which offloading all data to the cloud incurs high network overheads, and computing on the drones results in poor and unpredictable performance. Services like drone detection (*S3*) and obstacle avoidance (*S4*) exhibit smaller benefits, consistent with out findings in Sec. 3.

We now examine where these benefits come from. Fig. 12 shows the tail latency breakdown for the centralized system and HiveMind. Network acceleration in HiveMind has a drastic impact on latency, with time for networking dropping from 33% on average to 9.3%. Second, the time associated with management operations, such as container instantiation and scheduling also drops significantly. Most benefits come from HiveMind avoiding instantiation overheads, despite its scheduler incurring slightly higher overheads than the default OpenWhisk Controller. Third, HiveMind's remote

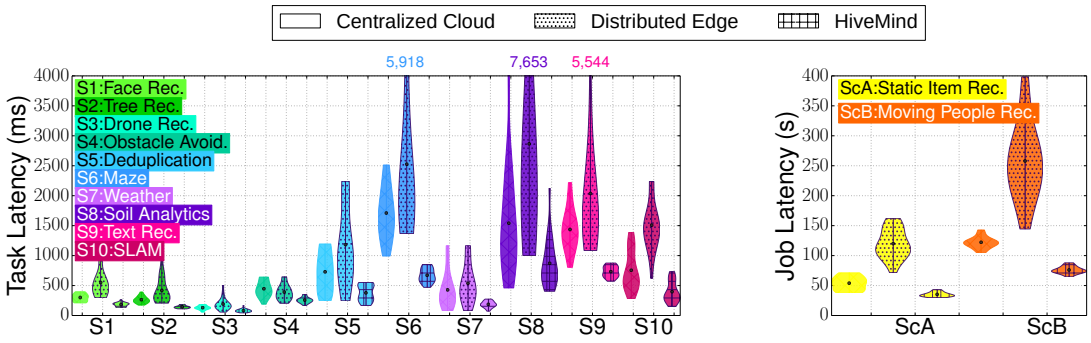

Figure 11: PDFs of task latency with centralized cloud execution, distributed edge execution, and HiveMind across all single-tier tasks and multi-tier scenarios.

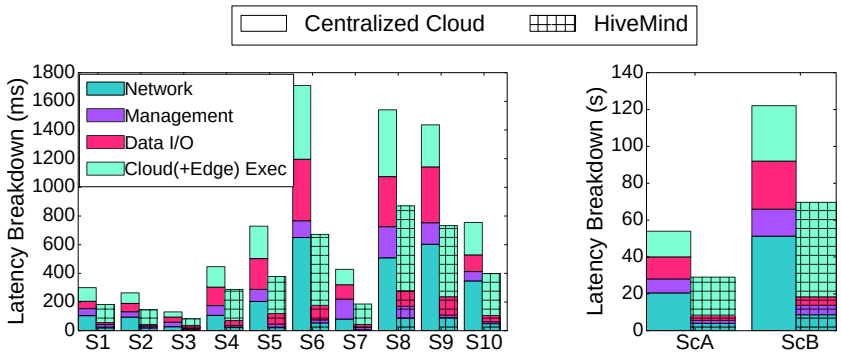

Figure 12: Latency breakdown compared to fully centralized execution to assess the benefits of HiveMind's various components. We show the latency impact from network acceleration, fast management operations (scheduling, instantiation, etc.), reducing data I/O overheads through remote memory access acceleration, and allowing partial edge task execution. Only the latter fraction of latency is slower in HiveMind, as edge devices are resource-constrained, but on-board execution reduces network traffic and improves scalability.

memory access considerably reduces data exchange latency, by avoiding CouchDB accesses. Finally, the fraction of end-to-end latency devoted to execution time in HiveMind increases compared to the centralized system. This is not surprising, as HiveMind maps some tasks on the slower edge devices. However, by eliminating all other system overheads, HiveMind's end-to-end performance is 56% better than Centralized on average, and up to 2.85×, while also using less battery and bandwidth.

