THE UNIVERSITY OF CHICAGO

CODING THE COMPUTING CONTINUUM: ENABLING FLUID EXECUTION OF FUNCTIONS IN HETEROGENEOUS COMPUTING ENVIRONMENTS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
ROHAN KUMAR

CHICAGO, ILLINOIS
JULY 2020

Table of Contents

ACKNOWLEDGMENTS

This project would not be what it is without the support, counsel, and help of many incredible people I have had the good fortune of knowing.

First and foremost, I thank my advisors, Dr Kyle Chard and Dr Ian Foster, for their constant encouragement, guidance, and mentorship, both during this thesis project and during the rest of my time at UChicago. Thank you for being visionaries of unparalleled fortitude, and for pushing me in all the right ways. I only hope that I was able to do justice to the work we set out to do.

I thank Dr Raul Castro Fernandez for serving on my thesis committee. I am grateful for your unbroken dedication to asking the important questions, and for your singular ability to provide criticism with kindness.

I owe a great deal of gratitude to the members of Globus Labs. I thank Dr Ryan Chard for being a fantastic sounding board, and for always being ready to provide assistance. I thank Dr Zhuozhao Li for always being reliable, and for not being afraid to speak his mind. I thank Yadu Babuji for dreaming big, for insisting on quality, and for never being too busy to help. I thank Tyler Skluzacek for his lightheartedness, and for his friendship from the very beginning. I thank Matt Baughman for his shrewd insights, and for his knack for selling any idea with flair and elegance.

I also thank Troy Hu and Wendy Li for their help in creating workloads to evaluate parts of this work, and Alex Foster for his investigation of queue wait times.

Finally, I thank my family and my friends, who constantly reminded me of the light at the end of the tunnel. This past year has taught me many things, among which is how unbelievably lucky I am to be surrounded by such loving and thoughtful people.

I am grateful to you all for being so wonderfully supportive, and for making this year-long journey infinitely memorable. Your kindness means everything to me.

ABSTRACT

With network speeds faster than they have ever been before, the once prominent distinction between local and remote execution is now rapidly disappearing. This has allowed users to begin exploring methods of running their analyses on remote devices that are not only more powerful, but often more suited to their analyses. No longer are users constrained by the limited capabilities of their local devices. Instead, with little effort, they can now access high-performance clouds and clusters, where their execution needs can be more than satisfied. As such, the world is experiencing a shift away from general-purpose devices that can be used for diverse applications. The modern age of computing is one of specialization, with special-purpose hardware being carefully optimized for particular applications. In such an environment of ubiquitous heterogeneity, there is an increasing need to automate the selection of hardware for arbitrary computations. However, the question of where to run a computation is full of complexities, and depends on a multidimensional array of factors, including task performance, communication costs, and resource-provisioning delays. The goal of this work is to develop techniques to model these different factors as part of a single comprehensive framework for heterogeneous execution. To accomplish this goal, we build upon existing work on function-serving platforms, since functions represent a convenient abstraction for remote computation. Our contributions in this work are threefold. First, we demonstrate that there are inescapable trade-offs between compute times, network speeds, start-up delays, and other latencies, which must be accounted for in the choice of hardware and location for computations. Second, we present *Delta*, a framework for heterogeneous function execution that is designed to account for these trade-offs, and provides improved performance by taking advantage of the heterogeneity in compute resources. Finally, we define a research agenda in this space, and motivate future work with the intention of building a world where fluid computation can exist.

CHAPTER 1

INTRODUCTION

The last decade of advances in computing has brought about a revolution in how we think about computation. With the end of Moore's law and an inherent limitation in scaling up general-purpose CPUs, the world is rapidly moving towards the natural next step in improving application performance — specialization. That is, instead of being limited by devices that are made for general use, users are developing specialized hardware that is optimized for their particular applications. Hardware specialization is quickly becoming not only the most efficient, but also the cheapest method of meeting performance requirements. Further, with network speeds increasing at a dramatic rate [1], it has never been easier for users to offload their analyses from their local desktops to remote machines that are more powerful, and often more suited to their specific computations.

Being able to offload computations in this way requires us to meet two distinct but complementary objectives. First, we need sophisticated infrastructure to handle remote task execution in a reliable and efficient manner. This objective is met, at least in part, by the advent of modern function-serving platforms such as *funcX* [2], which provide convenient abstractions for executing function calls on remote machines. Second, we need the ability to determine which remote machines are best suited for given computations — making this choice, quickly and reliably, is the focus of this work.

As the heterogeneity of devices grows, the choice of where to run a particular computation only becomes more complex. Selecting a destination for a computation is not simply a matter of choosing the fastest device, or even the closest one. Instead, the true "speed" of a device for a given application depends on a *combination* of factors, including the compute time, the network latency to the device, the time to provision a suitable compute environment, and the delay of transferring data. This fundamental idea, that the speed of computation depends not on one single cost, but on an amalgamation of seemingly orthogonal costs, has

been referred to as the *computing continuum* [3].

Consider, for example, a high-energy physics application that collects data at the Fermi National Accelerator Laboratory, and needs to run trigger analysis on it, regularly and with low latency [4]. It can either run this analysis on a local CPU, which would take 2 seconds, or it can offload this analysis to an Field Programmable Gate Array (FPGA) that is optimized for this task and can run this computation in 30ms. Suppose this FPGA is located in Amazon Web Service's cloud in Virginia, which with speed-of-light communication, is a network latency of 10ms away. Running the trigger analysis on this FPGA would provide a total execution time of 50ms, which is 40 times faster than running on the local CPU. There is no dearth of such high-performance applications which would benefit from the correct choice of execution device, thereby reinforcing the need to build a system that can autonomously learn to make such choices.

Perhaps unsurprisingly, such trade-off analysis can be extended beyond the two variables of compute time and network latency. Often, computations require substantial amounts of data that must first be transferred to the compute location, which bears additional overheads. Moreover, compute resources are often shared between multiple applications, and sometimes even multiple users, which introduces further delays. Our goal in this work is to account for these multidimensional trade-offs, and build a new model of computing that allows computations to be run on the devices that are most effective for them.

To capture the computing continuum, we build upon existing Function-as-a-Service (FaaS) infrastructure, since the FaaS model encapsulates the idea of abstracting remote computation. We simply borrow this model of remote computation as a way of thinking about computing in a federated environment. The emergence of FaaS has transformed how users think about computation, by allowing them to work at the function abstraction level, thus avoiding the need to manually deploy any execution infrastructure. Much in the same way, we would like our continuum model of computation to afford users the opportunity to

worry only about developing their applications, with all other aspects of running computations taken care of automatically. Moreover, working with modular functions, as opposed to monolithic applications, allows us to efficiently and reliably learn the runtime behavior of different computations on heterogeneous devices, while still allowing users to fully express the complexities of their tasks.

Existing FaaS frameworks are designed for one or more consumers to execute functions on a single, typically homogenous cloud. The task of realizing a fluid and federated computing model, however, requires more than anything the ability to run on a distributed ecosystem of heterogeneous devices, ideally on a single federated platform. Thus, our ideal testing ground is a FaaS framework that allows arbitrary devices to be connected as remote destinations for incoming computations. A FaaS framework that was developed with exactly this goal in mind is *funcX* [2]. Unlike most FaaS systems, *funcX* allows users to register devices of their choice as *endpoints* on which they can then execute functions. The bar for what is an eligible endpoint is quite low — if a machine can run Python, it can run *funcX*. This makes *funcX* an especially enticing choice for an execution fabric upon which the computing continuum can be built, since it allows us to easily connect a wide array of heterogeneous devices on a single federated grid.

Our contributions can be roughly categorized into three areas:

- First, we demonstrate the existence of ubiquitous trade-offs between latency, runtime, and data transfer costs in the context of function execution.
- Second, we present *Delta* — **D**istributed **E**xecution of **L**ambdas using **T**rade-off **A**nalysis — a proof-of-concept framework for heterogeneous function execution built on *funcX* that automatically learns to take advantage of these trade-offs, and yields improved performance by exploiting the heterogeneity in available devices.
- Third, we define a research agenda in this space and motivate directions for future work, with the hope of one day truly realizing the computing continuum.

The rest of this document is organized as follows. Chapter 2 provides a summary of related work. Chapter 3 outlines our guiding research questions, and formalizes the problem at hand. Chapter 4 describes various components of *Delta*, including details about its architecture. Chapter 5 uses experimental evidence to show how *Delta* performs on various realistic workloads. Finally, Chapter 6 concludes, and highlights directions for future work.

CHAPTER 2

RELATED WORK

Our work builds upon decades of foundational research in the domain of distributed computing, going back as far as the introduction of grid computing, and leading up to recent work in programming heterogeneous devices, as well as the rise of modern function-serving platforms.

2.1 Grid Computing

In the late 1990s and the early 2000s, there were efforts to create a new paradigm of distributed computing, one that enabled the sharing of computational resources on phenomenally large scales and across administrative domains, with sophisticated provisions for the dynamic and complex use-cases that real-world science often requires. This paradigm was referred to as *grid* computing [5]. The allure of grid computing arose from its bold foresight — a world where access to compute resources would be as common as access to electricity [6]. The development of such a *grid* of computational resources did not come without multidimensional challenges. These challenges included the development of methods for controlling different types of remote machines [7], standard protocols for communication between applications and their requested resources [8], as well as a trust fabric to ensure authenticated and authorized access to parts of the grid [9].

The grid is often thought of as a precursor to the familiar cloud computing paradigms that are widely available today, such as those offered by Amazon [10], Google [11], and Microsoft [12]. Modern cloud providers promise vast amounts of computation power to anyone willing to pay. Yet, in some senses, the notion of the grid is much more far-reaching than that of a centralized cloud [13]. The grid is inherently a distributed idea, and is thus not restricted to resources on any one cloud. Instead, the essence of the grid is in connecting

scattered resources — locally, in the cloud, in supercomputing clusters, and on the edge. In this sense, the grid is, at its core, defined by its potential for heterogeneity. Our work intends to reimagine this vision of the grid, by developing techniques that automatically take advantage of its intrinsic heterogeneity. While the original model of grid computing focused on stiching together resources in large units, our work aims to apply grid computing techniques at a more fine-grained level — i.e., to link together smaller individual computers with diverse capabilities in order to fluidly execute granular computations on them.

2.2 Heterogeneous Computing

Having embarked on a journey that involves using the diverse capabilities of heterogeneous devices, the natural next question is how to actually program across these devices. In cases where the heterogeneity is mainly in the parallelism scale available, several solutions exist. Popular data-parallel computation frameworks such as Hadoop [14] and Spark [15] allow users to write code to perform MapReduce-like computations, which can automatically scale to tens of thousands of nodes in high-performance clusters, and provide reliability in the face of failures. For more general-purpose parallelism, recent Python-based libraries such as Parsl [16] and Dask [17] aid users in performing parallel executions of arbitrary functions, with automatic scaling on clouds and clusters.

In addition, the last decade has seen an increasing number of efforts to allow users to write code that can execute not only on CPUs of different shapes and sizes, but on truly heterogeneous device architectures, such as GPUs, FPGAs, and ASICs. Ever since Nvidia's release of Compute Unified Device Architecture (CUDA) [18], which allows users to program GPUs for general-purpose uses, there have been new frameworks developed to simplify the lives of application developers. Ideally, a developer should only need to write code once and be able to run this code on an array of heterogeneous architectures. Initial work towards this goal was presented as OpenCL [19], which provides a unified template of writing code

for a variety of CPU, GPU, and ASIC designs. In the domain of deep learning, the popular libraries, Tensorflow [20] and PyTorch [21], allow users to write a single version of code for training and using deep neural networks that transparently transitions between using CPUs, GPUs, and in the case of Tensorflow, even TPUs [22]. Even more recent work has aimed towards extending these "write-once-execute-many" capabilities to more general-purpose computations. For instance, HeteroDoop [23] is a programming framework that extends MapReduce semantics to automatically make use of GPUs. Libraries such as Kokkos [24] and RAJA [25] provide a unified syntax for users to write C++ code that can seamlessly compile to a variety of different CPU and GPU architectures using carefully designed optimizations during compilation. While these unified programming capabilities are not strictly necessary for using heterogeneous devices, the convenience they provide significantly lowers the barrier to entry for developers to use specialized hardware.

Of course, being able to program heterogeneous devices is only one half of the battle. For our work, it is also essential to be able decide which type of device a computation should run on. Although there has not been much work on such decision-making in the context of function-serving, this problem has been explored to some depth in the more general context of high-performance computing. The problem of scheduling applications has long been known to be NP-complete, although efficient heuristic-based strategies have been proposed for heterogeneous settings. These strategies are often based on choosing resources which provide the earliest expected finish time for a task [26]. More recent work on heterogeneous scheduling [27, 28] has focused on using historical execution time of workloads on heterogeneous machines to create greedy scheduling strategies which try to maximize throughput and minimize application runtimes.

2.3 Function-as-a-Service

Function-as-a-Service (FaaS) is a relatively new paradigm of computing that allows its users to think at the abstraction level of functions. With a computing model based around (remote) function calls, FaaS users only need to write the code for their applications — all other aspects of execution are taken care of by the FaaS platform. Recent years have seen the introduction of FaaS capabilities from all popular cloud providers, including Amazon Lambda [10], Google Cloud Functions [11], and Microsoft Azure Functions [12]. Some of these also provide variations of their FaaS platforms that are more suited to distributed IoT use-cases, such as Amazon Greengrass [29]. As one would expect, each of these platforms provides support for a range of languages and "triggers", i.e., events that can cause a function execution to occur. With the exception of some remarkable investigative work [30], not much is known about the internal workings of these closed-source platforms.

Several open-source FaaS solutions have been developed, each aiming to solve different variants of function-serving problems. Some notable examples are as follows. The OpenLambda project [31] seeks to explore directions similar to Amazon Lambda, but in the open-source community. OpenLambda's research agenda includes investigating the best execution engines for serverless computation, supporting a wide range of languages and packages, and optimizing performance with improved database guarantees and load-balancing. Apache Openwhisk [32] provides a flexible model that allows "events" to be triggered by a spectrum of popular web-services, where each such trigger can be associated to (stateless) functions via user-defined rules. OpenWhisk provides support for deployment both locally and in the cloud, using multiple containerization options. OpenWhisk serves as the backbone of IBM Cloud Functions [33]. Kubeless [34] is a FaaS platform that is built for deployment on Kubernetes clusters, allowing users to take advantage of Kubernetes's container-orchestration mechanisms. Kubeless replicates the interface of Amazon Lambda, uses Apache Kafka for messaging, and allows users to group functions for more efficient resource usage.

