Coding the Computing Continuum: Fluid Function Execution in Heterogeneous Computing Environments

Rohan Kumar[1], Matt Baughman[1], Ryan Chard[2], Zhuozhao Li[1,2], Yadu Babuji[1,2], Ian Foster[1,2], and Kyle Chard[1,2]

[1]Department of Computer Science, University of Chicago, Chicago, IL, USA
[2]Data Science and Learning Division, Argonne National Laboratory, Lemont, IL, USA

***Abstract*—Advances in network technologies have greatly decreased barriers to accessing physically distributed computers. This newfound accessibility coincides with increasing hardware specialization, creating exciting new opportunities to dispatch workloads to the best resource for a specific purpose, rather than those that are closest or most easily accessible. We present Delta, a service designed to intelligently schedule function-based workloads across a distributed set of heterogeneous computing resources. Delta implements an extensible architecture in which different predictors and scheduling algorithms can be integrated to provide dynamically evolving estimates of function execution times on different resources—estimates that can be used to determine the most appropriate location for execution. We describe predictors for function runtime, data transfer time, and cold-start resource provisioning and configuration delay; dynamic learning methods that update predictor models over time; and scheduling strategies that take into account both function and endpoint information. We show that these methods can halve workload makespan when compared with a strategy that selects the fastest resource, and decrease makespan by a factor of five when compared to a round robin strategy, when deployed on a heterogeneous testbed with resources ranging from a Raspberry Pi to a GPU node in an academic cloud.**

***Index Terms*—Computing continuum, function as a service, serverless, heterogeneous computing, scheduling.**

I. Introduction

The last decade has seen rapid movement towards the natural next step in improving application performance—hardware specialization. Developers now have access to a near limitless range of computing devices, establishing what some refer to as a computing continuum [1, 2]. However, specialization leads to distribution, and thus the most suitable hardware for a computation is likely to be physically remote. Fortunately, corresponding increases in network performance [3] make it now possible to distribute workloads to remote computers at almost the speed of light.

Effective task distribution in such heterogeneous, hyper-connected environments requires two distinct but complementary functionalities: 1) mechanisms for easy, reliable, and efficient task execution on remote and heterogeneous resources; and 2) the ability to rapidly, effectively, and automatically determine good available resources. The first objective is met, at least in part, by modern function-as-a-service (FaaS) platforms [4] that provide convenient abstractions for executing function calls on remote computers without regard to underlying physical and virtual infrastructure. The second objective, however, becomes increasingly complex as devices become more heterogeneous. Additionally, resource selection is not simply a matter of choosing the fastest or closest device. Instead, the true cost of running a particular application on a specific resource can depend on many factors, including the performance of the application on that resource, time to provision and configure the resource, the time to transfer data to that resource, and the reliability of the resource.

Here we present **D**istributed **E**xecution of **L**ambdas using **T**rade-off **A**nalysis (Delta), a service for scheduling function executions across heterogeneous, distributed FaaS resources and thus enabling fluid computation from edge devices to clouds and supercomputers. Delta is designed to work with existing FaaS platforms and in particular those that can be deployed on arbitrary computing resources. It serves as a high-level scheduler able to profile and predict function performance using a set of extensible predictors, manage data movement across connected resources, and route function executions to resources based on extensible scheduling policies.

We implemented a Delta prototype using the funcX FaaS platform [5] to manage function execution on remote endpoints and using Globus [6] to manage data transfers between endpoints. We developed machine learning predictors that estimate function runtime on heterogeneous hardware, data transfer time between endpoints, and cold start time (e.g., node acquisition, container instantiation, and Python package loading). We combined these predictors into a single model for determining the expected execution time of a function on a given endpoint. We subsequently used this model to develop scheduling strategies that aim to select the fastest endpoint for a task and to minimize the expected execution time of individual tasks. We evaluate Delta on a testbed with 11 endpoints including Raspberry Pis, desktops, and Amazon and Chameleon [7] cloud instances. Experiments with benchmark workloads show that Delta can quickly learn tradeoffs between functions and resources, and that its estimated execution scheduling strategy can reduce makespan by more than a factor of five when compared to a round robin approach and by more than double when compared to a strategy that selects the fastest endpoint for a function without considering other factors.

The rest of this paper is as follows. Section II introduces our research questions and the distributed function scheduling problem. Section III describes the Delta architecture, and Section IV evaluates Delta's performance with benchmark workloads on a heterogeneous testbed. Section V reviews related work. Finally, Section VI summarizes our contributions.

II. Challenges and Problem Formulation

We discuss challenges associated with scheduling functions for execution across diverse and distributed resources, formalize the problem, describe how we model function execution time, and present assumptions we make.

A. Challenges

Choosing where to execute a function in a heterogeneous and distributed system depends on myriad factors ranging from the performance of a particular function on a particular resource through to the time to provision its execution environment and transfer input data to that environment. We describe the factors we consider here.

1) Function Runtime: Different applications perform differently on different resources, as illustrated in Fig. 1, which shows the performance achieved on seven resource types for three applications: a parallel counter (ParCount), a matrix multiplication (MatMul), and an I/O heavy application (FileIO).

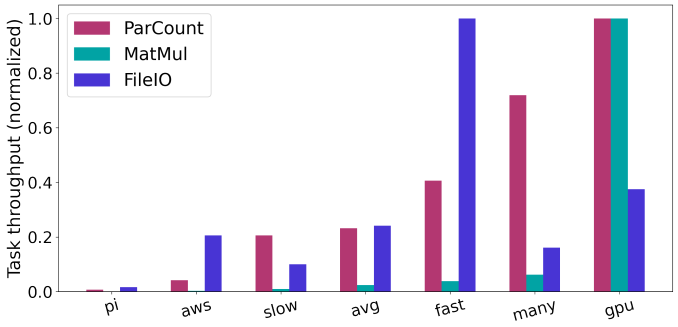

Fig. 1: Relative throughputs (tasks/sec) for three tasks on the resource types of Table I, with values normalized to the maximum observed on any resource.

2) Data Transfer: A large contributor to execution duration is the time required to move task-execution information to a remote endpoint. This involves not only the latency of communicating with the endpoint, but also the cost of transferring data in cases where the function needs to act upon large input data. Data transfer rates can vary enormously, depending on endpoint capabilities and available network bandwidth.

3) Cold Starts: The *cold start problem* is a well-known and well-documented issue in FaaS literature [5, 8, 9], and represents an important scheduling difference between serverless computing and conventional HPC. Broadly, the problem is that the first invocation of a function typically incurs the time to provision and configure infrastructure, which may include time to request resources, configure virtual machines, deploy containers, and load function dependencies. Contemporary FaaS systems have taken varied approaches to this problem, from keeping "pre-warmed" containers [9–11] to applying optimized container technologies [8]. While prior work has focused primarily on cloud environments, we focus on the more general case where resources include both cloud and HPC resources. Acquisition of nodes on HPC resources is typically the most significant (and variable) factor.