Finally, we examine the incremental benefits of the techniques in HiveMind. Specifically, we compare HiveMind to a centralized system with network acceleration, one that also has remote memory access acceleration, distributed systems with and without network acceleration, and Hivemind with no acceleration but hybrid execution. Fig. 13 shows the comparison in terms of median (bars) and tail latency (markers) for all ten jobs and two end-to-end scenarios. In the case of the centralized system with network acceleration, performance benefits from improving networking between edge and cloud resources, although still remains far from HiveMind. Even when remote memory access acceleration is enabled, HiveMind still outperforms the centralized system, as it does not overload cloud resources, by allowing some edge computation. On the other hand, the distributed system barely benefits from hardware acceleration, as most computation happens at the edge, and only final results are transferred to the cloud. Finally, HiveMind without acceleration still benefits from hybrid execution, but is prone to high networking and data transfer overheads. Overall, this analysis shows that no single technique in HiveMind is sufficient to address the performance and power requirements of edge applications in isolation, and that co-designing the software and hardware stack is critical.

5.2 Power Consumption

Fig. 14a shows the consumed battery, on average, across drones by the end of execution; each of the 10 jobs runs for 120s, and the two end-to-end scenarios run to completion; repeated 20 times. HiveMind consumes much less power compared to the distributed system, by offloading resource-demanding computation to the cloud. It also consumes less power than the centralized system, by avoiding excessive data transfer. Even though most power consumption is due to drone motion, communication can also exhaust the device's battery. There are two jobs (*S3* and *S4*) for which HiveMind consumes slightly higher power than the centralized system. This is in line with our previous findings that these jobs do not benefit from dividing their execution between cloud and edge. Finally, the efficiency improvement for the end-to-end scenarios is largely due to HiveMind completing the scenario faster.

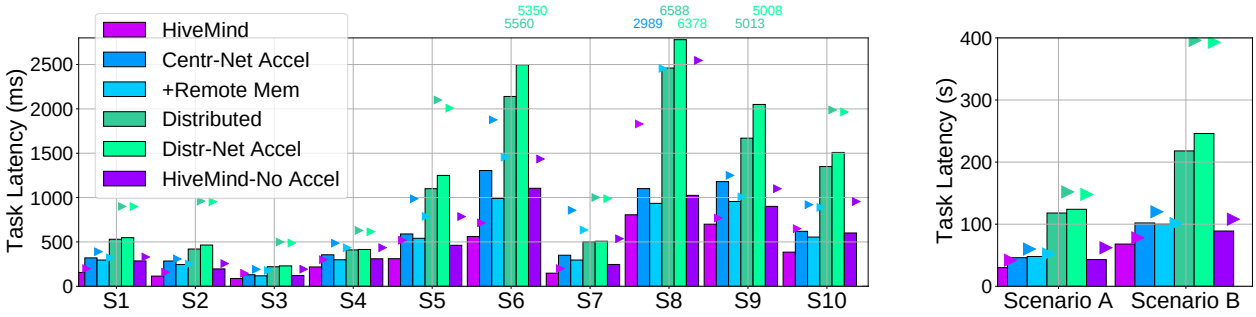

Figure 13: Latency across single-tier tasks and multi-tier scenarios as we disable different aspects of HiveMind. Bars show median, and markers 99^{th} percentile latency. The leftmost bars show performance with HiveMind, "Centr-Net Accel" executes all tasks in the cloud taking advantage of network acceleration, "+Remote Mem" additionally accelerates remote memory access, "Distr" runs all tasks at the edge without any cloud acceleration, "Distr-Net Accel" runs all tasks at the edge but can leverage RPC acceleration when transferring the final results, and "HiveMind-No Accel" is HiveMind without hardware acceleration.

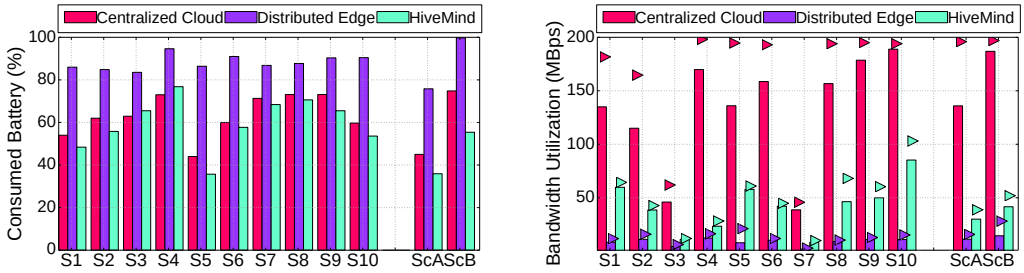

Figure 14: Battery and network bandwidth consumption (bars→median, markers→ 99^{th} %ile) across the three platforms.