2.4 Scheduling Functions

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. One of the most common issues that plagues FaaS platforms is *cold-start latency*, i.e., the initial overhead of setting up containers, dependencies, and executors when executing a function [35]. According to a recent investigation [30], all popularly used FaaS systems suffer from cold starts. For instance, Azure shows cold-start latencies of up to 3500ms, whereas AWS uses the optimized Firecracker virtualization [36] and likely maintains a set of "warm" virtual machines, thus hiding some cold-start costs. Often, cloud providers let allocated machines run idle for long periods (sometimes many hours), to account for the possibility of incoming function executions in the future. Other work has explored this cold-start problem from a different angle. The authors of SOCK [37] developed containers optimized for serverless execution with the goal of minimizing start-up costs. SOCK's virtualization techniques employ a minimal subset of the features of general-purpose containers like Docker [38], thus allowing for fast instantiation of containers. SOCK also uses *zygote* processes to run Python code with different package dependencies. By maintaining a collection of *zygote* processes with different packages imported, SOCK is able to minimize the initial import latencies of loading Python packages. With additional caching optimizations, SOCK is able to achieve 2-3x faster cold starts than AWS Lambda.

Recent work has also tried to optimize function executions by improving function colocation and resource usage. It has been discovered that some cloud providers (like AWS) prioritize colocating executions of the same function, whereas others (like Azure) do not [30]. SAND [39] is a FaaS system designed to decrease the latency of communication between functions of the same application and increase resource utilization. SAND achieves these goals by running functions within *application sandboxes*, so that functions of the same application are executed close to each other. This, coupled with a hierarchical mechanism for

communication between functions, allows SAND to provide low-latency chaining of function executions. FnSched [40] is a system that reduces resource costs by maximizing utilization while meeting service-level objectives (SLOs). FnSched meets these objectives by carefully regulating the resource usage of function executions, and making informed decisions about scaling up resources when the task load increases. More specifically, FnSched assigns *cpu-shares* to each executing container, which can be regulated to provide containers more or less resources in order to meet execution requirements. There has also been some work on meeting SLOs in heterogeneous computing environments. For instance, a probabilistic method of *task pruning* has been demonstrated [41], which involves using a function's execution history to predict whether a task will meet its SLO or not. In cases where a task is likely to be delayed, the task is dropped, allowing for other tasks to complete sooner.

While the results mentioned above tackle various aspects of efficient function scheduling, none of them provide a holistic model of function execution. However, such a model is necessary to fully account for all the different aspects of computation in a federated heterogeneous environment. Our contributions in this work include providing such a model and later applying it to build a framework for fluid function execution.

CHAPTER 3

PROBLEM DESCRIPTION

In this chapter, we briefly outline our research agenda, and introduce some formalism for thinking about the problem of choosing the best location to execute a computation. We then describe how we model a task's execution time, and list the assumptions we make.

3.1 Research Questions

We seek to answer the following questions in this work:

(1) How can we design a system that can transparently dispatch computations to heterogeneous computing resources?

(2) What information is required to make decisions about performance trade-offs between these resources? How can we capture this information and use it to place tasks on these resources?

(3) Can we understand what "good" scheduling decisions look like? Which features of realistic computations determine where they should be run?

3.2 Problem Formulation

Consider a set of endpoints, $\{\mathsf{EP}_1, \ldots, \mathsf{EP}_n\}$, that we can run functions on. We receive an incoming stream of tasks of the form $(\mathsf{f}, \mathsf{x}, \mathsf{data})$ where f is a function (that has been previously registered with *Delta*), x is a function input, and data is a list of files (on the requesting machine or elsewhere) that are needed to perform the computation. As we will see, this formulation of computation is simple enough to lend itself to predictive analysis, yet complete enough to model the execution and data needs of real-world computations.

There are many possible objectives we could try to optimize for when choosing where to run these tasks (e.g., minimizing data movement, maximizing resource utilization, etc.) —

this is simply a matter of policy. In this work, we focus on scheduling tasks across our n heterogeneous endpoints while minimizing their *time-to-completion*.

3.3 Modeling the Continuum

We break the execution of a function down into several smaller components, which when put together, model the different time costs associated with running a particular task on a particular endpoint. Running $\mathsf{f}(\mathsf{x}, \mathsf{data})$ on an endpoint EP involves the following time costs:

(1) *Scheduling Overhead:* Time taken by *Delta* to make a scheduling decision and queue a task for execution. Call this t_{sched}.

(2) *Cold-Start Latency:* Time taken to allocate a node, start a container, and load package dependencies for f, if the endpoint is not already warm. Call this $t_{\text{cold}}(\mathsf{EP}, \mathsf{f})$.

(3) *Pending-Tasks Delay:* Time taken for all previously scheduled requests on EP to finish.[1] Call this $t_{\text{prev}}(\mathsf{EP})$.

(4) *Function Transfer Time:* Time taken to transfer f and x from *Delta* to EP. Call this $t_{\text{trans}}(\mathsf{EP}, \mathsf{f}, \mathsf{x})$.

(5) *Data Transfer Time:* Time taken to transfer each input file in data from its source to EP. Call this $t_{\text{trans}}(\mathsf{EP}, \mathsf{data})$.

(6) *Runtime:* Time taken by EP to run f on input x and data, producing output res. Call this $t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}, \mathsf{data})$.

(7) *Result Transfer Time:* Time taken to transfer res from EP to *Delta*. Call this $t_{\text{trans}}(\mathsf{EP}, \mathsf{res})$.

Thus, the total time to execute $\mathsf{f}(\mathsf{x}, \mathsf{data})$ on an endpoint EP is the sum of the quantities above. So, if one could accurately predict these components, choosing an endpoint to

1. Since tasks are executed in FIFO order on the endpoint.

minimize the time-to-completion for a task would be a matter of evaluating the following:

$$\underset{1 \le i \le n}{\textbf{argmin}} \left(\begin{array}{c} t_{\text{sched}} \ + \ t_{\text{cold}}(\mathsf{EP}_i, \mathsf{f}) \ + \ t_{\text{prev}}(\mathsf{EP}_i) \ + \ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{data}) \\ + \ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}) \ + \ t_{\text{run}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}, \mathsf{data}) \ + \ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{res}) \end{array} \right)$$

3.4 Assumptions

We restrict the scope of the problem with the following simplifying assumptions:

(A1) Functions being requested have deterministic runtimes. *So that their runtimes lend themselves to prediction, when given necessary information like function inputs.*

(A2) Each endpoint runs one task at a time, in first-in-first-out (FIFO) order. *To give tasks unobstructed access to an endpoint's resources.*

(A3) Each task runs only on one endpoint (as opposed to running concurrently on multiple endpoints). *To disallow, for now, additional complexities in scheduling.*

(A4) There are no inter-task execution dependencies. *We leave the scheduling of workloads with multiple inter-dependent functions to future work.*

(A5) We are only concerned with minimizing execution time of individual tasks. *We do not optimize multiple tasks together, and we leave the accounting of money, energy, task priorities, etc. to future work.*

(A6) The overhead of scheduling a task, t_{sched}, is a constant. *This is true in practice for all "reasonably" sized functions, since the funcX overhead remains consistent.*

(A7) The cost of communicating functions to and results from different endpoints, $t_{\text{sched}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}) + t_{\text{trans}}(\mathsf{EP}_i, \mathsf{res})$, is a constant. *Any differences in communication times to different endpoints are on the order of a few milliseconds, which is negligible compared to all function executions we will be requesting.*

CHAPTER 4

DESIGN AND ARCHITECTURE

We now present the design and architecture of *Delta*, a framework for heterogeneous function execution that learns function behavior in order to make robust scheduling decisions. While *Delta* could be extended to support arbitrary scheduling policies, its current implementation focuses on minimizing the execution time of incoming tasks. To do this, *Delta* performs multidimensional trade-off analysis, taking into account factors such as compute time, data transfer time, cold-start time, as well as other execution latencies.

4.1 Background

Delta builds upon *funcX* [2], a federated FaaS platform, as well as Globus [42], a research data management platform that supports high-performance third-party data transfer. We first provide some necessary background about these platforms.

4.1.1 FuncX

funcX [2] consists of a central orchestrating service, which communicates both with *funcX* endpoints (which execute tasks) and *funcX* clients (which request tasks). In order for a user to execute a function via *funcX*, they must first set up one or more endpoints on machines that they have access to. When an endpoint is registered with the *funcX* service, it is assigned a universally unique identifier (UUID), henceforth known as an *endpoint-id*. The process of setting up an endpoint is authenticated via Globus Auth [9], which is subsequently used to ensure that only users who have authorization to access an endpoint can run tasks on it. Once a user has set up an endpoint, they can immediately start running functions on it. All aspects of function execution happen via a `FuncXClient` that a user must initialize (by authenticating via Globus Auth). The `FuncXClient` is first used to register a function. At

the time of registration, the function's body is serialized and sent to the main *funcX* service, which returns to the user a UUID, henceforth known as the *function-id*. When a user wants to run a function on some input, they must provide the function-id of a previously registered function, the endpoint-id of an endpoint to which they are authorized to access, and the input on which they would like to execute the function. We will refer to this combination of a function, endpoint, and input as a *task*. The *funcX* service returns to the user a unique *task-id*, which is then used by the user for tracking a task's status. All communication between the `FuncXClient` and the central *funcX* service occurs via an HTTPS-based REST API. This greatly simplifies *funcX*'s design since no state needs to be maintained for these communications. On the other hand, this means that *funcX* does not (currently) provide a blocking mechanism that waits until a task is finished — instead, all the user can do is query the *funcX* service to check whether a task has finished executing.

The *funcX* service communicates with an endpoint via a manager process on the endpoint, which is responsible for allocating incoming tasks to one or more worker processes. When a user requests a task to be sent to an endpoint, the *funcX* service sends the serialized function body and function input to the endpoint's manager process. Upon receiving this, the manager forwards it to one of its workers. The worker, upon receiving this task, deserializes the function body and function input, runs the function on the input, serializes the result and sends it to the manager. From here, the result is forwarded to the *funcX* service, where it is added to a task database. The next time the user's client polls for this task's status, this result is returned to the client.

funcX is built with the aim of running computations on a range of devices, from personal desktops and edge devices to clouds and supercomputers. To this end, *funcX* employs Parsl's provider interface [16], with which it is able to request and manage compute resources with a wide range of user-specified configurations. *funcX* also features multiple scaling strategies, allowing it to allocate more or fewer nodes based on the incoming stream of tasks on an

endpoint, as specified by the user's scaling parameters.

4.1.2 Globus

Globus is a research data management platform that, among other capabilities, provides data transfer [43] and identity and access management [9] capabilities. These capabilities are offered via a central Globus service which can be accessed both via web browsers as well as via a software development kit (SDK). Users and administrators can enable Globus data management on a storage system by deploying Globus Connect software on them, thereby turning them into Globus *endpoints*. This can be done both for personal machines (such as laptops) and for shared machines associated with dedicated server systems. For high-access server systems, Globus allows multiple machines to serve as data transfer nodes (DTNs), which not only yields increased transfer performance, but also more reliability with dynamic failovers. In either case, setting up Globus endpoints transforms storage devices into data management junctions that are able to transfer data efficiently and reliably.

Globus's highly performant methods of third-party data transfer are orchestrated by its central service. The Globus service is responsible for the entire process of executing a transfer, from authentication at the source and destination, to creating channels for transfer via the GridFTP protocol [8], and using checksums and recovery mechanisms to ensure the integrity of the data being moved. The Globus platform can be accessed through its expansive REST API, which is used by client libraries such as Globus's native Python SDK, allowing easy interfacing with third-party applications.

Globus endpoints enforce the file-access permissions of their underlying systems. Globus sharing allows for file-access decisions to be made by the Globus service. For instance, it is possible to specify access control lists (ACLs) for file paths [43]. These ACLs are enforced automatically by Globus when accessing and transferring files. Globus Auth [9] handles authentication and authorization by securely interfacing between identity providers, clients,

and server endpoints. This authentication can be integrated with third-party applications to allow them to perform actions on behalf of users, within the bounds of specified permissions (e.g., for moving a user's files). Thus, Globus's transfer capabilities are not only performant, but also properly authenticated and robust, which makes Globus an appealing choice for data transfers in dynamic grid-like environments.

4.2 Prediction

As described in Chapter 3, the key insight upon which we base *Delta*'s prediction design is to break up the total execution time for a task into several smaller components (such as runtime on an endpoint, transfer time, etc.). The advantage of this approach is that, compared to the entire execution time of a task, each of these smaller components is much easier for prediction models to learn. This is due to two reasons: first, that each of these components is specific enough that we can hope to apply specialized prediction techniques (including existing ones), and second, that for each of these components, it is easy to identify the (small) subset of features that are necessary for prediction. Moreover, adopting a modular approach to prediction allows us to readily swap out one predictor for another without any substantial changes to our infrastructure.

4.2.1 Function Performance

Perhaps the most conspicuous aspect of using heterogeneous devices is the observation that for every computation, there are some devices for which it is well-suited, and others which are less so. Yet, the question of whether a particular device is a good choice to run a computation does not admit a binary answer. Function performance varies across heterogeneous devices on a spectrum. Figure 4.1 shows how some simple every-day workloads perform on different devices, including Raspberry Pis, desktop computers, clusters, and GPU nodes (see Section 5.1 for a description of these different devices). The workloads shown are

as follows. The first is an embarrassingly parallel computation that divides up work across parallel processes (`map-reduce`). This computation performs best on machines with a large number of CPU cores. The second is a matrix multiplication workload that multiplies two square matrices (`matmul`). This computation performs best on GPUs, and is significantly worse on even powerful CPUs. The third involves reading and writing large files (`file-I/O`). This computation performs best on machines with faster disk speeds and CPUs.

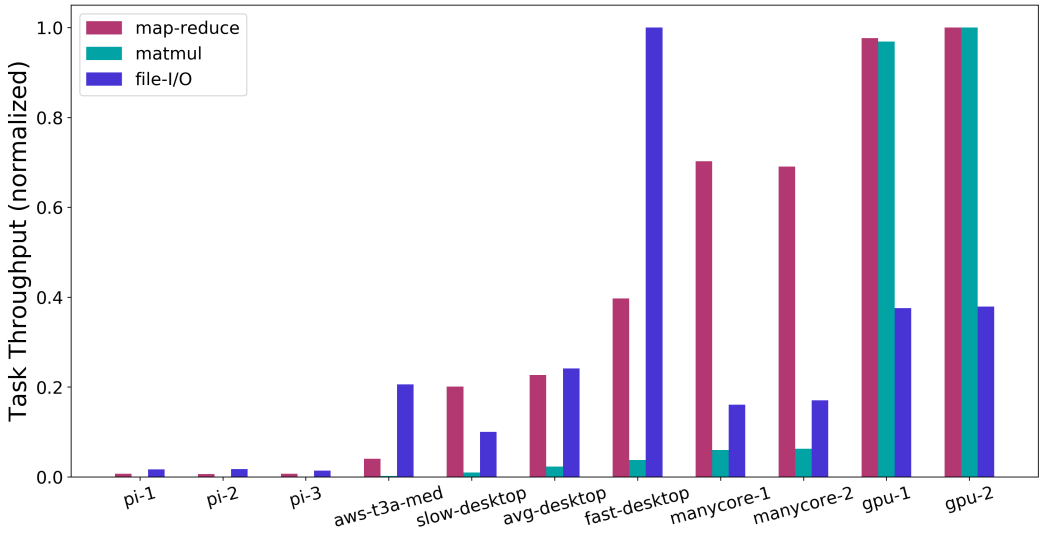

Figure 4.1: Relative throughputs of tasks on heterogeneous machines, with values normalized to the maximum throughput observed for a task.