B. Problem Formulation

We define an *endpoint* as a FaaS-enabled computing resource where users can execute functions: for example, a cloud instance, supercomputer, desktop, or edge device.

We have a heterogeneous set of n such endpoints, $\{\mathsf{EP}_1, \ldots, \mathsf{EP}_n\}$. We receive a stream of *tasks* of the form $(\mathsf{f}, \mathsf{x}, \mathsf{data})$ where f is a function, data is a list of files (on the requesting machine or elsewhere) needed by the computation, and x is function input (e.g., paths to data). Neither the timing nor nature of these tasks is known ahead of time. We must decide where to send each task when it is received.

1) Objective: We aim to select an endpoint for each task as it is received in order to minimize overall *time-to-completion*. (Other objectives that could be considered include minimizing data movement, maximizing resource utilization).

2) Modeling the Continuum: We decompose the time taken to execute a task $\mathsf{f}(\mathsf{x}, \mathsf{data})$ on an endpoint EP into:

- *Scheduling overhead* (t_{sched}): Time to make a scheduling decision and to queue the task for execution.
- *Cold-start latency* ($t_{\text{cold}}(\mathsf{EP}, \mathsf{f})$): Time to allocate a node, start a container, and load package dependencies for f, if the endpoint is not already warm.
- *Pending-tasks delay* ($t_{\text{prev}}(\mathsf{EP})$): Time for all previously scheduled tasks on EP to finish. (As noted below, we assume FIFO execution on an endpoint.)
- *Function transfer time* ($t_{\text{trans}}(\mathsf{EP}, \mathsf{f}, \mathsf{x})$): Time to transfer f and x from submit side to EP.
- *Data transfer time* ($t_{\text{trans}}(\mathsf{EP}, \mathsf{data})$): Time to transfer each input file in data from its source to EP.
- *Runtime* ($t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}, \mathsf{data})$): Time for EP to run f on input x and data, producing output res.
- *Result transfer time* ($t_{\text{trans}}(\mathsf{EP}, \mathsf{res})$): Time to transfer res from EP to submit site.

Thus, we determine the endpoint EP_i that minimizes a function's expected execution time (EET) as follows:

$$\underset{1 \le i \le n}{\mathbf{argmin}} \begin{pmatrix} t_{\text{sched}} + t_{\text{cold}}(\mathsf{EP}_i, \mathsf{f}) + t_{\text{prev}}(\mathsf{EP}_i) + \\ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{data}) + t_{\text{trans}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}) + \\ t_{\text{run}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}, \mathsf{data}) + t_{\text{trans}}(\mathsf{EP}_i, \mathsf{res}) \end{pmatrix} \quad (1)$$

3) Assumptions: We make the following assumptions to restrict the scope of the problem. While unrealistic, they allow us to explore aspects of the general case.

(A1) $t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}, \mathsf{data})$ is constant for a particular EP, f, x, and data.

(A2) The time to schedule a task, t_{sched}, is a constant.

(A3) The time to communicate a function to, and results from, different endpoints, $t_{\text{trans}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}) + t_{\text{trans}}(\mathsf{EP}_i, \mathsf{res})$, is constant for all EP, f, x, res. (These times were small relative to other costs in the studies reported here.)

(A4) Each endpoint runs one task at a time, in FIFO order.

(A5) Each task runs on a single endpoint (i.e., no task can be distributed over multiple endpoints).
(A6) There are no inter-task execution dependencies managed by Delta (these can occur freely from within a function).
(A7) We seek only to minimize task time-to-completion.

III. Delta Design and Implementation

Delta is a service that provides function profiling, prediction, and scheduling over existing FaaS systems: see Fig. 2. Users can use Delta to manage task execution across one or more function-serving endpoints. To do so, they use the Delta Python client to invoke tasks directly from within a programming environment, as illustrated in Listing 1.

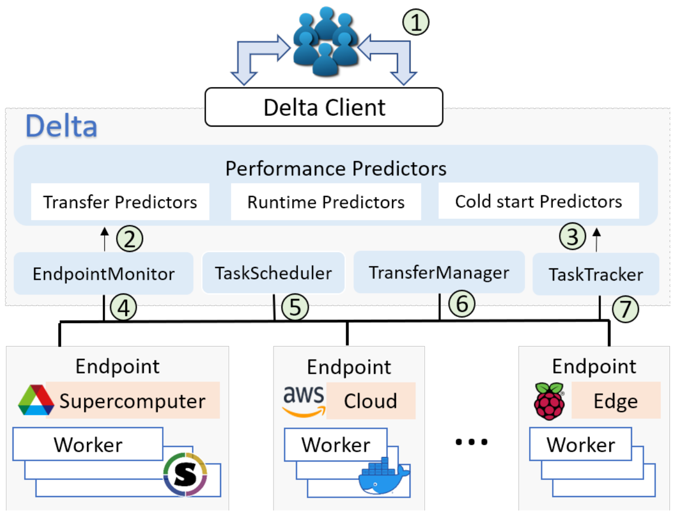

Fig. 2: Delta architecture, showing components of the scheduling service and the communications required to execute tasks. (1) Submit tasks and fetch results; (2) and (3) supply endpoint and task information to predictors; (4) regularly provide endpoint status; (5) distribute tasks across endpoints; (6) schedule and track data transfers; (7) regularly monitor each task's status.

Delta follows a standard FaaS model in which users register a function before invocation, providing self-contained function code, a clearly stated function signature, and any dependencies. Delta wraps the function and registers it in turn with the associated FaaS platform. When a Delta client requests that a task $\mathrm{f(x, data)}$ (i.e., a function plus arguments and data) be executed, Delta selects an endpoint, manages data transfers to that endpoint, waits for transfers to complete, and then forwards the request to the FaaS service to execute the task on that endpoint. Users can subsequently retrieve results asynchronously via Delta.

A. Core Components

The Delta service maintains various components to enable task scheduling and performance prediction, task tracking, data transfers, and resource monitoring. We describe these components in the following subsections.

1) Performance Predictors and Task Scheduling: Delta includes methods for training predictive models on historical data to estimate the various aspects of function execution time listed in Section II-B. Delta's `TaskScheduler` uses

```python
from delta import DeltaClient

# Define function to be scheduled via Delta
def classify(img):
    import tensorflow as tf
    ...
    return model.predict(img)

# Create client and register function
client = DeltaClient()
func_id = client.register_function(classify)

# Request task to run function on image
img = ...
task_id = client.run(img, function_id=func_id)

# Retrieve task result
res = client.get_result(task_id, block=True)
```

Listing 1: Using Delta SDK to register and invoke a function.

trained predictors to determine the most appropriate endpoint for a task's execution. When given an incoming task $\mathrm{t = (f, x, data)}$, it applies a scheduling strategy to determine which endpoint should be used, retrieving predictions from predictors as needed. The `TaskScheduler` is extensible, allowing for arbitrary predictors and scheduling strategies.