5.3 Network Bandwidth

Fig. 14b shows the network bandwidth usage across the three platforms. Bars show average and markers 99^{th} percentile bandwidth. While HiveMind consumes more bandwidth that the distributed system by offloading a fraction of data to the cloud, its usage is much lower than the centralized system, and it avoids the network congestion of offloading all data to the cloud. Note also that the difference between average and tail bandwidth is less pronounced for HiveMind, which contributes to its low performance variability.

5.4 Continuous Learning

We now explore the benefit of HiveMind's centralized backend in continuously improving the swarm's decision quality. Fig. 15 shows the swarm's accuracy in correctly identifying items (tennis balls) and people in the two end-to-end scenarios. If the recognition models are never retrained, there is a non-trivial number of false positives and negatives, even when recognition runs in the cloud. Retraining the models periodically only using feedback from each device's decisions improves accuracy, but still exhibits some incorrect detections. Finally, using the entire swarm's decisions to globally retrain the models, quickly resolves any remaining false negatives and false positives, showing that centralized control enables devices to learn faster than in a decentralized system.

5.5 Applicability to Other Swarms

HiveMind's design is not specific to drones. We now port HiveMind to a swarm of 14 robotic cars (Fig. 16), each equipped with

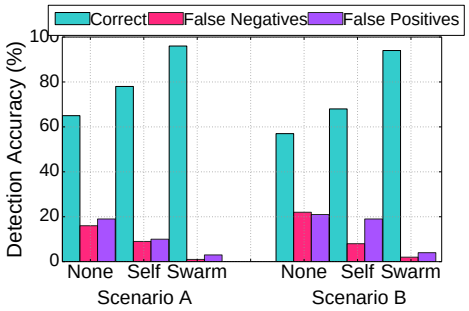

Figure 15: Decision quality without and with retraining, per drone and swarm-wide.

a high definition front camera, a Raspberry Pi board, and GPS, accelerometer, temperature, and altitude sensors [27]. The serverless cluster is the same as before. Communication happens over a wireless network using TCP/IP. We explore two scenarios; a "Treasure Hunt", where, robots navigate a space with panels providing them instructions on where to move next until they reach a final target, and a "Maze", where they have to navigate an unknown maze. The first scenario involves image-to-text conversion to interpret the provided instructions. The user again expresses each scenario's task graph in HiveMind's DSL and provides the necessary task logic, and the system determines how to place tasks. The cars are less power-constrained than the drones, so obstacle avoidance and sensor analytics almost always run on-board.

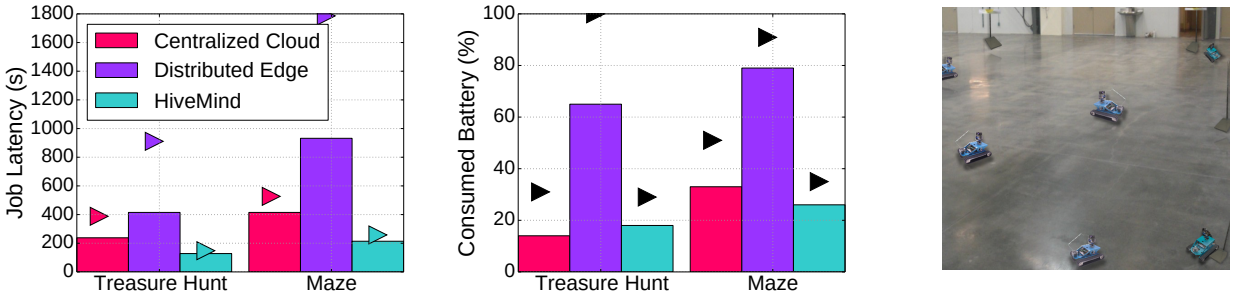

Figure 16: Latency and battery consumption of robotic cars (bars→median, markers→tail latency or battery consumption), and subset of car swarm in "Treasure hunt" scenario.