Thus, it becomes clear that in order to predict the total execution time of a task, it is crucial to take into account the capabilities of the device on which the task is executed, since the executing device can greatly affect the task's runtimes, sometimes by multiple orders of magnitude. In order to predict task runtimes on different types of devices, we could consider either a white-box approach or a black-box approach. A white-box approach would likely involve using tools such as static analysis of function code and a precise accounting of the specific capabilities of each device (e.g., number of cores, memory, architecture, etc.). We believe there is a lot of potential for white-box predictions of function runtimes, but

this is a difficult open problem which ultimately distracts from our central goal of building a complete model for the computing continuum. We leave more sophisticated methods of function runtime prediction to future work, and focus here on a black-box approach where the only information we use is a function's execution history, including its inputs and observed runtimes on different devices.

The challenge of predicting $t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}, \mathsf{data})$ is thus reduced to building an online predictor that uses a list of data points of the form $(\mathsf{x}_i, \mathsf{data}_i, t_{\text{run}}(\mathsf{EP}, \mathsf{f}, \mathsf{x}_i, \mathsf{data}_i))$, collected from all previous instances where f was run on EP. To this end, we created two comparable runtime-prediction strategies:

- *Rolling-Average:* A baseline strategy, which completely ignores the inputs given to the function, and only looks at a rolling average of runtimes for previous executions of the function on each endpoint.

- *Input-Size:* Often, the runtime of a function depends heavily on the input provided. To model this, we created a strategy that looks at the inputs to a function and tries to find a correlation between the size of the input and the observed function runtime. For this, we use online polynomial regression. Every `r` observations, the model is retrained on the entire execution history of the function (where `r` is configurable).

Of course, being black-box methods, both prediction strategies need data to learn from, i.e., a function's execution-history on different endpoints. As we will describe in Section 4.3.6, this data is gathered by an initial "exploration" phase where we try newly registered functions on different endpoints to gather performance information.

An immediate criticism of these strategies would be that they involve maintaining and retraining a sizeable number of models (i.e., one model for each pair of function and endpoint). As we see it, there are two types of transfer-learning approaches which would alleviate this problem. First, cross-function learning, i.e., using the execution history of one function to help predict runtimes for a different function. This would fit in well with the white-box

strategies briefly alluded to above, and is left for future work. Second, cross-endpoint learning, i.e., using the execution history of a function on one endpoint to help predict its runtime on another endpoint. This approach benefits from the following observation: devices with similar capabilities are near-perfect predictors of runtimes on each other. Here, *capabilities* refers to a combination of the device's available memory, disk types, CPU cores, the presence of specialized accelerators, and so on. As is evident from Figure 4.1, the runtimes on the 3 Raspberry Pis are excellent predictors for each other, as are runtimes on the GPU machines, as are the runtimes on the manycore machines. We used this insight to modify the runtime-prediction strategies described above by categorizing endpoints into *endpoint groups* (e.g., the 3 Raspberry Pis in one group, the 2 GPUs in another group). Learning and prediction can thus be done for endpoint groups instead of individual endpoints. Other than reducing the number of models being tracked, another benefit of this modification is that learning occurs faster, since it is no longer necessary to observe every function on every endpoint. This led to more accurate predictions more quickly.

4.2.2 Data Transfers

Often, function performance is the strongest factor one tends to consider when selecting a device on which to run a computation. But of course, a function's runtime is only one part of the cost associated with executing a function on a remote endpoint. One of the largest counteractive factors is the time it takes to move the requisite task-execution information to the remote endpoint. This involves not only the latency of communicating with the endpoint, but also the cost of transferring additional data in cases where the function needs to act upon a non-trivial amount of input data — for example, image-processing tasks such as facial-recognition, or the analysis of sensor data collected by edge devices. As described in Section 4.3.3, one of our contributions involves building an automated mechanism (using Globus) to move input files to *funcX* endpoints for function execution. Thus, it is necessary

for us to be able to predict how long data transfers to and from each endpoint will take.

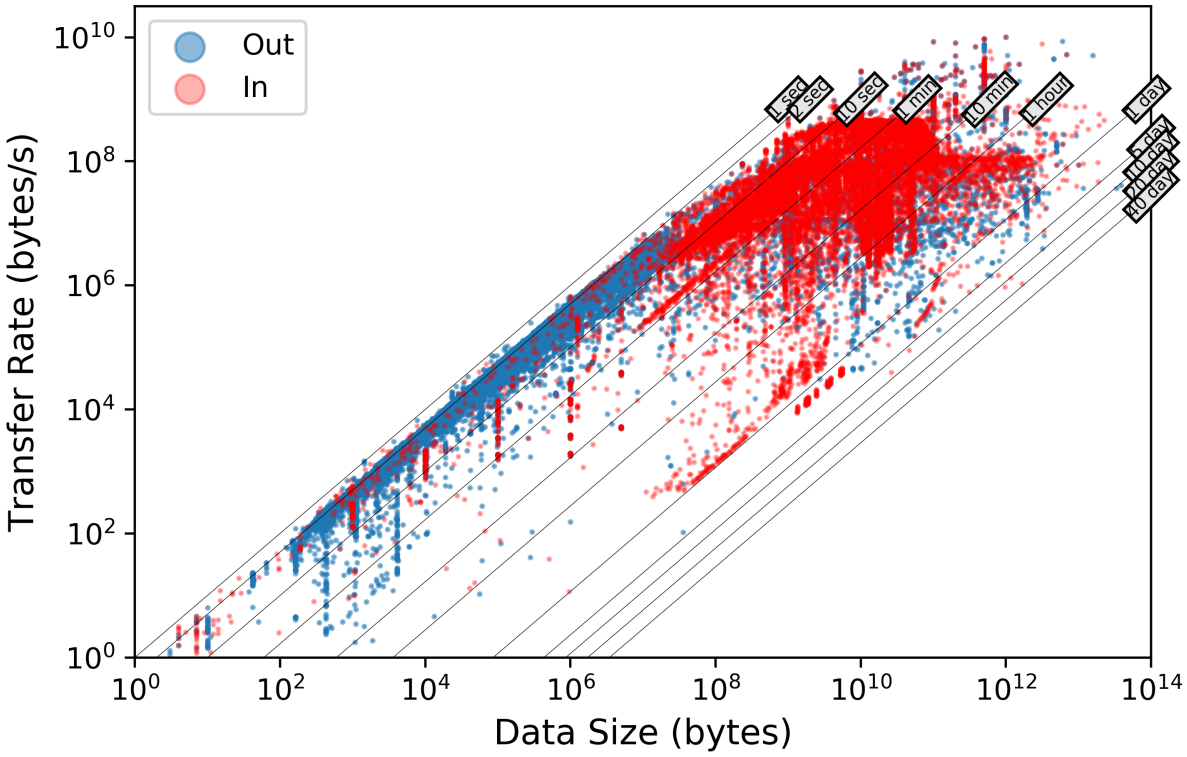

Figure 4.2: Historical data-transfer rates to and from Petrel [44], a large-scale data service for use by scientists.

Figure 4.2 shows historical Globus transfer times for Petrel [44], a large-scale data service deployed at Argonne National Laboratory, and other Globus endpoints distributed around the world. It is interesting to note that, up to a few MB, transfer times remain about constant (around 5 seconds). For such small transfers, the effective transfer rate observed seems to grow linearly with the size of the data. After around 100 MB, the transfer rate seems to plateau (in this case, at about 100 MB/s). This trend is explained by the overhead of conducting a Globus transfer, including interactions with the Globus service, setting up communication channels, and computing checksums for files.

We must note that Petrel has been carefully designed for optimizing movement of data. Moreover, being a high-performance data service, Petrel is comprised of several high-performance

DTNs, as well as network connectivity that is likely orders of magnitude better than that of a personal desktop computer. Transfer speeds depend not only on the effective network bandwidth and latency between two devices, but also the memory and networking capabilities of the devices; a powerful desktop connected to the local network via Ethernet will likely show higher transfer rates than a less powerful edge device connected via WiFi. Thus, much in the same way that we could not use a simple one-size-fits-all approach to model function performance, we cannot use a one-size-fits-all approach to model transfer times between different pairs of endpoints.

That said, similar to our analysis of function performance, what we *can* do is group endpoints by the set of features which affect transfer times. We grouped the endpoints in our testbed by their physical and network locations, as well as by the amount of memory they have. We then measured, for each such pair of *transfer groups*, how transfer rates scaled with data size. Figure 4.3 shows some of our findings. As hypothesized above, we do see that transfer rates vary greatly based on the source and destination of the transfer. Transfers between two desktop computers on a local network are much faster than transfers from this local network to an AWS instance in Virginia. On the other hand, transfers between AWS and the Chameleon cluster at Argonne are incredibly fast — likely due to sophisticated network infrastructure on both ends.

Having collected this transfer data, we built regression models to predict this logarithmic relation for each pair of source and destination transfer groups. The predictions of our models are shown in the dotted lines in Figure 4.3. Thus, we are able to predict transfer times reasonably well. These models were the ones used for transfer time prediction in our scheduling system described later. The fact that almost all of the transfer rate curves have the same shape (linear growth followed by plateauing) might suggest that it is possible to model these transfer times without having to train and maintain a regression model for each pair of endpoint transfer groups. Instead, one could hypothesize building a "universal"

model of transfer time prediction, which takes as input certain features of the source and destination (such as location, networking capabilities, etc.), along with the size of the data. Indeed, there is some recent work that tries to explain how these different features can be used to explain transfer times [45]. This would indeed be quite useful and a welcome addition to *Delta*, but is beyond the current scope of our work.

4.2.3 Cold Starts

The *cold-start problem* is a well-known and well-documented problem in FaaS literature [37, 2, 30], and lies at the core of why the scheduling of functions is quite different from the scheduling of jobs in HPC clusters. On the one hand, function executions require starting nodes, instantiating containers, and loading dependencies, much like regular HPC jobs. On the other hand, function executions are usually short-lived, triggered frequently, and often in need of low-latency responses. Thus, as the designs of almost all FaaS systems can attest to, it is impractical to incur the entire "cold start cost" — of acquiring nodes, containers, and package dependencies — for every function execution.

Contemporary FaaS systems have taken varied approaches to this problem, from keeping "pre-warmed" nodes [10, 32, 30] to applying optimized container technologies [37]. But for our purposes, the goal is not to explicitly reduce cold-start latencies, but to be able to *predict* them when they do inevitably occur. To this end, we treat cold starts as a sequence of three consecutive steps: node acquisition, container instantiation, and package loading.

The first of these, the acquisition of nodes (e.g., in an HPC cluster), is not only the slowest factor, but also the hardest to predict. Queue times are highly variable, depending on factors including (but not limited to) cluster capacity, cores requested, walltime requested, and the job-scheduling strategies employed by the system administrators. Figure 4.4 shows how queue wait times varied for cluster-jobs on Argonne National Labortory's Theta supercomputer [46], when plotted against the number of cores requested and the amount of

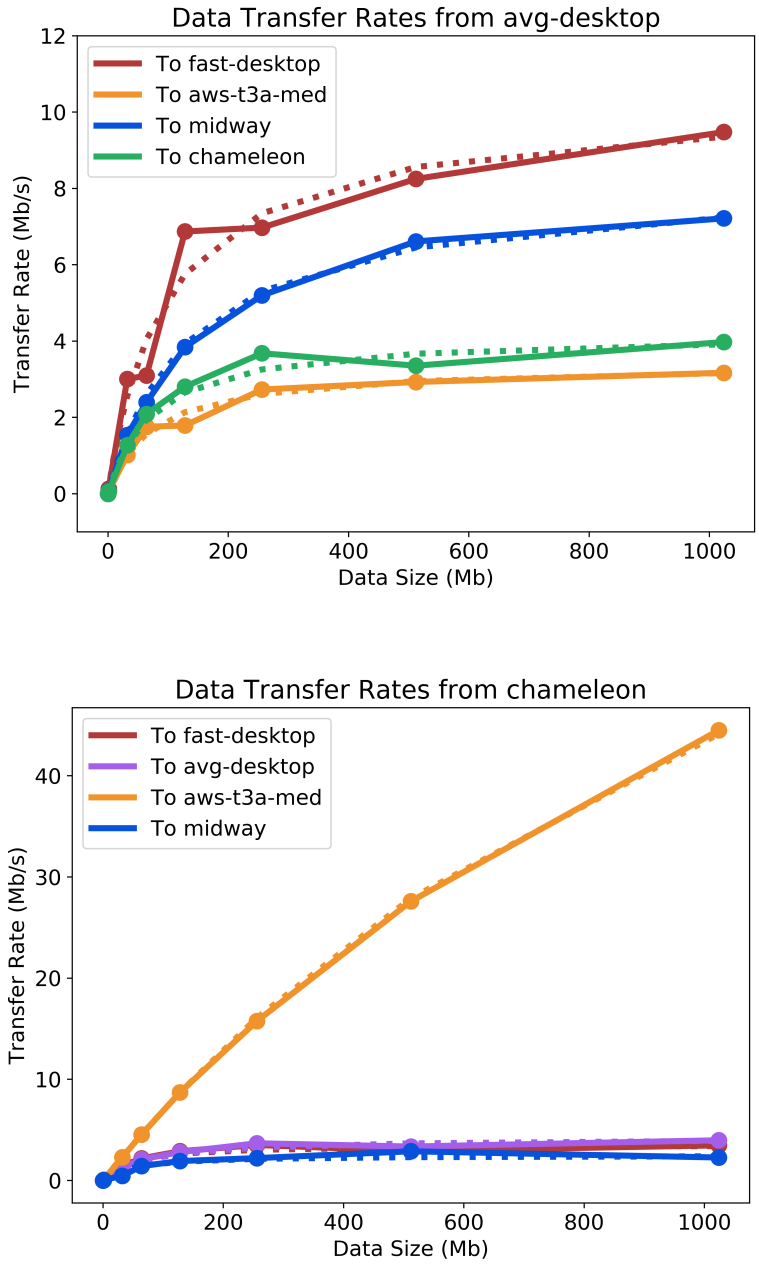

Figure 4.3: Data-transfer rates between Globus endpoints in our testbed (see Section 5.1). The circles represent medians of measured values, and our model's predictions are shown by the dotted lines. The two desktops are on the UChicago CS department's network, Midway is a cluster run by UChicago's Research Computing Center (also in Chicago), the Chameleon cloud is at Argonne National Laboratory, and the AWS instance is located in Virginia.

time requested. As is evident from this figure, it is not immediately obvious that queue wait times are predictable by simply looking at the features of a requested job. In fact, the wait times for jobs with similar requirements can sometimes differ by multiple orders of magnitude. This, along with the fact that queue times are incomparably large in relation to function execution times, leads most FaaS system designers to side-step this problem. This is often done by keeping a collection of "warm" nodes running at all times. For our work, we will assume that all our endpoints either do not need to wait for node allocation (e.g., edge devices and desktop computers), or in case they are in HPC clusters, they are already running on warm nodes.