2) Task Tracking: We assume that function execution endpoints execute tasks in FIFO order. An essential metric for execution time prediction is an estimate of when all tasks scheduled on each endpoint will finish running—the endpoint's *pending time*. For this, the `TaskTracker` maintains a FIFO queue of scheduled tasks for each endpoint that it updates whenever a new task is sent to the endpoint.

3) Data Transfer: The `TransferManager` is responsible for managing data transfers between endpoints. If input data is not available on the selected endpoint, the `TransferManager` starts data transfer(s) between the source(s) and the destination endpoint, tracks transfer status, and dispatches the task to the endpoint only when it is notified that the transfer has completed.

4) Resource Monitoring: The `EndpointMonitor` component tracks the status of each execution endpoint. It relies on a heartbeat message from the endpoint to the Delta service, which provide information about the endpoint's state, available capacity, queued workload, and utilization of its resources.

B. External Services

Delta leverages funcX [5] for function execution and Globus [6] for data transfers.

The **funcX** federated and distributed FaaS platform allows users to register any computing resource as an *endpoint* on which they can then execute functions via the funcX service. funcX is an attractive choice for an execution fabric upon which Delta can be implemented. All aspects of function execution happen via the `FuncXClient`. Functions are registered by the `FuncXClient` and assigned a *function-id* by the funcX service. Functions can then be executed, by specifying the function-id, endpoint-id, and any input parameters through the `FuncXClient`. The resulting task is assigned a unique *task-id* via which users can monitor task status and retrieve the results asynchronously.

A funcX endpoint uses Parsl [12] to provision and manage compute resources. Each endpoint hosts a *manager* for receiving incoming tasks from the funcX service. The endpoint manages a collection of *workers* deployed on the various nodes available for execution. For example, in an HPC setting, nodes are provisioned via the HPC scheduler, while in a cloud setting nodes are provisioned via the cloud API. The manager routes tasks to registered workers for execution. When the function completes, serialized results are returned to the funcX service and stored in a task database, awaiting a pull from the client.

Globus [13] provides managed high-performance data transfers between remote endpoints. It implements a third-party transfer model via which a user, or in this case the Delta service, can request a data transfer between pairs of data transfer endpoints. (More than 20,000 are active, and others can easily be deployed.) The cloud service offers a REST API for programmatic management of transfers.

Globus uses the GridFTP protocol for high-performance data transfer, and implements advanced features including parallel data streams for performance, automated integrity validation via checksums, automated restarts in the case of errors and file corruption, and a comprehensive security model that enables authentication at both source and destination as well as encryption of the data channel.

C. Function Execution

When a user registers a function, Delta wraps it with Delta-specific code to allow Delta to track task execution. Delta then registers the augmented function with funcX.

When making a task request, the user need not specify an endpoint for execution, as Delta determines this before sending the request to funcX. The user may optionally provide data specifications and the source Globus endpoint(s), which are included as part of the function execution request, and data will be transferred prior to function execution by the Delta service. The Delta service tracks pending tasks via a background `TaskWatchdog` thread which regularly polls the funcX service to inquire about the status of each pending task. This polling enables Delta to receive results as soon as they are ready and allows for more accurate runtime predictions.

Delta provides for resiliency by generating backup tasks when the `EndpointMonitor` detects that an endpoint has failed, or if a task takes significantly longer than expected. A backup task is always sent to an endpoint other than those chosen previously for a task. Multiple backups can be sent, up to a configurable `max_backups` limit. Delta does not send backup tasks during the exploration phase, as runtime predictions are unlikely to be accurate.

D. Endpoint Monitor Implementation

The Delta `EndpointMonitor` is responsible for determining the current state of an endpoint and reports information such as task queue depth, worker status, and resource utilization. This information allows Delta's predictors to estimate wait time for specific endpoints. For example, the `EndpointMonitor` can track if warm nodes are available by checking if the funcX endpoint has active managers and workers. If no active managers are reported, the `EndpointMonitor` reports the endpoint as *cold*. This fact is used for subsequent scheduling decisions. If a cold endpoint is chosen for execution, Delta marks it as *warming*, and once it regains an active manager, it is marked as *warm*. The `EndpointMonitor` also keeps track of which packages have been imported on the endpoint worker's environment. The required packages for a function are determined at the time of function registration by the Delta service.

E. Predictors

To validate the Delta architecture we implemented several proof-of-principle predictors.

A **runtime predictor** uses timings $(\mathsf{f}, \mathsf{x}_i, \mathsf{data}_i, t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}_i, \mathsf{data}_i))$ from previous runs of f on EP to estimate $t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}, \mathsf{data})$. As function runtime often depends heavily on its inputs, we implement for the studies reported here an *input size predictor* that uses online polynomial regression to fit observed $\{$input-size, runtime$\}$ pairs for each function. To reduce computational cost, we group endpoints with similar execution capabilities (e.g., CPU architecture, memory and disk capabilities) and create runtime models for these groups. We rely on our previous works [14, 15] to provide the necessary predictive abilities as this task is difficult and crucial.

Transfer predictors estimate the data transfer time between two locations. As with runtime predictors, we group endpoints by the features that affect transfer times, namely physical location, network connectivity, system memory, and network hardware. We can then measure for each *transfer group* how transfer rates scale with data size when explicit data scaling information is available through the function declaration.

Fig. 3 shows that transfer rates vary greatly with source and destination. Transfers between two desktop computers on a local network are much faster than transfers from a local network to the AWS instance; on the other hand, transfers between a commercial and research cloud are incredibly fast.

We apply simple regression models to predict the relation between size and transfer rate for each pair of source and destination transfer groups. Our model predictions are shown in the dotted lines in Fig. 3. We see that we can predict transfer times in our testbed reasonably well.

Cold start predictors estimate one-off startup costs. As mentioned in Section II-A3, we treat the cold start process as a sequence of three consecutive steps: node acquisition, container instantiation, and package loading.

Node acquisition times are frequently both long and hard to predict, particularly on HPC clusters where queue times can depend on many factors—capacity, resources requested, and job-scheduling policies—and differ by several orders of magnitude [16]. One solution is to maintain a collection of warm nodes at all times. For our work, we assume that our

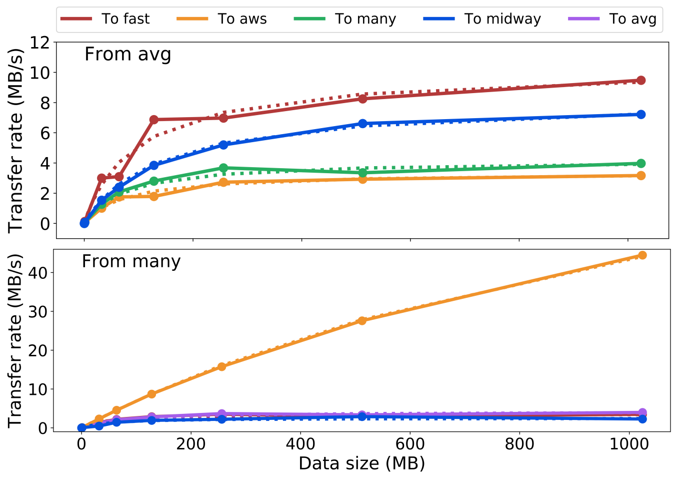

Fig. 3: Data-transfer rates between Globus endpoints in our testbed (see Section IV-A): specifically, from endpoints **avg** (a desktop on the departmental network) and **many** (at Argonne National Laboratory) to those endpoints and also **fast** (another department desktop), **aws** (an AWS instance in Virginia), and **midway** (a cluster in UChicago's Research Computing Center). The circles represent medians of measured values and our model's predictions are shown by the dotted lines.

endpoints either need not wait for node allocation (e.g., they are dedicated edge devices or desktop computers) or, if in HPC clusters, they are already running on warm nodes.