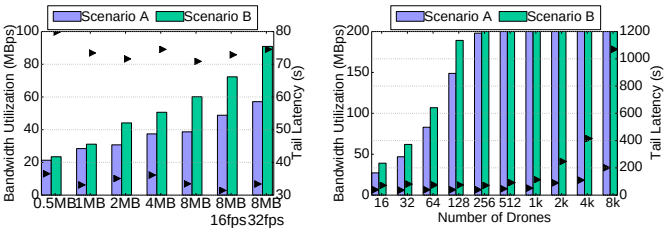

Figure 17: Bandwidth (bars) and tail latency (markers) as we increase (a) resolution, and (b) #drones.

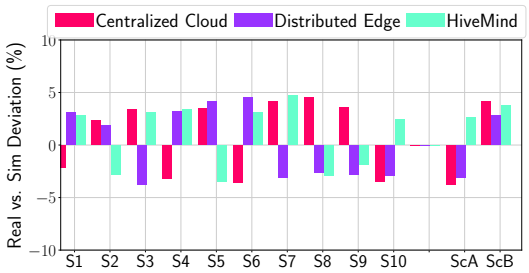

Figure 18: Validation of the simulator in terms of tail latency for the 16-drone swarm across the three configurations.

Fig. 16a shows the median (bars) and tail (markers) job latency on HiveMind, and the centralized and distributed systems. Fig. 16b shows the corresponding average (bars) and worst-case (markers) battery consumption. Performance is better and more predictable with HiveMind, especially compared to the distributed system. As with the drone swarm, the cars significantly benefit from network acceleration (22% lower latency on average), but they also benefit from offloading expensive computation to the serverless cluster, without high instantiation overheads. Finally, because both scenarios have multiple phases, HiveMind's fast remote memory access significantly reduces their latency (19% on average).

5.6 Scalability

Fig. 17a shows the bandwidth usage and tail latency for the two scenarios on the drone swarm, as the image resolution increases. Even for the maximum resolution and frame rate (32 fps), HiveMind does not saturate the network links, keeping latency low. In contrast, in the centralized system, network quickly became congested, only supporting modest resolutions and low frame rates (Fig. 3).

Since experimenting with hundreds of drones is impractical, we have also built a validated, event-driven simulator that accurately captures the performance, battery consumption, and network bandwidth of the real swarm. The simulator is based on queueing network principles and tracks the processing and queueing time both on cloud and edge resources. We have used the real drone and robotic car testbeds to validate the simulator's accuracy. Fig. 18 shows a snapshot of the simulator's validation for the 16-drone swarm in HiveMind, the centralized, and distributed systems. We show the deviation in tail latency between the real experiments and simulation. In all cases deviation is less than 5%. The results are similar for the robotic cars.

We use the simulator to explore HiveMind's scalability to the swarm size. Fig. 17b shows the network bandwidth usage and tail latency for the two scenarios. We scale up the network links proportionately to the real experiments. Although for larger swarms the bandwidth usage increases, the increase is much slower than increase rate in devices, compared to a linear increase with the centralized system, especially for the first scenario, which accommodates more computation on-board.

6 CONCLUSION

We have presented HiveMind, a hardware-software system stack for edge swarms, which bridges the gap between centralized and distributed coordination. HiveMind implements a DSL to improve programmability for these systems, automatically handles task mapping between cloud and edge resources, and proposes hardware acceleration fabrics for remote memory access and networking. On real swarms with 16 drones and 14 robotic cars, HiveMind significantly outperforms prior systems, reducing overheads and abstracting away complexity.

ACKNOWLEDGEMENTS

We sincerely thank Shuang Chen, Yanqi Zhang, Daniel Sanchez, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was in part supported by an NSF CAREER Award CCF-1846046, NSF grant NeTS CSR-1704742, a Google Faculty Research Award, a Sloan Research Fellowship, a Microsoft Research Fellowship, an Intel Faculty Rising Star Award, a Facebook Research Faculty Award, and a John and Norma Balen Sesquisentennial Faculty Fellowship.