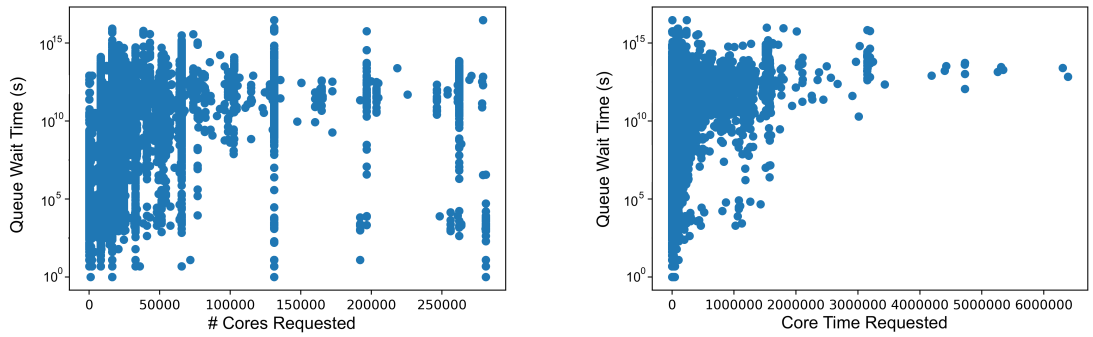

Figure 4.4: Observed queue wait times for jobs requested on Theta [46], a supercomputing cluster run by the Argonne Leadership Computing Facility.

The second component of cold-start latencies is the cost of starting containers. As before, when functions need to be executed inside containers, it does not make sense to start a new container for each execution. Instead, different approaches have been documented to direct incoming tasks to already-running containers [30, 39, 40]. Even these approaches, however, are not completely devoid of container cold-start costs. When increasing load requires spinning up new containers, the first few function executions often require cold starts. Thus, it is important to be able to predict the latencies these cold starts will incur, to improve our function scheduling decisions. Moreover, as Table 4.1 shows, the cold-start latencies for different types of containers vary significantly across machines. These differences

are often attributed to a combination of differing clock-speeds and file-system contention in different systems [2]. Yet again, we encounter the problem of heterogeneous devices providing heterogeneous results.

System	Container	Min (s)	Max (s)	Mean (s)
Theta	Singularity	9.83	14.06	10.40
Cori	Shifter	7.25	31.26	8.49
EC2	Docker	1.74	1.88	1.79
EC2	Singularity	1.19	1.26	1.22

Table 4.1: Replicated from *funcX* [2]. Cold container instantiation times for different container technologies on different resources.

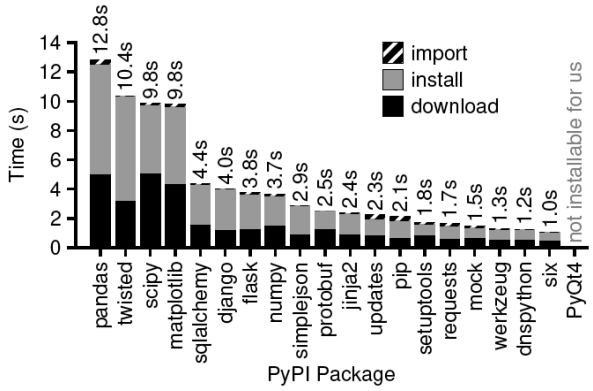

Figure 4.5: Replicated from SOCK [37]. Download and installation latencies for the most common Python packages.

The third component of cold-start latencies is the cost of loading dependencies, which, for us, means Python packages. This includes the latency of downloading, installing, and importing packages. The authors of SOCK [37] measured these costs for the most popular packages — their results are replicated in Figure 4.5. We see that this component of cold-start costs is quite substantial, especially for short-running tasks for which low-latency is of utmost importance. As one might expect, these costs too vary from device-to-device. Figure 4.6 depicts the time taken to import some common Python packages on the different devices in our testbed — one can expect download and installation costs to vary similarly.

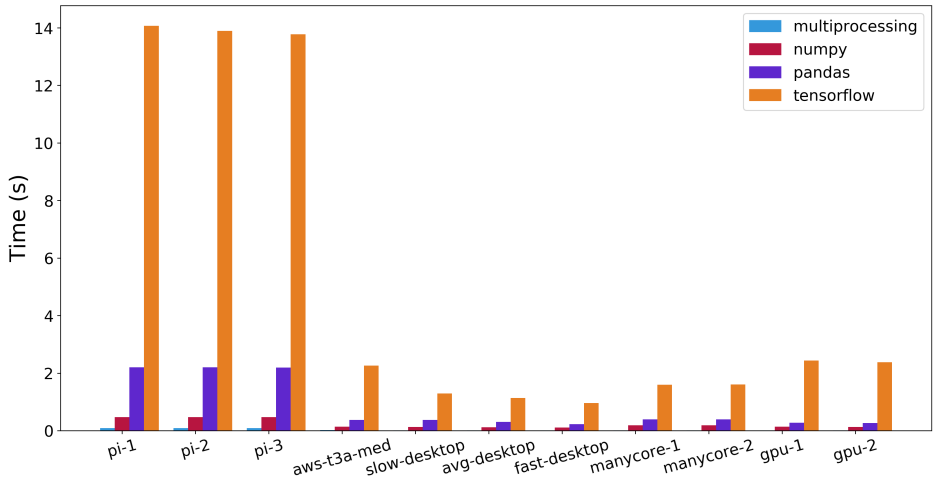

Figure 4.6: Import latencies for some common Python packages on our heterogeneous array of devices. Download and installation latencies can be expected to vary similarly.

These are easily predictable quantities, so accounting for these package-loading latencies is a matter of tracking which dependencies exist on each endpoint, and which ones are required for each function. We describe how we do this in Section 4.3.4.

4.3 System Architecture

Having described the design of the various predictors used by *Delta*, we now describe *Delta*'s architecture (shown in Figure 4.7). We wanted to build *Delta* on top of *funcX* without modifying any of *funcX*'s underlying execution fabric. Indeed, we were able to achieve this goal. This is noteworthy in two regards. First, that our development process was orthogonal to that of the *funcX* team, allowing existing users and developers of *funcX* to operate as usual. Second, and more importantly, this indicates that it is possible to extend existing FaaS systems without major changes to build a computation ecosystem that automatically takes into account heterogeneity.

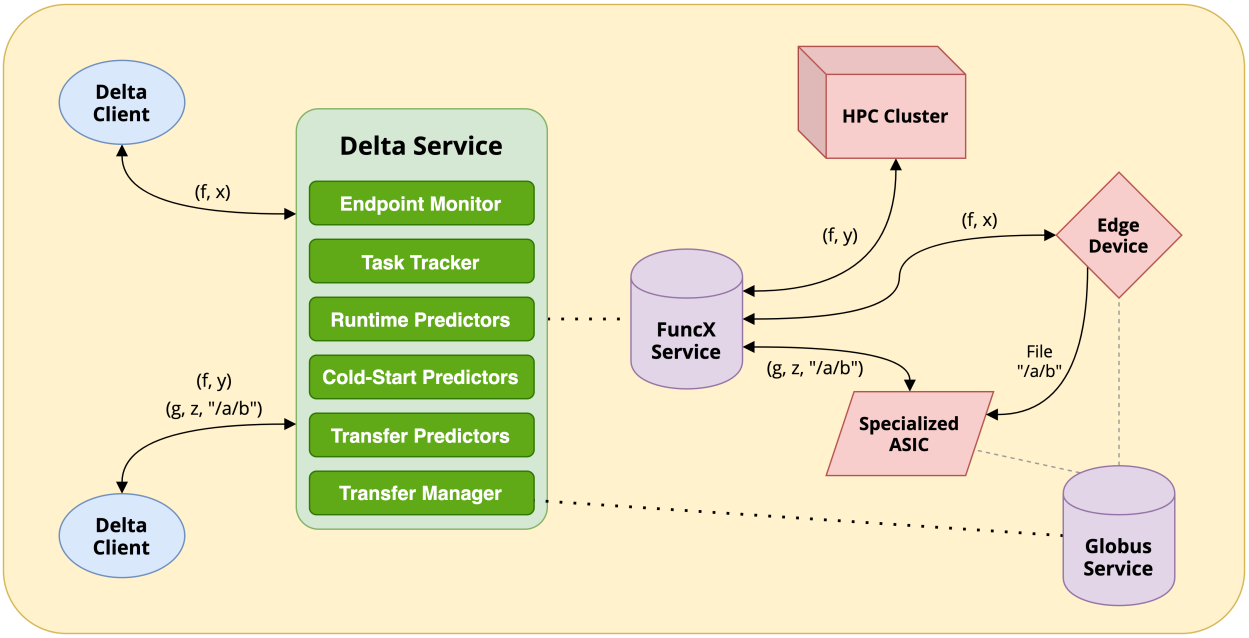

Figure 4.7: *Delta*'s architecture, including different components of the scheduling service, and the communications required to execute functions.

4.3.1 Continuum-as-a-Service

The main component of *Delta*'s architecture is a scheduling service — hereafter referred to as the *Delta service* — that serves as a proxy for the central *funcX* service. Instead of communicating directly with the *funcX* service, clients are configured to communicate with the *Delta* service. The *Delta* service responds to requests (almost) exactly how the *funcX* service would. To do this, it forwards requests to the *funcX* service when needed, for example, to register functions or request a task's status. Whenever a task is requested to be executed, the *Delta* service uses its scheduling strategies (described later) to choose an endpoint, and forwards the task request to the chosen endpoint via the *funcX* service.

The *Delta* service maintains various components to enable task tracking, runtime prediction, data transfers, and endpoint monitoring, as described in subsequent sections.

4.3.2 Client Wrapper

We created a wrapper for the `FuncXClient` — hereafter referred to as the *Delta client* — to allow for some necessary functionality that had to be added to the client. The *Delta* client is different from the regular `FuncXClient` in the following ways:

- When a user asks to register a function `func`, instead of registering `func` with the service, the *Delta* client wraps the function with code to record the start and end times of the function's execution and returns them as part of the function's return value. This is how function runtimes are measured. Of course, before returning results to the user, the actual result of `func` is extracted from the larger returned value.

- When asking to run a function on an input, the user does not need to specify an endpoint to run on. As mentioned above, the endpoint is filled in by the *Delta* service before forwarding the task request to *funcX*.

- When asking to run a function on an input, the user may optionally provide a list of files along with the Globus endpoint(s) on which these files are located. This list of files is forwarded as part of the function execution request, and will be transferred to the destination endpoint before function execution by the *Delta* service. Note that this does *not* restrict the files to originate from the requesting client's device.

- The *Delta* client tracks all pending tasks on the endpoint and runs a `TaskWatchdog` thread in the background. This `TaskWatchdog` regularly polls the service to inquire about the status of each pending task. This is necessary since *funcX* function executions are asynchronous and *funcX* does not provide a mechanism to wait on a result in a blocking way. It is crucial for *Delta* to receive results as soon as they are ready without depending on the user to query a task status, both so that it can learn to predict runtimes, and so it can keep track of which tasks are still running on each endpoint (described in more detail in Sections 4.3.5 and 4.3.6).

4.3.3 Automated File Transfer

funcX [2] places limits on the size of the input data that can be submitted as part of a task request, for reasons of scalability. It would be much harder to design the central *funcX* service if we allowed users to submit inputs of arbitrarily large sizes, since in this case, the *funcX* service would have to not only store large amounts of input data for tasks that have been requested, but would also struggle to efficiently communicate with endpoints due to large message sizes which would be sent to endpoints. Moreover, data transfers can often be unreliable, especially since *funcX* endpoints can be set up on arbitrary machines connected by arbitrarily unreliable networks, which would compel *funcX* designers to build efficient and reliable data transfer infrastructure. Finally, it does not seem sensible to create a bottleneck for data transfers by making all data go through a central service. It would make more sense to move data directly from its source to the destination endpoint.

Motivated by this line of reasoning, we looked to Globus [42], since it is an existing system that has managed millions of wide-area data transfers around the world. Globus provides high-performance data transfers using parallel data streams and other performance enhancements, ensures the integrity of transferred data by comparing checksums, and includes a comprehensive security model that enables authentication at both source and destination as well as encryption of the data channel. Moreover, Globus supports third-party data transfers, which can be orchestrated by the *Delta* service.

In order to enable endpoint-to-endpoint transfer of data, we deployed a Globus endpoint on every machine where we had a *funcX* endpoint.[1] The *Delta* service maintained a mapping between *funcX* endpoints and their corresponding Globus endpoints. We supplemented the *Delta* service with a `TransferManager` component which is responsible for coordinating data transfers with the Globus service. When a new task is submitted to the *Delta* service with a

1. Note that, at the time of writing, Globus does not support machines with ARM architectures. Because of this, we were unable to deploy a Globus endpoint on 3 of the machines in our testbed.

list of files required for execution, *Delta* first chooses an endpoint to direct the task to (according to the strategies in Section 4.3.6). Once an endpoint is chosen, the `TransferManager` is instructed to transfer all the given files to this chosen endpoint, using the corresponding Globus endpoints of the source and destination machines. The `TransferManager` regularly monitors the status of the Globus transfers via the Globus API. The Globus *transfer-ids* of these transfers are sent as part of the task input, so that the *funcX* endpoint can wait until the transfers are complete before executing the function. Note that this is inefficient since it does not allow other tasks to execute on the endpoint while it is waiting for data transfers. We will describe an optimization for this in Section 4.4.2.