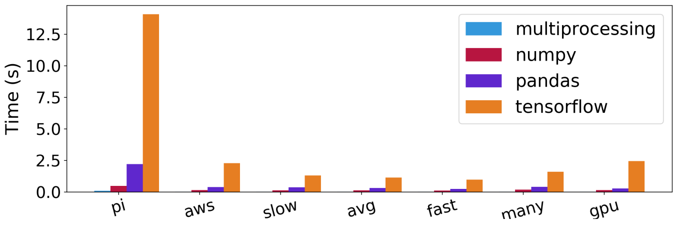

Fig. 4: Import latencies for four common Python packages on different devices. Download and installation latencies vary similarly.

Dependency loading involves downloading, installing, and importing packages. Fig. 4 shows the times to import common Python packages on the different devices in our testbed. These are easily predictable quantities, so accounting for these package-loading latencies is a matter of calculating the tradeoff between matching installed packages with application dependencies and incurring package loading overheads where a dependency is missing.

F. Transfer Manager Implementation

A user specifies, when registering a funcX endpoint, its associated Globus endpoint. When a new task is submitted to the Delta service with a list of files required for execution, the `TransferManager` uses the Globus API (with delegated Globus Auth tokens for the requesting user) to orchestrate the transfer. The `TransferManager` monitors the transfer and dispatches the task to the endpoint when the transfer has completed. This just-in-time submission allows the endpoint to execute other tasks while waiting for transfer overheads.

G. Scheduling

We evaluate three scheduling algorithms in this paper:

- `round-robin` routes each task to the next endpoint in a list of endpoints.
- `fastest-endpoint` routes each task to the endpoint for which our runtime predictor yields the shortest estimated function runtime, $t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}, \mathsf{data})$.
- `smallest-EET` routes each task to an endpoint selected according to Equation 1.

Resource selection in a heterogeneous environment requires knowledge of how different tasks will perform on different machines. If such knowledge is not initially available, then exploratory runs must be performed: a classic explore-exploit tradeoff [17]. We have explored such tradeoffs elsewhere [18]. Here, we implement a first exploration phase in `fastest-endpoint` and `smallest-EET` in which one task is sent to each of the seven endpoint groups to collect performance data, after which tasks are dispatched based on predictions from models trained with those data.

IV. Evaluation

We use **micro experiments** to evaluate to what extent the various Delta components (e.g., ability to learn function runtime and data transfer time) work for holistic computing ecosystems, and **macro experiments** to evaluate to what extent Delta, as a whole, yields satisfactory performance when put under load with realistic workloads.

A. Experimental Setting

We performed experiments on a heterogeneous testbed with three Raspberry Pis, four nodes in the Chameleon research cloud (two CPU-optimized, two GPU-optimized), an AWS EC2 instance, and three desktop computers: see Table I.

For both micro and macro experiments, we used four benchmark applications, each with a parameter n (and for MatMul, also a parameter m) that can be set in an experiment: 1) **Counter**, which uses a single thread to increment an integer value n times; 2) **ParCount** (parallel counter), a happily parallel computation that runs c instances of Counter, one on each of a resource's c cores, each counting to n/c; 3) **MatMul**, which uses TensorFlow to perform m multiplications of different $n \times n$ matrices; and 4) **FileIO**, which writes and reads n bytes to/from disk.

B. Micro Experiments

We first present three experiments to highlight different features of Delta and verify that Delta can transparently route tasks to different endpoints across the computing continuum. The first experiment demonstrates that, when asked to run a workload that favors a particular endpoint, Delta quickly learns to send this workload to this endpoint. The second experiment demonstrates that, when tasks involve data transfers, Delta accounts for the cost of data movement, thus making smarter decisions than baseline strategies. The third experiment demonstrates that, when faced with endpoint failures and slowdowns, Delta quickly recovers and completes tasks.

TABLE I: The heterogeneous testbed used for experiments included 11 devices.

Description	Hardware	Name	Count
Edge device (Raspberry Pi 3B)	ARM Cortex-A53, 4-core, 1GB	pi	3
Slow desktop (UChicago CS)	Intel Core i7-3770, 8-core, 8GB	slow	1
Average desktop (UChicago CS)	Intel Core i7-6700, 8-core, 8GB	avg	1
Fast desktop (UChicago CS)	Intel Core i7-8700, 12-core, 16GB	fast	1
Cloud CPU (AWS EC2 T3a.medium)	AMD EPYC 7571, 2-core, 4GB	aws	1
Manycore CPU (Chameleon)	Intel Xeon E5-2670, 48-core, 125GB	many	2
GPU node (Chameleon)	Nvidia Quadro RTX 6000 GPU (Intel Xeon Gold 6126, 48-core, 187GB)	gpu	2

1) Learning Under a Small Load: A simple situation in which to observe the benefits of heterogeneity is when tasks arrive infrequently enough that the only factor affecting EET is the function's runtime, t_{run}. Fig. 5 shows the execution times observed when 100 MatMul tasks, each with $m = 50$ and $n = 1000$, arrive at 0.5 Hz: an approximation to an application that regularly requests a neural-network inference.

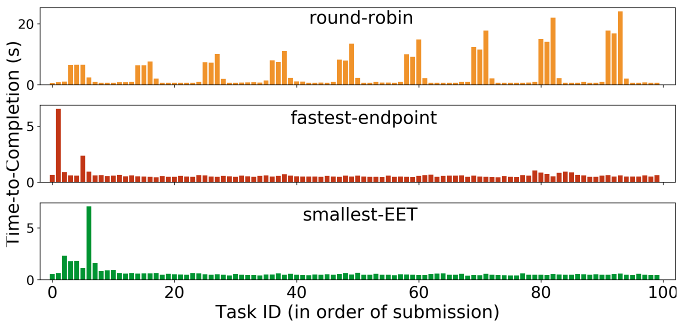

Fig. 5: Learning function performance behavior with different strategies under a small load. Note different scale for `round-robin`.

Both the `fastest-endpoint` and `smallest-EET` strategies use the first few tasks to explore the different endpoint groups. Both strategies then favor the most suitable endpoints for MatMul tasks, which unsurprisingly are the GPU endpoints. The naive `round-robin` strategy performs noticeably worse, and as the slowest endpoints cannot keep up with the arrival rate of tasks, `round-robin` execution times become increasingly slow. While the `fastest-endpoint` strategy performs optimally in this simple case, we see below that it does not in others.