REFERENCES

[1] "5 ways you're probably messing up your microservices," https://www.overops.com/blog/5-ways-to-not-f-up-your-microservices-in-production/.
[2] "Apache couchdb," https://couchdb.apache.org/.
[3] "Apache thrift," https://thrift.apache.org.
[4] "Ar drone pose estimation with the orb-slam library."
[5] "ardrone-autonomy," https://ardrone-autonomy.readthedocs.io/en/latest/.
[6] "Aws lambda," https://aws.amazon.com/lambda.
[7] "Azure functions," https://azure.microsoft.com/en-us/services/functions/.
[8] "Coco: Common objects in context," http://cocodataset.org/.
[9] "Contracts, addressing, and apis for microservices," https://cloud.google.com/appengine/docs/standard/java/designing-microservice-api.
[10] "Cylon.js," https://cylonjs.com/.
[11] "Eight common microservices performance problems (and how to solve them)," https://jaxenter.com/microservices-performance-problems-172291.html.
[12] "fission: Serverless functions for kubernetes," http://fission.io.
[13] "Fleets of drones could aid searches for lost hikers," http://news.mit.edu/2018/fleets-drones-help-searches-lost-hikers-1102.
[14] "Google cloud functions: Event-driven serverless compute platform," https://cloud.google.com/functions/.
[15] "Linksys ea8300 max-stream ac2200 tri-band wifi router," https://www.linksys.com/us/p/P-EA8300/.
[16] "Microservice issues, challenges, and hurdles," https://dzone.com/articles/microservice-issues-challenges-and-hurdles.
[17] "Mit creates control algorithm for drone swarms," https://techcrunch.com/2016/04/22/mit-creates-a-control-algorithm-for-drone-swarms/.
[18] "Openfaas: Serverless functions, made simple," https://www.openfaas.com/.
[19] "Openlambda," https://open-lambda.org.
[20] "Opewhisk: Open source serverless cloud platform," http://openwhisk.apache.org/.
[21] "Parrot ar.drone 2.0 edition," https://www.parrot.com/us/drones/parrot-ardrone-20-elite-edition.
[22] "Pyflow: A lightweight parallel task engine," http://illumina.github.io/pyflow/.
[23] "Robond-path planning: Wall follower for maze traversal," https://github.com/udacity/RoboND-PathPlanning.
[24] "Simplify iot development and drive digital transformation with your azure free account," https://azure.microsoft.com/en-us/free/iot/.
[25] "Tensorflow model zoo."
[26] "Who will control the swarm?" https://platformlab.stanford.edu/news.php.
[27] "Yahboom professional raspberry pi smart robot kit, diy tank robotics kit."
[28] "Intel acceleration stack for Intel Xeon CPU with FPGAs core cache interface (CCI-P) reference manual," accessed May, 2020, https://www.intel.com.
[29] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," in *Proceedings of OSDI, 2016*.
[30] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 923–935.
[31] R. N. Akram, K. Markantonakis, K. Mayes, O. Habachi, D. Sauveron, A. Steyven, and S. Chaumette, "Security, privacy and safety evaluation of dynamic and static fleets of drones," in *Proc. of the IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*. 2017.
[32] D. Albani, D. Nardi, and V. Trianni, "Field coverage and weed mapping by uav swarms," in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017.
[33] A. L. Alfeo, M. Cimino, N. D. Francesco, A. Lazzeri, M. Lega, and G. Vaglini, "Swarm coordination of mini-uavs for target search using imperfect sensors," in *Intelligent Decision Technologies, IOS Press, Vol. 12, Issue 2, Pages 149-162*. 2018.
[34] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 435–446. [Online]. Available: https://doi.org/10.1145/2486001.2486031
[35] S. Allen, C. Aniszczyk, C. Arimura, B. Browning, L. Calcote, A. Chaudhry, D. Davis, L. Fourie, A. Gulli, Y. Haviv, D. Krook, O. Nissan-Messing, C. Munns, K. Owens, M. Peek, and C. Zhang, "Cncf serverless whitepaps v1.0." CNCF Technical Report.
[36] M. Almeida, H. Hildmann, and G. Solmaz, "Distributed uav-swarm-based real-time geomatic data collection under dynamically changing resolution requirements," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLII-2/W6, pp. 5–12, 08 2017.
[37] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols in high-speed NICs," *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2020.
[38] L. Baresi, D. Filgueira Mendonça, and M. Garriga, "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture," in *6th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, ser. Service-Oriented and Cloud Computing, F. D. Paoli, S. Schulte, and E. B. Johnsen, Eds., vol. LNCS-10465. Oslo, Norway: Springer International Publishing, Sep. 2017, pp. 196–210, part 6: Internet of Things and Data Streams.
[39] L. Barroso, "Warehouse-scale computing: Entering the teenage decade," in *Proceedings of the 38th Intl. symposium on Computer architecture*, San Jose, CA, 2011.
[40] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.
[41] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.
[42] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with FPGAs," *Workshop on Hot Topics in Cloud Computing*, 2013.
[43] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. J. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 894–907.
[44] M. Campion, P. Ranganathan, and S. Faruque, "Uav swarm communication and control architectures: a review," *Journal of Unmanned Vehicle Systems*, vol. 7, no. 2, pp. 93–106, 2019.
[45] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 7:1–7:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195647
[46] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 301–314.
[47] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefle, "rfaas: Rdma-enabled faas platform for serverless high-performance computing," in *arXiv 2106.13859v1 [cs.DC], 25 Jun 2021*.
[48] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*.
[49] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 49–62.
[50] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-driven tail-aware balancing of Ms-scale RPCs," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
[51] J. Dean and L. A. Barroso, "The tail at scale," in *CACM, Vol. 56 No. 2, Pages 74-80*.
[52] J. R. Del Rosario, J. Sanidad, A. Lim, P. Uy, A. Bacar, M. Cai, and A. Dubouzet, "Modelling and characterization of a maze-solving mobile robot using wall follower algorithm," *Applied Mechanics and Materials*, vol. 446-447, pp. 1245–1249, 11 2013.
[53] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.
[54] C. Delimitrou and C. Kozyrakis, "QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon," in *ACM Transactions on Computer Systems (TOCS), Vol. 31 Issue 4*. December 2013.
[55] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.
[56] C. Delimitrou and C. Kozyrakis, "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems," in *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.
[57] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.
[58] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 939–954.