4.3.4 Monitoring Endpoints for Cold Starts

As Section 4.2.3 elucidates, cold starts are a significant factor that add to function execution latency, and are especially detrimental to short-running tasks. In order to predict cold start latencies, we need to track the status of each *funcX* endpoint that we can execute on. To this end, the *Delta* service runs an `EndpointMonitor` component which regularly communicates with each endpoint and tracks the endpoint's status. Each *funcX* endpoint is configured to send regular heartbeat messages to the service. These heartbeat messages consist of the endpoint's current state, including the number of pending tasks queued on the endpoint, the number of active worker and manager processes, resource usage, etc. Since there is a manager process on each node allocated to the endpoint, checking if there is a warm node running on the endpoint is simply a matter of checking that there is at least one active manager. If no active managers are reported, the `EndpointMonitor` marks the endpoint as *cold*. This fact is used for subsequent scheduling decisions. If a cold endpoint is chosen for execution, it is marked 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, so that package-import costs can be accounted for in the cold-start costs. Tracking this requires tracking two things — the list of packages currently imported on each endpoint, and the list of packages required by each function. The former is done by modifying the function-wrapper that is registered (see Section 4.3.2) to also return a list of all imported Python modules at the end of every function execution.[2] The latter is done by analyzing the function body at the time of function registration, and extracting all lines of code which execute package imports. With these two modifications, it becomes straightforward for the *Delta* service to identify all packages required by a function execution that will need to be imported for the first time on an endpoint. Note that tracking which packages are *installed* on each endpoint would require a similar modification, which we leave for future work.

4.3.5 Pending-Task Tracking

All of our endpoints are configured to execute tasks in first-in-first-out order, which means that when a new task is scheduled on an endpoint, it must wait for all previous tasks on this endpoint to finish executing. Thus, an essential component for execution time prediction that we must estimate is the time at which the most recent task scheduled on each endpoint will finish running. We refer to this as the endpoint's *pending time*. For this, the *Delta* service maintains a first-in-first-out queue of scheduled tasks for each endpoint. Every time a new task is scheduled on an endpoint, its ETA is calculated — this ETA calculation takes into account the previous estimate for this endpoint's pending time. Upon sending the new task to this endpoint, the endpoint's pending time estimate is updated to the ETA of this new task.

One possible shortcoming of this approach is that it compounds errors in ETA prediction, which are inevitable when dealing with real-time systems. For example, if the ETA prediction for the first task underestimates the actual execution time by 10 seconds, then the ETA

2. The built-in `sys` module in Python provides this functionality.

prediction for subsequent tasks will also have an added error of 10 seconds. To avoid such an undesirable compounding of errors, the *Delta* service also maintains a *pending-error* for each endpoint. When there are no pending tasks on an endpoint, its pending-error is 0. When there is at least one pending task on an endpoint, its pending-error is set to the error in the ETA prediction of the most recent task completed on the endpoint. In other words, when a completed task result is received from an endpoint, the endpoint's pending-error is reset to the difference of the task's ETA and its actual observed execution time. This pending-error is added onto the pending time estimate for an endpoint at the time of ETA prediction. In this way, we at least partly mitigate compounding errors from previous predictions.

4.3.6 Choosing Endpoints and ETA Prediction

Having described the main components of our function-orchestrating system, we now describe how the choice of endpoint is actually computed for each incoming task. *Delta*'s design inherently depends on having reasonably good estimates of multiple types of cost. Because of this, we wanted to ensure that we include modularity in the very core of our design. Thus, our primary scheduling strategy allows for arbitrary predictors to be plugged-in for each component described below. We briefly describe the initial predictors we have built for our experiments. We reemphasize that these predictors are not the focus of this work, and further enhancements will be necessary to productionize a function-serving framework that resembles *Delta*.

- **Runtime:** We currently have two runtime predictors, as described in Section 4.2.1 — one simply uses a rolling average of past runtimes of a function on the endpoint, whereas the other performs online polynomial regression on the size of the input to predict function performance. As mentioned earlier, all runtime learning is done after grouping endpoints by their capabilities.

- **Data Transfer Time:** Our current predictor for data transfer times relies on historical

Globus transfer statistics between pairs of endpoints, as described in Section 4.2.2. The endpoints are grouped by their network location and their network transfer capabilities when predicting transfer times.

- **Cold-Start Latency:** As described in Section 4.2.3, we do not model the time taken for node allocation in HPC clusters, instead assuming the existence of pre-warmed nodes. We described how we track which packages are installed on each endpoint and which are required by every function execution in Section 4.3.4. Using this information, we use the data collected for import times on the different endpoints to predict the associated latency. We do not currently account for container start-up time, but this is an important area of future work.

- **Scheduling Overhead:** We observed that for tasks of reasonable sizes (including any tasks we ran in our experiments), the overhead of the *funcX* service to run the task was surprisingly consistent. Thus, we do not currently employ any clever techniques to predict scheduling overhead, and instead, just add the observed average *funcX* overhead to our ETA calculation.[3]

On top of this, we use the approach from Section 4.3.5 to track the pending time for each endpoint. Recall that, as mentioned in our assumptions in Chapter 3, for simplicity, we do not currently include the cost of transferring large results. For this, we would need to extend our file-transfer infrastructure to allow function executions to output result files to a Globus endpoint. This could be done by using the predictor for data transfers along with a user-provided estimate of the size of the result.

When given an incoming task $\mathsf{t} = (\mathsf{f}, \mathsf{x}, \mathsf{data})$, our primary scheduling strategy, which we call `smallest-ETA`, computes the sum of each of these predictions, i.e., the *ETA* of running

3. It would be interesting to see how *funcX*'s overhead scales with batch-submission of tasks. If there are significant differences in overhead with batch size, we would need to model this to a sufficient degree of accuracy, especially for short-running tasks.

`t` on an endpoint. It does so for each *funcX* endpoint, and simply chooses the endpoint which minimizes the ETA. Now of course, in order for the (black-box) runtime predictions to be any good, the runtime predictor must be trained on historical data points. To this end, the first few executions of a function are sent to different endpoint groups, so as to collect data about how the function performs on different types of devices. In a production system, this "exploration" stage could be done offline at the time of function registration. Once this exploration stage finishes, the strategy starts choosing endpoints as described above.

We will compare the performance of our scheduling strategy against two baseline strategies, `round-robin` and `fastest-endpoint`. As their names suggest, the `round-robin` strategy is the naive approach of simply cycling through all endpoints for an incoming stream of tasks, whereas the `fastest-endpoint` strategy first goes through the same "exploration" stage described above, after which it starts choosing the endpoint which minimizes the function runtime for a task. In Chapter 5, we will evaluate how our scheduling strategy performs in comparison to these baselines.

4.4 Optimizations

While the description of *Delta* until now is complete in itself, there are many ways in which the presented design can be improved. We now present some such improvements.

4.4.1 Local Execution

Until now, we have focused on choosing which remote endpoint a computation should be sent to. As we have seen, the two biggest conflicting forces at action when weighing where to run a computation are the cost of running on an endpoint, and the cost of communication to and from this endpoint. Thus, a natural question arises — is it always necessary to incur the cost of communicating with a remote endpoint? The answer is a resounding no. One can easily think of workloads which would be optimally scheduled to run locally on the requesting

device, instead of incurring the network-cost of executing on a remote endpoint. Indeed, in addition to choosing the best endpoint for a computation, we must also be able to decide *if* the computation should be run remotely at all. Only then can our model of computation be justifiably described as fluid.

The naive approach to enable function execution on the client device would be to register a *funcX* endpoint on the client's machine, so the central *Delta* service may choose to execute tasks on it. However, this approach would defeat the purpose of avoiding the latency of running a task remotely, as every task would have to incur, at the minimum, the round-trip network latency and the *funcX* scheduling overhead. This motivates the need to run functions locally without going through a central service. Usually, this would likely be problematic, since there is significant infrastructure required to set up a FaaS endpoint. However, luckily, *funcX* is designed to execute on all sorts of devices, and so as it turns out, it is not terribly cumbersome to build a local *funcX* task executor. Naturally, we built a lightweight `LocalExecutor` using the same task serialization and execution semantics as the regular *funcX* executor. A `LocalExecutor` process is (optionally) started as part of the *Delta* client, and communicates tasks and results via inter-process communication, which is significantly faster than network communication with the *Delta* service. Of course, this means that the *Delta* client now needs to be involved in the decision process for task scheduling. For this, the *Delta* client was modified to track the rolling average of the total execution times that were seen for a function locally and remotely. The *Delta* client uses this information to make the binary decision of whether to run a computation remotely (send to *funcX*) or locally (send to `LocalExecutor`).[4]

4. We could, of course, improve this decision-making process by employing similar predictors as in the *Delta* service.

4.4.2 Just-in-Time Task Submission

Back in Section 4.3.3, we described our mechanism for transferring required files to the chosen endpoint for a task execution. This approach has a severe limitation, which is that while a data transfer is occurring for a task, the endpoint is idle, lying in wait for the transfer to finish. This time could be used to execute other incoming tasks, so that the device's available resources are not wasted. To this end, we built "Just-in-Time" submission of tasks which required data transfers. Since the `TransferManager` actively coordinates the data transfer for each task with Globus, the *Delta* service is alerted as soon as the transfer finishes. Only at this time is the task actually submitted to the endpoint via *funcX*. Note that while the data transfer is in progress and the task is not submitted to *funcX*, the task is *not* accounted for in the pending time estimation for the endpoint.

Observe that since data transfers and pending task execution are now allowed to occur concurrently, we need to tweak our ETA prediction computations, so that we find the endpoint with the smallest ETA via:

$$\underset{1 \leq i \leq n}{\textbf{argmin}} \left(\begin{array}{c} t_{\text{sched}} \ + \ t_{\text{cold}}(\mathsf{EP}_i, \mathsf{f}) \ + \ t_{\text{wait}}(\mathsf{EP}_i, \mathsf{data}) \\ + \ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}) \ + \ t_{\text{run}}(\mathsf{EP}_i, \mathsf{f}, \mathsf{x}, \mathsf{data}) \ + \ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{res}) \end{array} \right)$$

where $t_{\text{wait}}(\mathsf{EP}_i, \mathsf{data}) = \max(t_{\text{prev}}(\mathsf{EP}_i),\ t_{\text{trans}}(\mathsf{EP}_i, \mathsf{data}))$.

4.4.3 Blocking Endpoints for Functions

Functions that have uncommon dependencies or require specialized resources can often not run on all endpoints. Of course, it is not feasible to only use the endpoints which can run *all* of a user's functions (if such endpoints even exist). Instead, we added the capability to block specific endpoints on a function-wise basis. The *Delta* service was modified to maintain per-function lists of blocked endpoints, which could be added to via API calls

from the *Delta* client. For every task-scheduling decision, *Delta* simply ignores all blocked endpoints for the function in concern. Further, on top of the capability to manually block endpoints for a function, we noticed that it should also be possible to automatically detect some cases when an endpoint should be blocked. For instance, when running a function on an endpoint consistently runs out of memory (e.g., a Python `MemoryError`), or is missing required dependencies (e.g., a `ModuleNotFoundError`). When such exceptions are detected, the *Delta* service automatically blocks that endpoint for the given function, thus allowing for quick discovery of which devices are capable of running which computations.

4.4.4 Endpoint Failures and Slowdowns

A significant challenge when working with heterogeneous devices is providing reliability in the face of device failures and slowdowns. While testing our framework, we ran into issues where some endpoints would either stop responding completely or would take an order of magnitude longer than expected to finish a function execution. There were many reasons for such errors, including memory leaks causing use of swap memory, overheating (especially for Raspberry Pis), and queuing delays on overloaded endpoints (usually the fastest ones).

To combat this, we introduced backup tasks to *Delta*. We made the `EndpointMonitor` regularly check for two types of conditions in which backup tasks were to be sent. The first condition is when an endpoint seems to have died. To detect this, the `EndpointMonitor` was modified to track heartbeat messages coming from *funcX* endpoints. If no heartbeat is received from an endpoint within a configured time-window, the `EndpointMonitor` assumes that the endpoint is either dead or on the other side of a network partition. The second condition is when a task takes significantly longer than expected (e.g., 2 or 3 times longer). While the *Delta* service does indeed track ETAs for all tasks, it is important to note that these ETAs are not always reliable, especially not during the initial "exploration" stage in which *Delta* is just trying endpoints and has not learned function behavior yet. So, during

the initial learning period for each function, being unable to trust our ETA predictions, we choose not to send backup tasks.

Backup tasks are always sent to an endpoint other than the ones chosen previously for a task. Multiple backups can be sent, up to a configurable `max_backups` parameter. Of course, the underlying *funcX* service cannot distinguish between a regular task and a backup task, so we had to redesign our task-tracking mechanism, as described in Section 4.4.5.

4.4.5 Task-ID Translation

In two separate situations, we encountered a need to separate the task-ids returned to the client from the task-ids used by the *funcX* service and *funcX* endpoints for execution. The first situation was Just-in-Time submission for tasks, as described in Section 4.4.2. The *funcX* client expects a valid task-id to be returned as soon as a task-execution request is received by the service. However, in the case of Just-in-Time submission, we did not want to actually submit the task for execution to *funcX* until the data transfer to the endpoint had (almost) completed. Thus, we needed a mechanism for delayed task submission, without complicating the client's logic. The second situation was backup tasks, as described in Section 4.4.4. In the case of endpoint failure or task slowdown, we wanted the *Delta* service to be able to transparently send a replica of the task to a different endpoint, which would improve the reliability guarantees *Delta* could make. This meant that for each task that the user requested, it was possible that multiple tasks were submitted to *funcX*.

To solve both problems, we redesigned the task-tracking component of the *Delta* service, and maintained a *task-translation table*. Every time a user submits a task for execution, a *virtual task-id* is created and immediately returned to the user. This task is added to an internal scheduling queue. This queue is be monitored by a task-tracking thread, which regularly pops an incoming task off the queue and, once it is ready to be sent (e.g., once data transfers finished), sends it to the *funcX* service, receiving a *physical task-id*. Similarly, if a

backup (physical) task needs to be sent for a (virtual) task, the virtual task is simply added to the scheduling queue again, and once scheduled, the task-translation table is updated. Every time the client requests a (virtual) task's status, the task-translation table is used to fetch statuses for all corresponding (physical) tasks.

This change to our task-tracking mechanism allows us to transparently send multiple copies of a task requested by a client. In such a case, we need only record the first result that is returned by the task. Future work could explore using this mechanism to send multiple copies of tasks which need to meet service-level objectives, in the hope of more reliable execution times.

CHAPTER 5

EVALUATION

We evaluate *Delta* in two ways. First, using *micro-experiments* to show that the primary features of *Delta* work well in situations which are designed to highlight the need for a holistic computing ecosystem; and second, using *macro-experiments* to show that *Delta* yields satisfactory performance when put under load with realistic workloads, which were inspired by existing FaaS benchmark suites [47]. These two objectives are described in Sections 5.2 and 5.3, respectively.

5.1 A Heterogeneous Testbed

We evaluate our approach using a testbed consisting of various compute devices, which we have assembled to model the differences one can expect to find across widely available computing devices. This testbed, as described in detail in Table 5.1, features Raspberry Pis, HPC clusters, general-purpose GPUs, an AWS instance, as well as several commodity desktop computers. Thus, this testbed represents a broad range of computing resources, from slow edge devices and desktops to high-performance cluster nodes and specialized accelerators.