2) Data Transfer Trade-offs: To evaluate the effects of data transfers, we conducted an experiment involving a series of Counter tasks, each with n selected randomly from $\{2^{24}, 2^{26}, 2^{28}\}$, and with each also specified to require two 1KB files located on the `avg` endpoint. As Delta does not cache files, both files must be transferred once for each task that is run on an endpoint other than `avg`; as this requires a request to the cloud-hosted Globus service, there is a non-trivial cost. Thus, for every execution, the Delta service must consider both execution and transfer times to predict the lowest overall EET. We did not use the three `pi` endpoints in this experiment, as they lacked Globus support; thus, we had eight endpoints and six endpoint groups. After 18 exploration tasks (one per size for each endpoint group) used by Delta to learn function performance behavior, we requested a total of 100 tasks at 0.5 Hz.

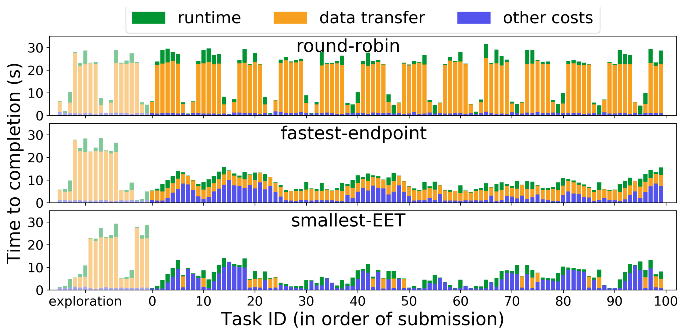

Fig. 6: Accounting for data transfer cost as part of function execution. The workload is 100 Counter tasks; the figure distinguishes between runtime costs, data transfer costs, and all other costs, with the lattermost including time spent waiting for resources.

Fig. 6 shows that the `round-robin` strategy naively cycles through the available endpoints, incurring a data transfer cost for every endpoint except the one that holds the required files. The `fastest-endpoint` strategy also ignores transfer costs and thus simply chooses the endpoint that has historically run the task the fastest. Since the files were located on the `avg` endpoint, which has slower CPU cores than three of the other endpoints, the `fastest-endpoint` strategy always chooses to offload the computation to a different endpoint, incurring non-negligible transfer costs. Finally, the `smallest-EET` strategy, which considers both transfer costs and runtime costs, only offloads the computation to a different endpoint (incurring transfer costs) some of the time. Closer analysis of the figure reveals the following pattern: the `avg` endpoint was chosen for execution several times, as indicated by the lack of transfer cost, until enough pending tasks accumulated (shown by the growing size of the blue bars, which include wait time), at which point it was best to offload the next task to a different endpoint and incur transfer costs.

3) Tolerating Failures and Slowdowns: Here a stream of MatMul tasks with $m = 50$ and $n = 2500$ arrives at 0.33 Hz. To simulate a failure, we removed the `gpu-1` endpoint after the 30th task and restarted it after the 60th task.

Fig. 7 shows how after a first exploratory phase, execution times are consistently small. Since the period from task 30 to task 60 is small, Delta does *not* observe a missed heartbeat from the `gpu-1` endpoint. However, it does see delays, and when it has not received results in more than twice the expected execution time, it sends backup tasks to other (slower) endpoints. After task 60, `gpu-1` resumes responding to task requests as expected and so execution returns to normal.

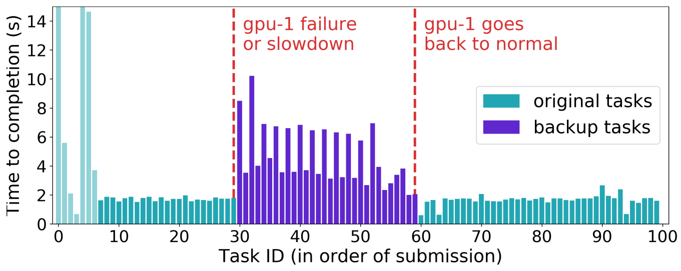

Fig. 7: Tolerating failure of endpoints and slowdown of tasks with automatic delay-detection and backup tasks.

C. Macro Experiments

These experiments evaluate Delta's performance when subjected to high loads with tasks being scheduled in large batches (frequent on FaaS platforms). The first experiment shows how throughput and time-to-completion vary when many copies of the same task are run. The second shows how input size must be taken into account when making scheduling decisions. The third experiment shows that Delta performs well when subjected to multiple task types at the same time.

1) Overloading Tasks: To test if Delta's scheduling system can distribute tasks across endpoints in a way that maximizes task throughput while maintaining low execution times, we ran 500 ParCount tasks with $n = 10^8$, in batches of 50.

Fig. 8 shows that the `round-robin` strategy struggles to keep the average execution time of tasks low, since it simply cycles through endpoints and inevitably hits slowdowns when it sends tasks to the slower endpoints. While the `fastest-endpoint` strategy provides a slightly better task throughput than `round-robin`, its median task execution time is, in fact, *higher* than that of `round-robin`. This is because whereas the `round-robin` strategy naively balances load amongst the different endpoints, the `fastest-endpoint` strategy sends *all* tasks to the endpoint which provides the fastest runtime for the function in concern. As tasks are executed FIFO, this leads to tasks quickly piling up on this endpoint, which explains the higher median time-to-completion. The `smallest-EET` strategy does not suffer from such ailments since it takes into account pending-task predictions for each endpoint. We see that, after the first batch (in which it explores all endpoint groups), the `smallest-EET` strategy consistently keeps the time-to-completion for tasks low and demonstrates a task throughput that is several times higher than the other strategies.

Given a goal of explainability, we repeated the experiment above but with the simple Counter function with $n = 10^8$. Fig. 9 shows the distribution of tasks per endpoint produced by the `smallest-EET` strategy in this case. The figure also shows the relative speeds with which each endpoint can run this function—this is essentially just a measure of CPU clock-speed. Ignoring other execution costs, the optimal distribution

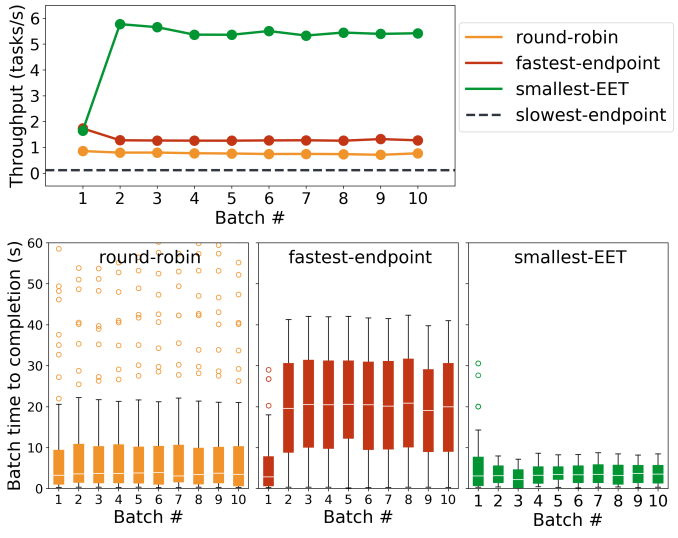

Fig. 8: Overloading Delta with ParCount tasks shows that, compared to baseline strategies, EET prediction significantly reduces time-to-completion and boosts throughput. The top figure reports the throughput of each batch. The bottom figure reports the box plot of task time-to-completion within each batch for each strategy. The box shows the first and last quartile; whiskers are shown at 1.5 interquartile range.

for this task should match the relative speeds of the different endpoints. Fig. 9 demonstrates that the `smallest-EET` strategy achieves a close approximation of this distribution, suggesting that its scheduling decisions are close to optimal.