[59] M. Dogar, A. Spielberg, S. Baker, and D. Rus, "Multi-robot grasp planning for sequential assembly operations," in *IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, WA, 2015.

[60] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481.

[61] M. A. Estrada, S. Mintchev, D. L. Christensen, M. R. Cutkosky, and D. Floreano, "Forceful manipulation with micro air vehicles," *Science Robotics*, vol. 3, no. 23, 2018.

[62] F. Faticanti, F. D. Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Swarm coordination of mini-uavs for target search using imperfect sensors," in *arXiv:1810.04442*. 2018.

[63] M. Femminella, M. Pergolesi, and G. Reali, "Performance evaluation of edge cloud computing system for big data applications," in *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, 2016, pp. 170–175.

[64] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[65] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," *USENIX Annual Technical Conf. (ATC)*, 2013.

[66] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. S. Wahby, and K. Winstein, "Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 267–282.

[67] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376.

[68] Y. Gan and C. Delimitrou, "The Architectural Implications of Cloud Microservices," in *Computer Architecture Letters (CAL), vol.17, iss. 2*, Jul-Dec 2018.

[69] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices," in *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[70] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[71] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou, "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[72] H. Genc, Y. Zu, T.-W. Chin, M. Halpern, and V. J. Reddi, "Flying iot: Toward low-power vision in the sky," in *IEEE Micro (Volume: 37, Issue: 6, November/December 2017)*.

[73] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. R. Das, "Spock: Exploiting serverless functions for slo and cost aware resourceprocurement in public cloud," in *Proc. of of IEEE Cloud Computing (CLOUD)*. 2019.

[74] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236.

[75] Y. Han, X. Wang, V. C. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," in *arXiv:1907.08349 v1 [cs.NI] 19 Jul 2019*.

[76] J. Hellerstein, J. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *Proc. of Conference on Innovative Data Systems Research (CIDR)*, 2018.

[77] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016.

[78] S. Ibanez, M. Shahbaz, and N. McKeown, "The case for a network fast path to the CPU," *ACM Workshop on Hot Topics in Networks*, 2019.

[79] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: A highly scalable user-level tcp stack for multicore systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 489–502. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616493

[80] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "Eyeq: Practical network performance isolation at the edge," in *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, 2013.

[81] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 152–166.

[82] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 158–164.

[83] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," *USENIX Annual Technical Conf. (ATC)*, 2016.

[84] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2016.

[85] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[86] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: a language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 296–311.