5.2 Micro-Experiments

We first present three experiments which highlight different features of *Delta*, and verify that *Delta* is a system capable of transparently routing 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

Description	**Device Info** *(CPU Info, Logical Cores, Memory)*	**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_desktop	1
Average Desktop *(UChicago CS)*	Intel Core i7-6700, 8-core, 8GB	avg_desktop	1
Fast Desktop *(UChicago CS)*	Intel Core i7-8700, 12-core, 16GB	fast_desktop	1
Cloud CPU *(AWS T3a.medium)*	AMD EPYC 7571, 2-core, 4GB	aws_t3a_med	1
Manycore CPU *(Chameleon)*	Intel Xeon E5-2670, 48-core, 125GB	manycore	2
GPU Node *(Chameleon)*	Nvidia Quadro RTX 6000 GPU (Intel Xeon Gold 6126, 48-core, 187GB)	gpu	2

Table 5.1: Our heterogeneous testbed for experiments. A *funcX* endpoint was deployed on each of the 11 devices described.

strategies. The third experiment demonstrates that, when faced with endpoint failures and task slowdowns, *Delta* quickly recovers and reliably completes tasks.

5.2.1 Learning Under a Small Load

Perhaps the simplest situation in which we can expect to see the benefits of heterogeneity is when the only factor that needs to be considered is function performance. Figure 5.1 shows the execution times observed in an experiment where we repeatedly requested a matrix multiplication computation on a regular schedule. Each function execution involved multiplying two square matrices of size 1000x1000, fifty times. There were a total of 100 such function executions requested, at the rate of 2 tasks per second. This can be thought of as a close approximation for an application that regularly requests a neural-network inference.

Both the `fastest-endpoint` and `smallest-ETA` strategies spend the first few function

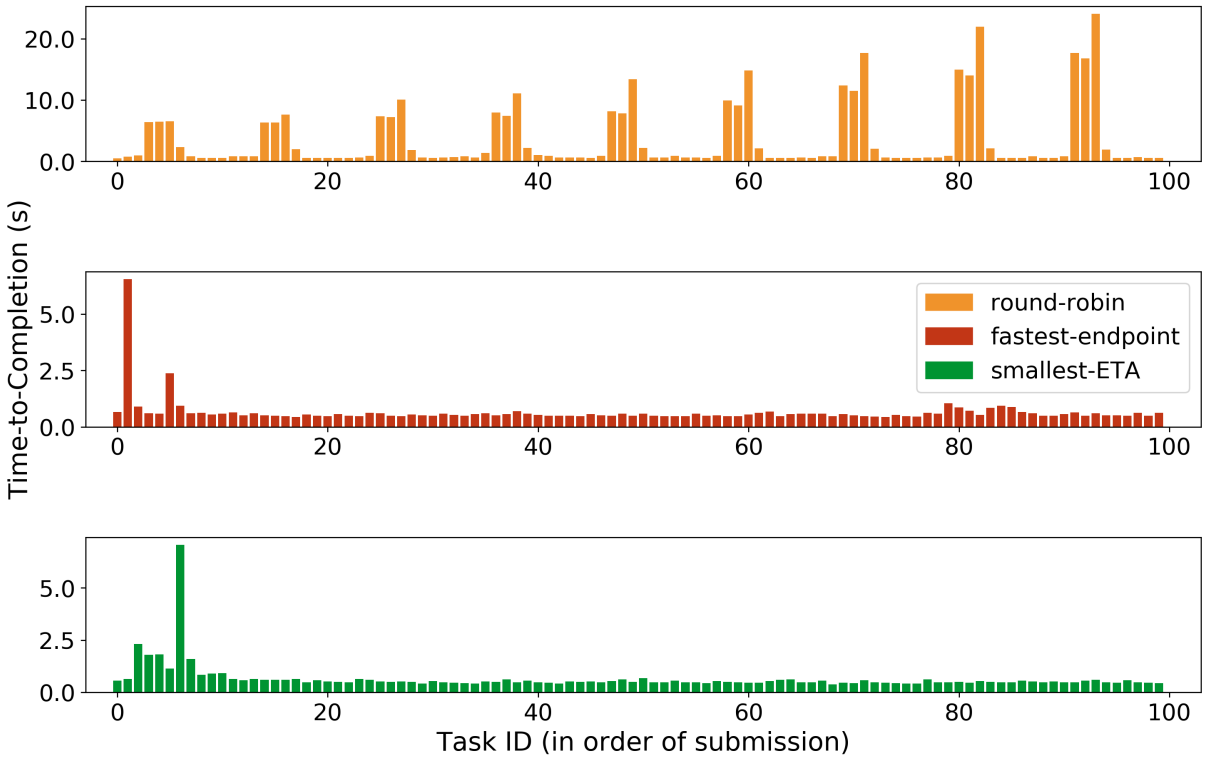

Figure 5.1: Learning function performance behavior under a small load. We see that *Delta* learns to send this specialized workload to the endpoints it is most suited to, drastically reducing execution time. *Note the different scale for* `round-robin`.

executions exploring the different (groups of) endpoints in the testbed. After this, both strategies learn to favor the most suitable endpoints for matrix multiplication (unsurprisingly, these are the GPU endpoints). On the other hand, the naive `round-robin` strategy performs much worse, alternating between endpoints which are slow and fast for this workload. Moreover, the slowest endpoints are incapable of meeting the arrival rate of tasks, and thus function requests must be queued. This situation explains why the slowest `round-robin` execution times become slower as time goes on. In summary, when not under load, it is simple to optimally schedule workloads that strongly favor a certain type of endpoint. While the `fastest-endpoint` strategy performs optimally in this simple case, we will soon see that this is not always the case.

5.2.2 Data Transfer Trade-offs

The second major factor that contributes to the cost of function execution in our grid-FaaS model is that of transferring data. To demonstrate this, we conducted an experiment in which we requested a stream of tasks (of different runtimes) which required certain files for execution. These files were located on a single endpoint in our testbed, `avg-desktop`. Thus, for every execution, the *Delta* service had to make a decision of whether to run the computation on this endpoint (incurring no transfer cost), or to run the computation on a different endpoint (incurring a transfer cost). After 18 exploration tasks (1 of each size for each endpoint group) which were used by *Delta* to learn function performance behavior, we requested a total of 100 tasks, at a rate of 1 task every 2 seconds. For each task, there were two 1KB files required for execution. Each time, the task was randomly chosen to be one of three different computation loads — specifically, spin-loop tasks which incremented a counter up to a value of 2^{24}, 2^{26}, or 2^{28}.

Figure 5.2 shows the results of this experiment. The `round-robin` strategy naively cycles through the available endpoints, thus incurring a (sometimes quite large) data transfer cost for all but one endpoint. The `fastest-endpoint` strategy also ignores data transfer costs, and simply chooses the endpoint which has historically run each task the fastest. Since the files were located on the `avg-desktop` endpoint, which has slower CPU cores than at least 3 of the other endpoints, the `fastest-endpoint` strategy always chose to offload the computation to a different endpoint, incurring small but not negligible transfer costs. On the other hand, the `smallest-ETA` strategy, by taking into account transfer costs, chose to offload the computation to a different endpoint only some of the time. Closer analysis of the figure shows that the following pattern occurred multiple times: the `avg-desktop` endpoint was chosen for execution several times, as indicated by the lack of transfer cost, until enough pending tasks piled up (shown by the growing size of the purple bars), at which point it became favorable to offload the next task to a different endpoint and incur a transfer cost.

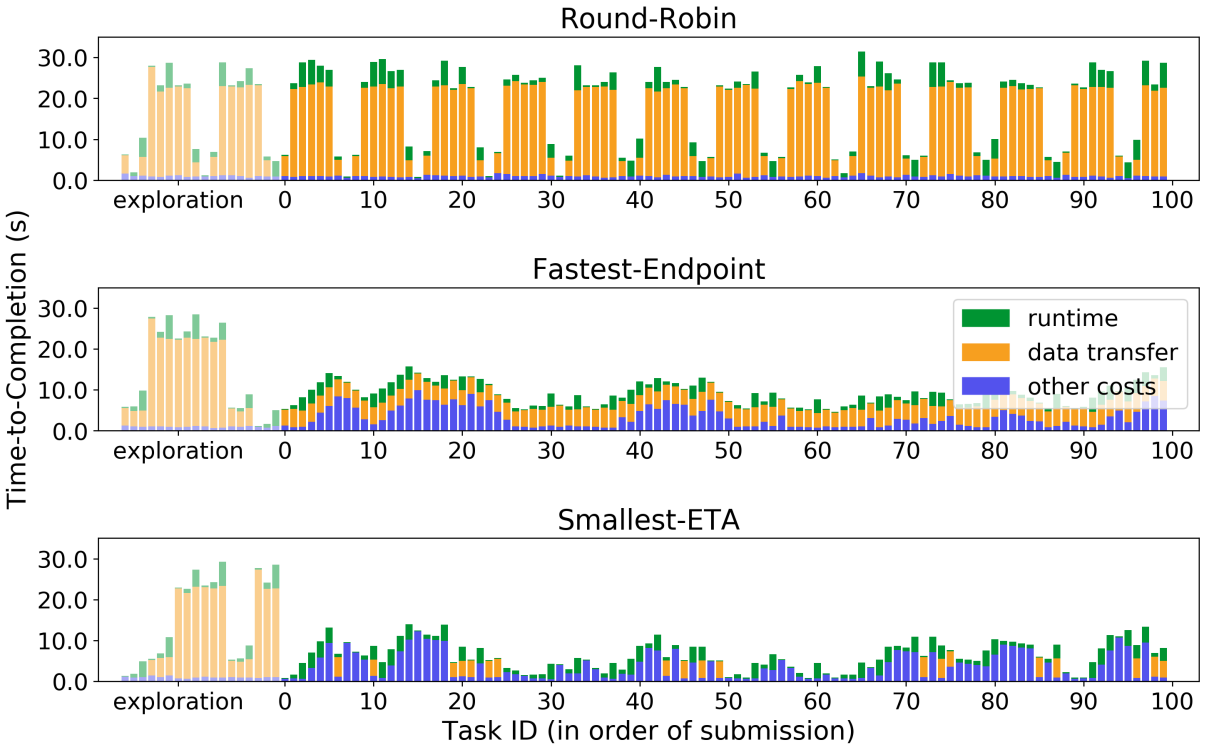

Figure 5.2: Accounting for the cost of transferring data as part of function executions.

We note that, even though transfer time predictions are often imprecise due to ever-changing network conditions, we only need rough approximations to be able to make these trade-off decisions.

5.2.3 Tolerating Failures and Slowdowns

Our next experiment showcases how *Delta* automatically detects task delays and endpoint loss, and sends backup tasks to other endpoints to improve the chances of task completion. The details of this experiment are similar to those in Section 5.2.1 — a constant stream of matrix multiplication tasks at the rate of 1 task every 3 seconds. Each task consisted of multiplying 50 different pairs of square matrices, of size 2500x2500. Note that for this experiment, we used only one of the two GPU endpoints, in order to demonstrate what

happens when an accelerator fails and recovers. After the 30th task, the `gpu-1` endpoint was taken offline, and after the 60th task, it was brought back online.

Figure 5.3 shows the results of this experiment. Note that only the `smallest-ETA` strategy is shown. As usual, the first few function executions are spent in exploration, after which, the execution time is consistently small. Since the period from task 30 to task 60 is quite small, the *Delta* service does *not* observe a missed heartbeat from the `gpu-1` endpoint. However, it does see task delays for each of these tasks. When the *Delta* service hasn't received results for these tasks in more than twice the expected execution time, it sends backups for these tasks to other (noticeably slower) endpoints. After task 60, `gpu-1` resumes responding to task requests as expected, and so, execution returns to normal.

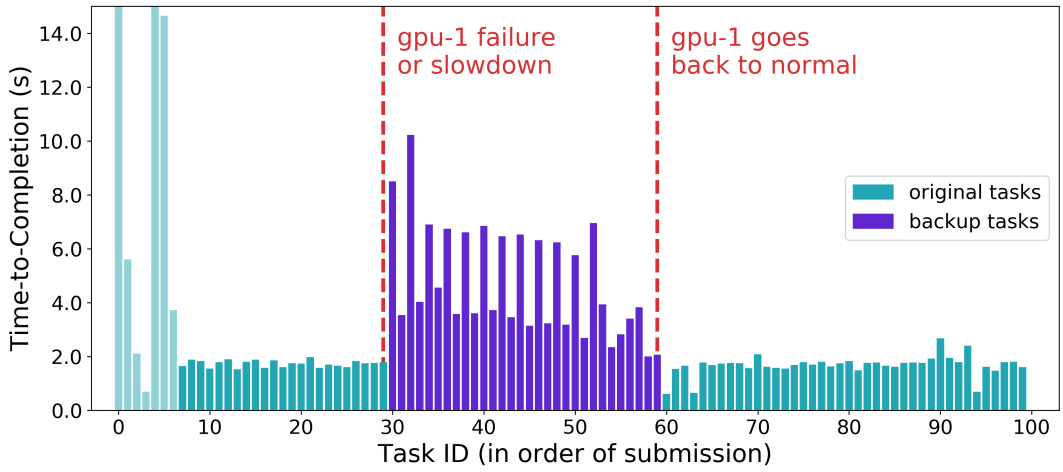

Figure 5.3: Tolerating failure of endpoints and slowdown of tasks with automatic delay-detection and backup tasks.

5.3 Macro-Experiments

The experiments in this section aim to show how *Delta* performs when subjected to high levels of load, with tasks being scheduled in large "bursts" (as FaaS platforms are often subject to), as opposed to in a gradual stream of incoming requests. The first experiment

shows how throughput and time-to-completion vary when many copies of the same task are run. The second experiment shows how the size of input must be taken into account when making scheduling decisions. The third experiment shows that *Delta* performs well when subjected to multiple different types of tasks at the same time.

5.3.1 Overloading Tasks

As opposed to the Micro-Experiments described above where it is often optimal to send tasks to the fastest endpoint, when under load, we would expect a smart scheduling system to distribute tasks across endpoints in a way that maximizes task throughput, while keeping the execution time of each task low. To put this to the test, we designed the following experiment. We requested 500 tasks, in bursts of 50 at a time, where each task was an embarrassingly parallel Map-Reduce computation. Each function execution first kicked off as many concurrent processes as there were CPU cores on the machine, and then evenly distributed work amongst them. Here, the computation for each execution consisted of counting up to a total of 10^8 between the parallel processes.

Figure 5.4 shows the results of this experiment. We see 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. This leads to tasks quickly piling up on this endpoint (recall that tasks are executed in FIFO order), which explains the higher median time-to-completion. The `smallest-ETA` 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-ETA` strategy consistently keeps the time-to-completion for tasks low. On the flip side, the `smallest-ETA` strategy also consistently shows a task throughput that is several times higher than the other strategies.