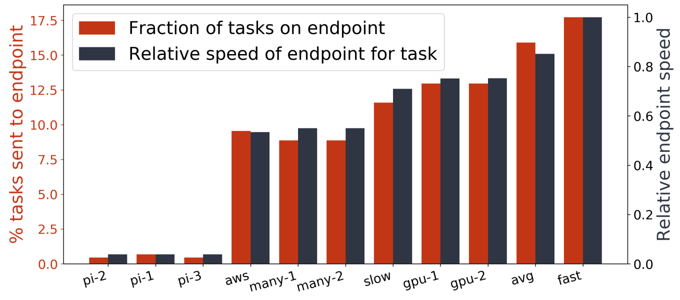

Fig. 9: In red, fraction of tasks sent to each endpoint when Delta is overloaded with Counter tasks; in black, relative endpoint speed for a Counter function.

2) Input Size Trade-offs: The previous experiment demonstrated how tasks should be scheduled across heterogeneous endpoints when all tasks are uniform. We now consider the case where tasks are non-uniform, specifically, when different function inputs result in different runtimes: a common occurrence seen in many workloads. We use Counter tasks with $n \in \{2^{22}, 2^{25}, 2^{28}\}$ and schedule a total of 600 tasks, divided into 10 batches of 60 tasks each. Each batch consisted of 20 tasks of each of the three input sizes, randomly ordered.

The results in Fig. 10 show that the `round-robin` strategy, by ignoring all features of incoming tasks and available endpoints, performs poorly and yields low throughput. The variance in the observed throughput between batches is because of the randomness in the order of tasks. The `fastest-endpoint` strategy, similar to the previous experiment, overwhelms the one endpoint which has the fastest

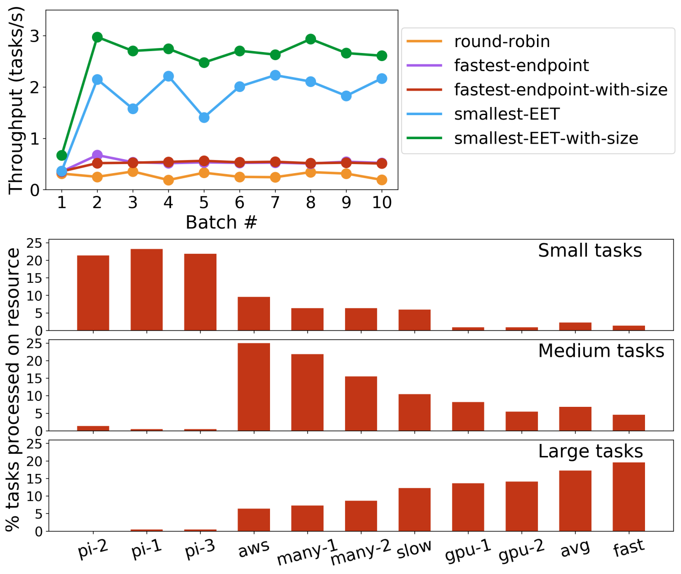

Fig. 10: Running tasks with multiple input sizes shows that treating different input sizes differently yields increases in performance. When under load, it is beneficial to send smaller tasks to slower endpoints, even Raspberry Pis. The top figure reports throughput when the `fastest-endpoint` and `smallest-EET` consider ("with-size") and do not consider input size in runtime predictions. The bottom figures show how the `smallest-EET` strategy allocates different sizes of tasks to endpoints.

runtime for the function. This is because this fastest endpoint, unsurprisingly, is the best for *each* of the three sizes. The figure also shows how the `smallest-EET` strategy performs, with and without taking into account input size in its runtime predictions. Without considering input size, the `smallest-EET` strategy has a high variance in the observed throughput. This is because it indiscriminately sends tasks to where the function has been performing well recently. Occasionally, it sends tasks to the correct endpoints but, other times, it sends tasks to endpoints which have previously offered short runtimes due to the fact they were allocated tasks with small input sizes. When the `smallest-EET` strategy takes into account task sizes, its throughput remains consistently high.

Fig. 10 also shows how tasks are allocated to endpoints under the `smallest-EET` strategy. We see that tasks are distributed across the endpoints: small tasks are sent to the slowest endpoints, medium tasks to the mediocre endpoints, and the largest tasks to the fastest endpoints. We glean from this experiment that when put under load with tasks of different sizes, we can and must take advantage of even the slowest endpoints for improved performance.

3) Multiple Heterogeneous Tasks: This final experiment seeks to emulate real-world FaaS behavior via a workload involving three functions and, for each, three problem sizes: MatMul with $m = 20$ and $n \in \{2^8, 2^9, 2^{10}\}$; ParCount for $n \in \{2^{24}, 2^{26}, 2^{28}\}$; and FileI/O for $n \in \{2^{20}, 2^{22}, 2^{24}\}$. We assembled 111 of each of these nine {function, size} pairs, for a total of 999 tasks. These tasks were randomly shuffled and then dispatched in batches of 100, with each batch starting only when the previous batch had completed. Fig. 11 shows the throughput observed when using different scheduling strategies. We used the input size predictor described in Section III-E in all cases.

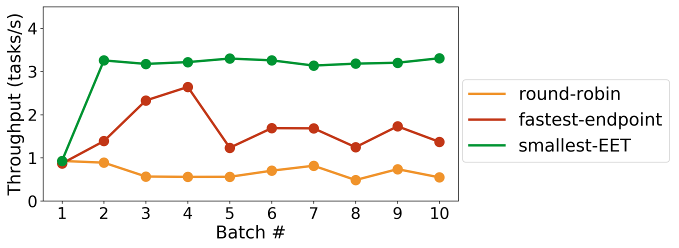

Fig. 11: Throughput achieved by different scheduling strategies for the Multiple Heterogeneous Tasks experiment.