[87] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 115–127. [Online]. Available: https://doi.org/10.1109/ISCA.2016.20

[88] C. Lai, J. Kimball, T. Zhu, Q. Wang, and C. Pu, "milliscope: A fine-grained monitoring framework for performance debugging of n-tier web services," in *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, 2017, pp. 92–102.

[89] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou, "Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs," in *Computer Architecture Letters (CAL)*, July-December 2020.

[90] S.-C. Lin, "Cross-layer system design for autonomous driving," in *Doctoral Dissertation, The University of Michigan*. 2019.

[91] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. Williamsburg, VA, USA: ACM, 2018, pp. 751–766. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173191

[92] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-efficient microservices on SmartNIC-accelerated servers," *USENIX Annual Technical Conf. (ATC)*, 2019.

[93] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proc. of IEEE International Conference on Cloud Engineering (IC2E)*. 2018.

[94] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[95] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017.

[96] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawełczak, "Dynamic task-based intermittent execution for energy-harvesting devices," *ACM Trans. Sen. Netw.*, vol. 16, no. 1, Feb. 2020.

[97] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[98] S. Nasser, A. Barry, M. Doniec, G. Peled, G. Rosman, D. Rus, M. Volkov, and D. Feldman, "Fleye on the car: Big data meets the internet of things," in *Proc. of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*. Seattle, 2015.

[99] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[100] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[101] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 57–70.

[102] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of SOSP*. Farminton, PA, 2013.

[103] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments." IEEE International Conference on Fog Computing (ICFC), 2020.

[104] D. Pinto, J. P. Dias, and H. Sereno Ferreira, "Dynamic allocation of serverless functions in iot environments," in *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2018, pp. 1–8.

[105] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[106] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. D. Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, 2016, pp. 651–665.

[107] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 43–56.

[108] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge AI," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. Renton, WA: USENIX Association, Jul. 2019.

[109] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozych, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of SOCC*. 2012.

[110] L. Sanneman, D. Ajilo, J. DelPreto, A. Mehta, S. Miyashita, N. A. Poorheravi, C. Ramirez, S. Yim, S. Kim, and D. Rus, "A distributed robot garden system," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[111] S. Sarkar, R. Wankar, S. N. Srirama, and N. K. Suryadevara, "Serverless management of sensing systems for fog computing framework," *IEEE Sensors Journal*, vol. 20, no. 3, pp. 1564–1572, 2020.

[112] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

[113] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of EuroSys*. Prague, Czech Republic, 2013.

[114] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1063–1075.

[115] M. M. Shurman and M. K. Aljarah, "Collaborative execution of distributed mobile and iot applications running at the edge," in *2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, 2017, pp. 1–5.

[116] A. Singhvi, S. Banerjee, Y. Harchol, A. Akella, M. Peek, and P. Rydin, "Granular computing and network intensive applications: Friends or foes?" in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI. New York, NY, USA: ACM, 2017, pp. 157–163.

[117] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "Babelfish: Fusing address translations for containers," *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 501–514, 2020.

[118] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The NeBuLa RPC-optimized architecture," *Int'l Symp. on Computer Architecture (ISCA)*, 2020.

[119] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture formobile computing," in *Proc. of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*. San Francisco, 2016.

[120] S. Trilles, A. González-Perez, and J. Huerta, "An iot platform based on microservices and serverless paradigms for smart farming purposes." Sensors 2020.

[121] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 33–48.

[122] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is more: From paas to present cloud computing," in *IEEE Internet Computing (Volume: 22 , Issue: 5 , Sep./Oct. 2018)*.

[123] G. Vásárhelyi, C. Virágh, G. Somorjai, T. Nepusz, A. E. Eiben, and T. Vicsek, "Optimized flocking of autonomous drones in confined environments," in *Science Robotics 18 Jul 2018, Vol. 3, Issue 20*.

[124] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A memory-disaggregated managed runtime," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 261–280.

[125] Z. Yuan, J. Jin, L. Sun, K.-W. Chin, and G.-M. Muntean, "Ultra-reliable iot communications with uavs: A swarm use case," in *IEEE Communications Magazine (Volume: 56, Issue: 12, December 2018)*.

[126] Q. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Data processing and sharing for hybrid cloud-edge analytics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2004–2017, 2018.

[127] Y. Zhang, Y. Gan, and C. Delimitrou, "μqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices," in *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2019.