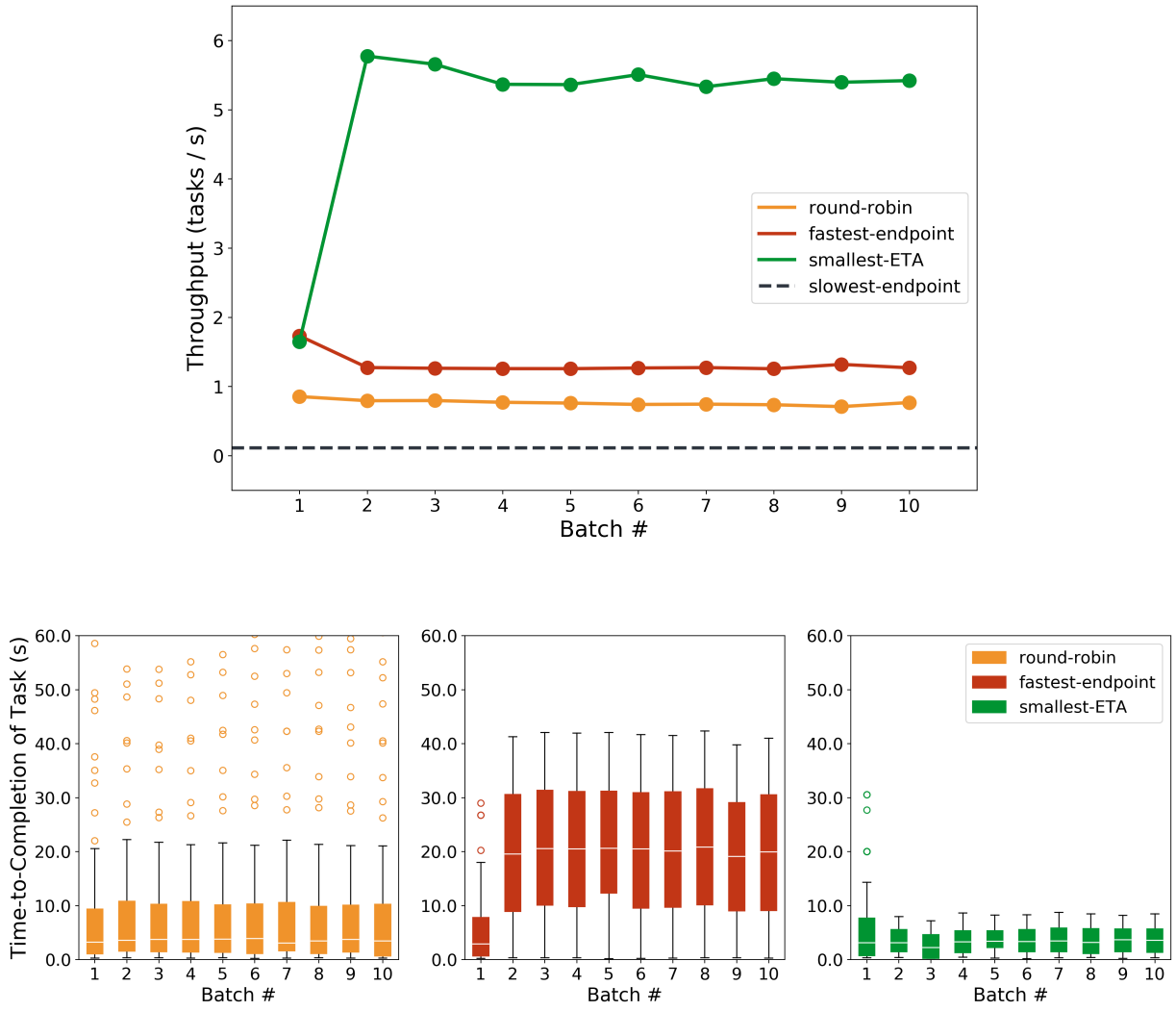

Figure 5.4: Overloading *Delta* with MapReduce tasks shows that, compared to baseline strategies, ETA prediction significantly reduces time-to-completion and boosts throughput.

The above results don't explicitly demonstrate how tasks are distributed across the endpoints. Moreover, it is important to ask the question of what it means for a sequence of scheduling decisions to be "optimal". To take a step towards this question, and with the

goal of explainability, we ran the same experiment as above, but with the simplest possible task — spin-loop, which simply involves incrementing a counter up to a certain value (of 10^8). Figure 5.5 shows the distribution of tasks per endpoint produced by the `smallest-ETA` 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. If we were to ignore other execution latencies for a moment, we would observe that the optimal distribution of tasks is exactly in proportion to the relative speeds of the different endpoints for this task. Figure 5.5 demonstrates that our `smallest-ETA` strategy naturally reaches a close approximation of this distribution, allowing us to claim that its scheduling decisions are close to "optimal".

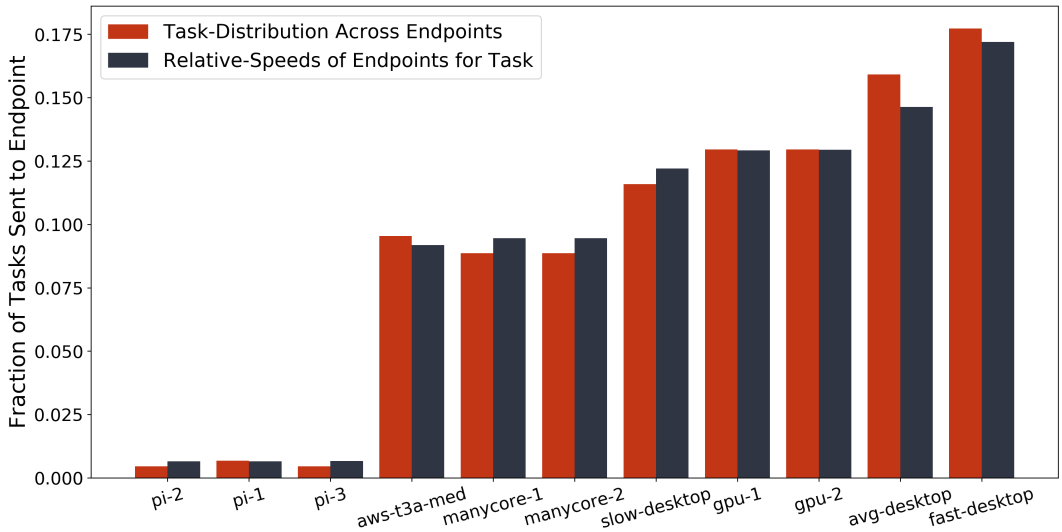

Figure 5.5: Distribution of endpoints when the system is overloaded with spin-loop tasks, demonstrating that when tasks are large enough that latency is negligible, ETA prediction distributes tasks nearly optimally.

It is worth noting that the optimality of this distribution holds only if we ignore all other execution costs (such as *funcX* overhead, network latency, etc.). Thus, the observed scheduling decisions would perfectly reflect the relative speeds of the endpoints only in

the limit case, i.e., where the task runtimes are infinitely larger than all other latencies. Conversely, if the task runtimes are infinitely small, the optimal scheduling decision would be to round-robin between the endpoints, since the (common) overhead would be the only cost worth considering.

5.3.2 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 inputs to a function result in different runtimes. Note that this is a common occurrence that can be observed in many workloads, from convolutional neural-networks to operations on Pandas DataFrames. We considered spin-loop tasks, i.e., tasks involving incrementing a counter to a given integer value, with three different choices of input — 2^{22}, 2^{25}, and 2^{28}. We scheduled a total of 600 tasks, divided into 10 bursts of 60 tasks each. Each burst consisted of 20 tasks of each of the three input sizes, in a random order.

The results are shown in Figure 5.6. As usual, 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 bursts 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 runtime for the function, both with and without accounting for input sizes. This is because this fastest endpoint, unsurprisingly, is the best for *each* of the three sizes. The figure also shows how the `smallest-ETA` strategy performs, with and without taking into account input size in its runtime predictions. Without considering how runtimes are affected by varying inputs, the `smallest-ETA` strategy has a high variance in the observed throughput. This is because it obliviously sends tasks to where the function has been performing well recently. Occasionally, it sends tasks to

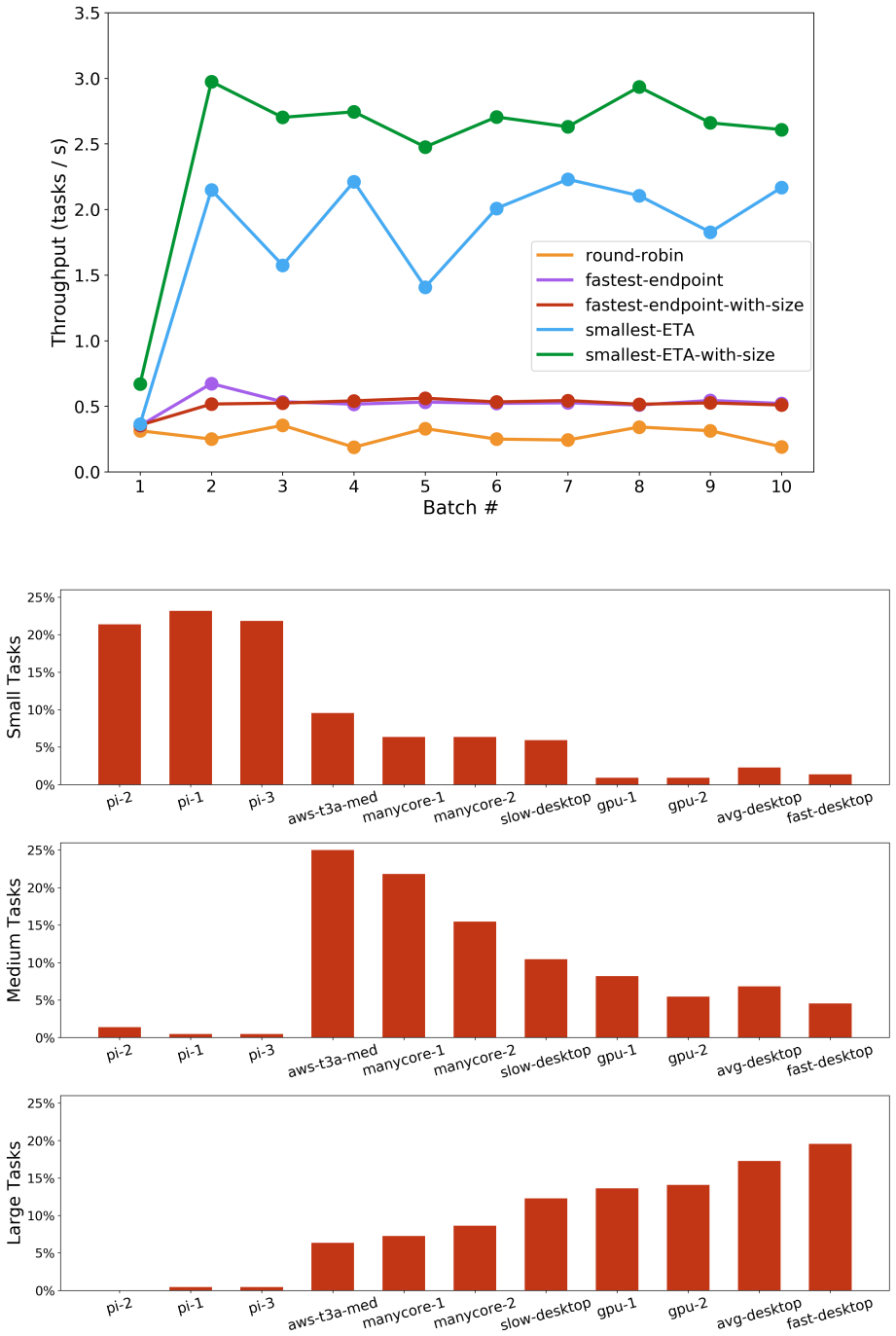

Figure 5.6: Running tasks with multiple input sizes shows that treating different input sizes differently yields increases in performance. When under load, it is beneficial to send smaller tasks to slower endpoints, even Raspberry Pis.

the correct endpoints, but other times, it ends up sending tasks to endpoints which have previously offered short runtimes only because they were given small inputs in the (recent) past. When the `smallest-ETA` strategy takes into account task sizes, its throughput remains consistently high.

Perhaps the most interesting part of this experiment is how the `smallest-ETA` strategy distributes tasks of different sizes across the endpoints, which is also depicted in Figure 5.6. Note that the endpoints in the figure are ordered by their speeds when running this function, just like Figure 5.5. As the figure shows, the smallest of the tasks are sent to the slowest endpoints, the medium-sized ones to the mediocre endpoints, and the largest tasks to the fastest endpoints. Upon some reflection, one should note that this is, in fact, what we would expect to happen for "optimal" scheduling. It makes sense to send the short-running tasks to the weaker endpoints (even Raspberry Pis) because the differences in task runtimes for these are negligible compared to other overheads. Conversely, it makes sense to send the long-running tasks to the more powerful endpoints because the differences in runtimes across endpoints for these tasks are quite large. So, we glean that when put under load with tasks of different sizes, we must take advantage of even the slowest of our endpoints for improved performance.

5.3.3 Multiple Heterogeneous Tasks

In our final experiment, we pushed *Delta* by subjecting it to multiple heterogeneous tasks, each of multiple different sizes, all at the same time. In the absence of a production workload, we developed synthetic workloads that aimed to emulate real-world FaaS applications by sending bursts of different types of tasks, which can be thought of as coming from different applications. The three workloads we ran were:

- *Matrix Multiplication:* Each task consisted of 20 multiplications of square matrices using Tensorflow, of one of three sizes — 2^8, 2^9, and 2^{10}.

- *MapReduce:* Each task consisted of starting as many parallel processes as there were CPU cores on the machine, and then dividing between them an embarrassingly parallel task. The task was incrementing a counter up to one of three possible inputs — 2^{24}, 2^{26}, and 2^{28}.

- *File I/O:* Each task consisted of writing out a file of the input size, and then reading it back into memory, checking that the file's contents were as expected. There were three possible file sizes — 2^{20}, 2^{22}, and 2^{24} bytes.

There were a total of 1000 tasks run, in bursts of 100 tasks at a time. The order of the tasks (and inputs) was randomly shuffled at the beginning of the experiment. Figure 5.7 depicts the throughputs observed with different scheduling strategies. Note that input size prediction was used in all cases.