Unsurprisingly, `round-robin` does not distribute tasks efficiently to different endpoints. `fastest-endpoint` learns to send each function to the endpoint on which it runs fastest, but as it fails to account for any other factors, it shows high variance in throughput, getting lucky in some batches and overwhelming a few endpoints in others. `smallest-EET` quickly learns to allocate tasks to suitable endpoints and maintains a consistently high throughput. Fig. 12 shows how each strategy allocates tasks to endpoints. `round-robin` of course distributes tasks evenly across the endpoints (the slightly uneven peaks are due to the random order of tasks submitted). `fastest-endpoint` overwhelms the handful of endpoints that it determines to be the fastest for each task type, sending most FileIO tasks to `fast-desktop` (fastest CPU and fast disk) and most MatMul tasks to the GPUs. On the other hand, the distribution observed for `smallest-EET` is more nuanced, depending on both task types and sizes. The smallest MatMul tasks are sent almost exclusively to the Raspberry Pis; longer-running task are sent to the GPU and manycore endpoints, where Tensorflow can exploit massive parallelism. Similarly, larger ParCount tasks are sent to the manycore endpoints and larger FileIO tasks to endpoints with fast disks and fast CPUs, whereas the smaller tasks are distributed amongst slower endpoints.

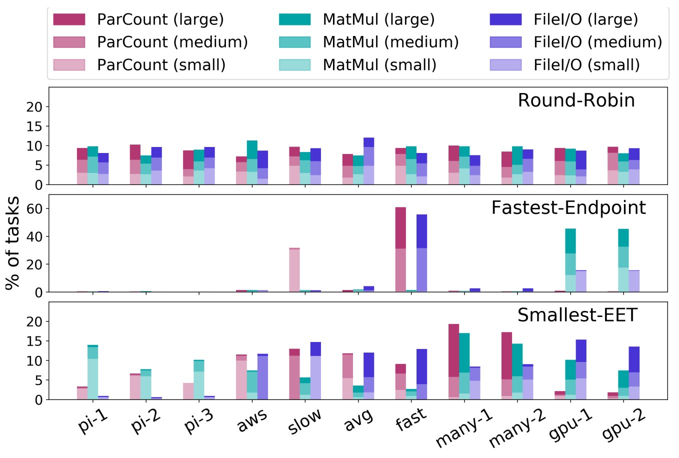

Fig. 12: Distribution of tasks across endpoints for the Multiple Heterogeneous Tasks experiment. Note different scale for `fastest-endpoint`.

TABLE II: Statistics for Multiple Heterogeneous Tasks experiment: average completion time for each task type, and average makespan both across across all batches and across only non-exploration phases. All times are in seconds.

Strategy	ParCount			MatMul			FileI/O			Avg Makespan	Avg Makespan Post-Exploration
	Small	Medium	Large	Small	Medium	Large	Small	Medium	Large		
Round-Robin	28.7	30.2	41.8	30.6	26.8	32.6	30.1	32.6	42.4	154.2	159.4
Fastest-Endpoint	19.4	34.6	31.6	20.7	19.5	19.9	20.2	34.2	40.3	67.9	62.6
Smallest-EET	23.4	24.9	23.9	25.3	23.7	24.6	24.3	24.4	27.0	38.7	31.0

Table II shows the time to complete each task as an average across batches. It also shows the average makespan per batch, both across all batches and when ignoring batches in the exploratory phase. We see that `smallest-EET` achieves the fastest average completion time for only four of the nine {function, size} pairs, while `fastest-endpoint` gives the fastest average completion time for the other five. This result is not surprising as `smallest-EET` takes a more holistic view and spreads workload across the testbed. This distribution is further highlighted by the fact that the average makespan for `smallest-EET` is less than half that achieved as `fastest-endpoint` and five times better than `round-robin`, when the exploration phase is excluded.

These results underscore our claim that scheduling task executions in a heterogeneous environment is a non-trivial undertaking that requires modeling the many complexities involved in running a computation remotely—and that doing so can yield considerable benefits.

V. Related Work

This work builds upon decades of foundational research in distributed computing, including in shared clusters, grid [19] and peer-to-peer [20] computing, cloud computing, scheduling [21], and programming heterogeneous devices [22].

Federated computing: The desire to federate disparate computing resources is not new and has motivated the development of grid [19] and cloud computing. Seminal work focused here on developing the foundation for remote, federated computing with middleware to support job submission, data transfer, and federated security. Cloud computing extended this foundation, providing virtualized access to resources hosted by a single provider. Cloud platforms now offer hundreds of different instance types, and increasingly specialized hardware such as cloud TPUs. In the context of grid and cloud computing, there has been widespread investigation of heuristic-based strategies for heterogeneous resources, including completion time minimization [21], greedy scheduling through profiling and execution histories [23, 24], data-computation scheduling [25], and use of computational economies [26]. Our approach here builds on this prior work, adapts it to increasingly heterogeneous and distributed computing environments, and addresses scheduling of lightweight programming functions.

Workflows: Considerable research has focused on scheduling workflows on distributed resources. While workflows have traditionally focused on orchestrating jobs, for example using Pegasus [27], new workflow models, such as Parsl [12], orchestrate collections of loosely-coupled functions. With the rapid adoption of machine learning, there are also systems, such as DLHub [28, 29], incorporating inferences of machine learning models into workflows. Researchers have explored various approaches for scheduling workflows (e.g., scientific and big data processing MapReduce) in grid [30, 31] and cloud [32–35] environments. Unlike these papers that focused on orchestrating and scheduling workflows on a single environment, we propose a model for function-level execution across heterogeneous and distributed computing environments, which makes a first attempt towards the computing continuum.

Function-as-a-service: Public cloud provider FaaS capabilities [10, 36, 37] are deployed on homogeneous cloud infrastructure, with the exception of Amazon Greengrass that can be deployed locally for IoT use cases [38]. Open-source FaaS solutions, such as OpenLambda [39], Apache Openwhisk [11], and Kubeless [40] enable FaaS deployments on local resources, often using Kubernetes clusters for container provisioning. Our approaches could be applied to these various open-source FaaS deployments, and perhaps even cloud FaaS platforms if they expose access to heterogeneous resources in the future. Recent FaaS research has focused on challenges associated with centralized deployments and on reducing cold-start times [8, 9]. For example, Azure shows cold-start latencies of up to 3500ms, whereas AWS uses optimized Firecracker virtualization [41], and likely maintains warm virtual machines, to hide some cold-start costs.

FaaS scheduling: While we are not aware of any prior work on building a complete model of function scheduling, optimizing different aspects of function execution has been a major focus in recent FaaS research. FnSched [42] seeks to maximize utilization while meeting service-level objectives (SLOs) by regulating function resource usage and scaling utilization of resources based on load. Other work on meeting SLOs in heterogeneous computing environments includes a probabilistic *task pruning* method [43], which uses a function's execution history to predict whether a task will meet its SLO. Wukong implements a decentralized model for scheduling functions expressed in a workflow DAG on FaaS platforms [44]. Additionally, Maheswaran et al. [45] defined significant groundwork in heterogeneous task allocation.

VI. Conclusion and Future Work

We have presented Delta, a system unifying heterogeneous computing environments and managing the flow of diverse function execution requests to endpoints with different connectivity and computational capabilities. Delta uses dynamically updated predictive models to determine the best destination for a particular function at a particular time.

Delta learns to predict not only how fast different functions execute across the computing environment, but also generalizeable execution characteristics of that environment.

We showed that leveraging Delta to execute a workload in a heterogeneous environment can more than halve makespan when compared to simply selecting the fastest available compute resources, and reduce makespan by a factor of five when compared to a round robin strategy.

We are now working to refine this foundation and framework in several directions, namely, to understand the specific requirements of scientific use cases and enhancing Delta to consider diverse and complex notions of "cost," in terms of both monetary and time values. To account for complexities in real world considerations of computational tradeoffs, we will explore using learned embeddings of cost and execution characteristics. We also plan on a performance scaling study of Delta in diverse computing environments.

Acknowledgment

We thank Alex Foster, Troy Hu, and Wendy Li for their efforts exploring trade-offs in the computing continuum. This research was supported in part by DOE contract DE-AC02-06CH11357 and by NSF grants 1816611, 2004894, and 1550588. We used computing facilities provided by NSF Chameleon [7] and the UChicago Research Computing Center.

References

[1] D. Balouek-Thomert *et al.*, "Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows," *Intl J. High Performance Computing Applications*, vol. 33, no. 6, pp. 1159–1174, 2019.
[2] P. Beckman *et al.*, "Harnessing the computing continuum for programming our world," in *Fog Computing: Theory and Practice*. Wiley Online Library, 2020, pp. 215–230.
[3] G. P. Agrawal, "Optical communication: Its history and recent progress," in *Optics in Our Time*, 2016, pp. 177–199.
[4] E. Jonas *et al.*, "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.
[5] R. Chard *et al.*, "funcX: A federated function serving fabric for science," in *29th Intl Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020.
[6] K. Chard *et al.*, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, 2014.
[7] K. Keahey *et al.*, "Lessons learned from the Chameleon testbed," in *USENIX Annual Technical Conference*. USENIX Association, 2020.
[8] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference*, 2018.
[9] L. Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *USENIX Annual Technical Conference*, 2018, pp. 133–146.
[10] "Amazon Lambda," aws.amazon.com/lambda/, Seen: 10/2020.
[11] "Apache OpenWhisk." http://openwhisk.apache.org/, Seen: 10/2020.
[12] Y. Babuji *et al.*, "Parsl: Pervasive parallel programming in Python," in *28th Intl Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.
[13] B. Allen *et al.*, "Software as a service for data scientists," *CACM*, vol. 55, no. 2, pp. 81–88, 2012.
[14] M. Baughman *et al.*, "Profiling and predicting application performance on the cloud," in *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2018.
[15] R. Chard *et al.*, "Cost-aware cloud profiling, prediction, and provisioning as a service," *IEEE Cloud Computing*, vol. 4, no. 4, pp. 48–59, 2017.
[16] W. Smith *et al.*, "Predicting application run times with historical information," *J. Parallel & Distributed Computing*, vol. 64, no. 9, 2004.
[17] P. Maes *et al.*, "Explore/exploit strategies in autonomy," in *4th Intl Conf. on Simulation of Adaptive Behavior*, vol. 4, 1996, pp. 325–332.
[18] C. Wu *et al.*, "ParaOpt: Automated application parameterization and optimization for the cloud," in *Intl Conference on Cloud Computing Technology and Science*. IEEE, 2019, pp. 255–262.
[19] I. Foster *et al.*, "The anatomy of the grid: Enabling scalable virtual organizations," *The Intl Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.
[20] D. S. Milojicic *et al.*, "Peer-to-peer computing," 2002, Technical Report HPL-2002-57, HP Labs.
[21] H. Topcuoglu *et al.*, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.
[22] S. Mittal *et al.*, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–35, 2015.
[23] C. Gregg *et al.*, "Dynamic heterogeneous scheduling decisions using historical runtime data," in *Workshop on Applications for Multi-and Many-Core Processors*, 2011, pp. 1–12.
[24] C. Delimitrou *et al.*, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.
[25] K. Ranganathan *et al.*, "Decoupling computation and data scheduling in distributed data-intensive applications," in *11th IEEE Intl Symposium on High Performance Distributed Computing*, 2002, pp. 352–358.
[26] K. Chard *et al.*, "High occupancy resource allocation for grid and cloud systems, a study with DRIVE," in *19th ACM Intl Symposium on High Performance Distributed Computing*, 2010, pp. 73–84.
[27] E. Deelman *et al.*, "Pegasus, a workflow management system for science automation," *Future Gen. Computer Sys.*, vol. 46, pp. 17 – 35, 2015.
[28] R. Chard *et al.*, "Dlhub: Model and data serving for science," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 283–292.
[29] Z. Li *et al.*, "Dlhub: Simplifying publication, discovery, and use of machine learning models in science," *Journal of Parallel and Distributed Computing*, vol. 147, pp. 64–76, 2021.
[30] V. Hamscher *et al.*, "Evaluation of job-scheduling strategies for grid computing," in *Intl Workshop on Grid Computing*, 2000.
[31] R. Buyya *et al.*, "Economic models for resource management and scheduling in grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, pp. 1507–1542, 2002.
[32] F. Wu *et al.*, "Workflow scheduling in cloud: A survey," *The Journal of Supercomputing*, vol. 71, no. 9, pp. 3373–3418, May 2015.
[33] Z. Zhu *et al.*, "Evolutionary multi-objective workflow scheduling in cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1344–1357, 2015.
[34] M. Zaharia *et al.*, "Improving MapReduce performance in heterogeneous environments." in *Osdi*, vol. 8, no. 4, 2008, p. 7.
[35] Z. Li *et al.*, "An exploration of designing a hybrid scale-up/out Hadoop architecture based on performance measurements," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 386–400, 2016.
[36] "GCP Functions." https://cloud.google.com/functions/, Seen: 10/2020.
[37] "Azure Functions." https://azure.microsoft.com/en-us/services/functions/, Seen: 10/2020.
[38] "Amazon Greengrass," aws.amazon.com/greengrass/, Seen: 10/2020.
[39] S. Hendrickson *et al.*, "Serverless computation with OpenLambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing*, 2016.
[40] "Kubeless." https://kubeless.io/, Seen: 10/2020.
[41] "Firecracker," firecracker-microvm.github.io/, Seen: 10/2020.
[42] A. Suresh *et al.*, "FnSched: An efficient scheduler for serverless functions," in *5th Intl Workshop on Serverless Computing*, 2019, pp. 19–24.
[43] C. Denninnart *et al.*, "Improving robustness of heterogeneous serverless computing systems via probabilistic task pruning," in *Intl Parallel and Distributed Processing Symposium Workshops*, 2019, pp. 6–15.
[44] B. Carver *et al.*, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.
[45] M. Maheswaran *et al.*, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *Journal of parallel and distributed computing*, vol. 59, no. 2, pp. 107–131, 1999.