Unsurprisingly, `round-robin` is too naive to be able to efficiently distribute tasks to the different endpoints. The `fastest-endpoint` strategy learns to send each function to the endpoint it runs fastest on, but as we have observed before, fails to account for any other factors. Because of this, it shows high variance in throughput, getting lucky in some bursts, and overwhelming a handful of endpoints in others. The `smallest-ETA` strategy quickly learns how to allocate the bursts of tasks to endpoints, and maintains a consistently high task throughput. Figure 5.8 shows how each of the strategies allocates tasks to the different endpoints. `round-robin` of course distributes tasks evenly across the endpoints without looking at any task features (the slightly uneven peaks are due to the random order of tasks submitted). `fastest-endpoint`, as expected, overwhelms the handful of endpoints which it determines to be the fastest for each of the 3 types of tasks. For instance, most file-I/O tasks are sent to `fast-desktop` (fastest CPU and fast disk), whereas most matrix multiplication tasks are sent to the GPUs. On the other hand, the distribution observed for `smallest-ETA` is much more nuanced. The pattern shown depends on both the type of the tasks and their sizes. We see that the smallest matrix multiplication tasks are almost exclusively sent to the

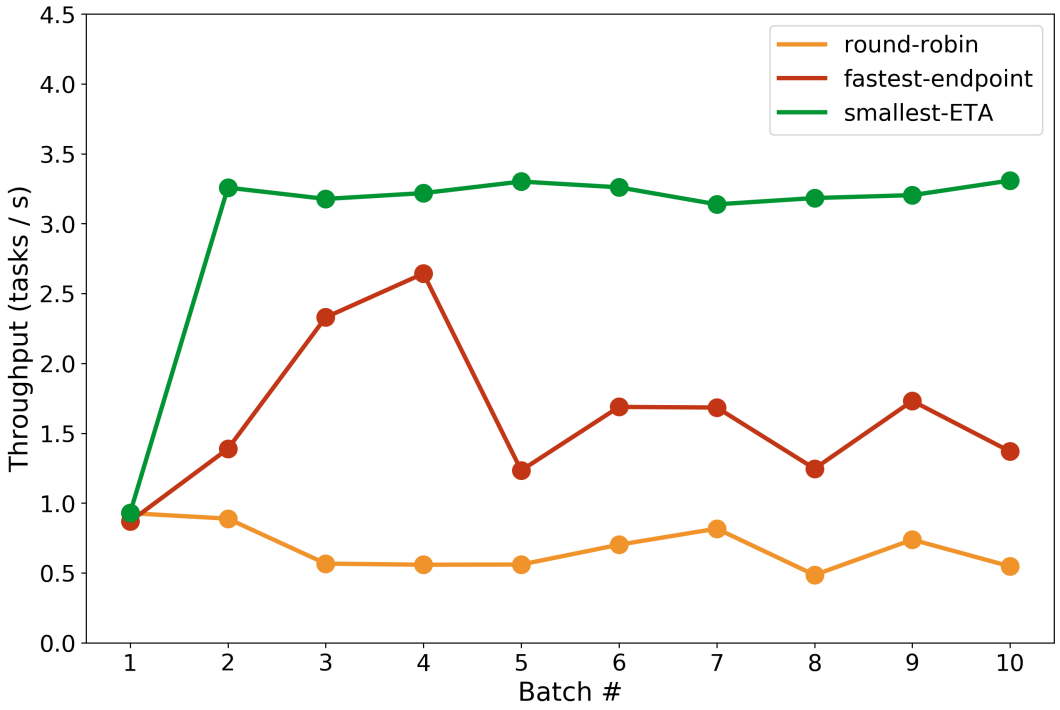

Figure 5.7: Performance of *Delta* when subjected to three different types of tasks (Matrix multiplication, MapReduce, and File I/O), each of multiple input sizes.

Raspberry Pis, whereas the longer running ones are sent to the GPU and manycore endpoints (where Tensorflow can exploit massive parallelism). Similarly the larger MapReduce tasks are sent to the manycore endpoints, and the larger File-I/O tasks go to endpoints with fast disk speeds and fast CPU cores, whereas the smaller of these tasks are distributed amongst various slower endpoints.

This experiment, along with the ones before it, underscores the claim that scheduling function executions in an environment with heterogeneity is a non-trivial undertaking that requires modeling the multitude of complexities involved in running a computation remotely. While *Delta* achieves significant improvements over baselines, there is no dearth of potential for further enhancements.

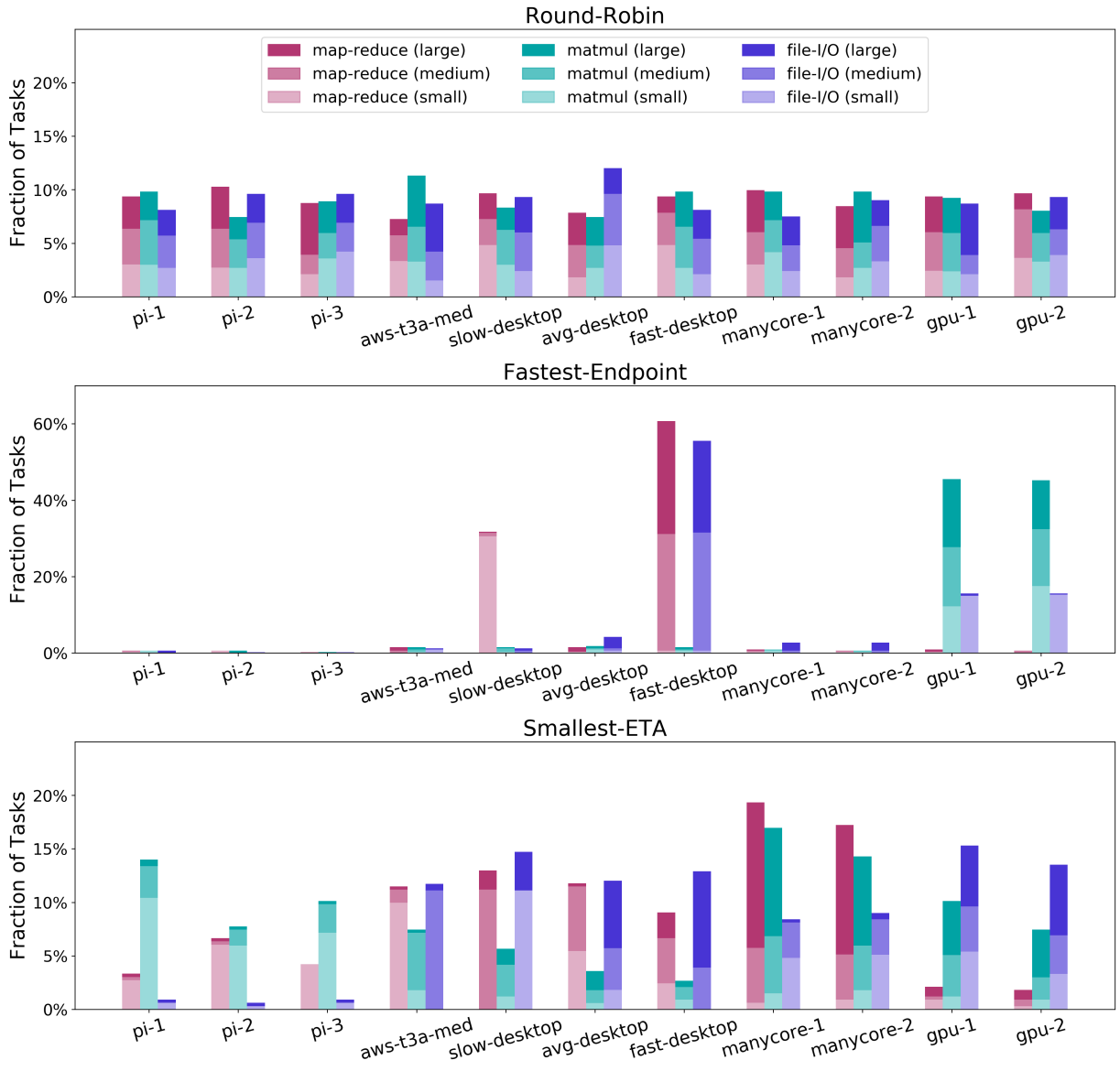

Figure 5.8: Distribution of tasks across endpoints when subjected to three different types of tasks, each of multiple sizes. Increased performance is owed to the use of both task type and input size when choosing where to run. *Note the different scale for* `fastest-endpoint`.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Any effort to model the computing continuum must account for the many complexities that naturally arise in heterogeneous computing environments. In the distributed function execution setting, this means accounting for compute times, network latencies, cold starts, queuing delays, as well as transfer times when the movement of data is required. A careful analysis of how each of these quantities can trade-off with each other allows us to build a model of computation that can fluidly allocate tasks while minimizing execution time.

6.1 Summary

Previous work that has focused on optimizing function execution has largely been in homogeneous settings, whether in commercial clouds or in research environments. Since these kinds of trade-offs only become apparent in heterogeneous grid-like settings, the need to model them in a comprehensive manner has not been a focus of prior work. In cases where existing function execution frameworks do provide some semblance of heterogeneity, the choice of where to execute a function is left up to the user. Moreover, this choice is usually made at the time when a function is registered, after which it remains fixed. But, as our analysis has shown, the best choice of where to execute a function is constantly evolving, and so, cannot be decided a priori.

In this work, we have presented *Delta*, a function-serving framework built to account for the complexities of heterogeneous execution. *Delta* embraces the constantly evolving nature of heterogeneous environments, and is able to make rapid scheduling decisions dynamically, achieving significant performance gains over baseline scheduling strategies. *Delta* learns to predict different components of function execution in an online fashion. This includes learning function performance behavior by exploring execution on different endpoints, predicting

data transfer time based on empirical transfer information, and accounting for other latencies such as cold-start times, queuing times, and communication overhead. *Delta* represents a starting point in the formidable journey of coding the computing continuum.

6.2 Limitations

While *Delta* is the first complete effort towards capturing the fluidity of heterogeneous function-serving environments, it has several limitations. The design of *Delta* is inherently based on predicting different components of function execution. This has two immediate shortcomings. The first shortcoming is that *Delta*'s performance depends heavily on the accuracy of its various predictors. Without accurate predictions for function performance, transfer time, and cold-start latency, *Delta* would not be able to schedule tasks efficiently. This is especially concerning in cases where functions do not perform deterministically, and so, do not lend themselves well to prediction. The second shortcoming is that *Delta*'s online learning approach may not scale well. For instance, *Delta*'s current prediction mechanisms for runtime and transfer time involve maintaining a sizeable number of prediction models, i.e., runtime predictors for all pairs of functions and endpoint groups, and pairwise predictors for endpoint-to-endpoint data transfers. This would likely become a scalability bottleneck.

The function-serving problem that *Delta* is built to solve is limited in its scope. *Delta* can only handle individual function executions, but user workloads in FaaS environments often involve executing multiple inter-dependent functions. This limitation restricts the complexity of workloads *Delta* can orchestrate. Moreover, *Delta* makes strong assumptions about how tasks are executed on endpoints. In particular, *Delta* assumes that each endpoint only runs one task at a time, and that each task can only make use of one endpoint. A production system similar to *Delta* will need to allow for more complex execution capabilities.

Delta is also limited by the execution objectives it is able to meet. Its current implementation is tailored towards minimizing the time-to-completion of individual tasks. An

ideal framework for fluid function execution would allow users to choose which objective they would like to optimize for, such as minimizing monetary cost, restricting power usage, increasing resource utilization, etc.

6.3 Future Work

There are many directions that future work could take towards improving *Delta* and realizing the vision of the computing continuum. *Delta*'s runtime prediction approach could be improved and made more scalable by using (deep) transfer learning. This is based on the hypothesis that a function's runtimes on one device are good predictors of its runtimes on other devices (e.g., across different Nvidia GPU models). Similarly, in a white-box model of prediction, we could apply natural language processing techniques to learn a function's behavior from its similarity to previously observed functions. This would greatly reduce our overhead of maintaining many prediction models.

Predictions for cold start latencies and data transfer times could also be substantially improved. *Delta* does not currently account for the latency of container instantiation — this would be essential to fully model cold start costs. Containerization costs would inevitably vary across heterogeneous devices, and so, future work is needed to accurately model these latencies. As for data transfer times, *Delta* currently maintains pairwise models based on empirically observed transfer rates. In reality, transfer times are prone to variance due to factors such as network congestion and the load on the network interface cards of the source and destination. Accounting for these factors, and abandoning our pairwise prediction approach for a "universal" model of transfer rate prediction would be a valuable contribution.

There will inevitably be cases where task runtimes are unpredictable, and so, *Delta*'s prediction-based scheduling strategies do not perform well. Overcoming this limitation would involve supplementing *Delta* with orthogonal scheduling strategies which do not share this dependence on accurate predictions. Cluster job-management frameworks such as Slurm [48]

have often used heuristic-based strategies that involve backfilling, which allows them to make reasonable scheduling decisions without the need for runtime predictions. However, jobs in such contexts are almost always provided time and resource constraints by the user. In the cloud setting, providers like Amazon use spot-filling techniques to let users run on unused instances, with no guarantees on allocation time [49]. It would be interesting to see if any such heuristic strategies could be translated over to the FaaS setting.

Several modern FaaS systems provide support for chaining function executions, e.g., for workflows that require multiple interdependent analyses. Since *funcX* does not currently support such chaining, we have not explored its effects on scheduling. If our goal is to minimize the execution time of a workflow consisting of interdependent functions, our scheduling decisions must be smarter than simply minimizing the execution time of individual functions. Existing work on scheduling computation graphs in contexts outside FaaS would likely be useful in this endeavor. Moreover, in this work, we have assumed that tasks are to be scheduled in FIFO order, and that each endpoint can only execute one task at a time. If we were to relax these assumptions and allow tasks requested within a short window to be scheduled out-of-order, the complexity of our function-scheduling decisions would likely increase substantially.

Future work may also explore function scheduling in heterogeneous environments when working with additional constraints, and when optimizing for objectives other than execution time. The decision of where to send tasks would likely be affected by the addition of service-level objectives, such as execution deadlines. Constraints could also be placed on the location of execution, for example, if functions or data may only be offloaded to certain devices, for security or legal reasons. Finally, for many use cases, it is important to not only minimize execution time, but to also provide low monetary and energy costs. To accommodate different notions of cost, our analysis of trade-offs would have to evolve substantially, yielding yet another exciting avenue for future work.

REFERENCES

[1] Govind P Agrawal. Optical communication: its history and recent progress. In *Optics in Our Time*, pages 177–199. Springer, Cham, 2016.

[2] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. *funcX*: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC 20. Association for Computing Machinery, 2020.

[3] Ian Foster. Coding the Continuum. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–1. IEEE, 2019.

[4] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07):P07027, 2018.

[5] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[6] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[7] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[8] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The Globus striped GridFTP framework and server. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 54–54. IEEE, 2005.

[9] Steven Tuecke, Rachana Ananthakrishnan, Kyle Chard, Mattias Lidman, Brendan McCollam, Stephen Rosen, and Ian Foster. Globus Auth: A research identity and access management platform. In *2016 IEEE 12th International Conference on e-Science (e-Science)*, pages 203–212. IEEE, 2016.

[10] Amazon Lambda. `https://aws.amazon.com/lambda/`. Accessed: 2020-06-29.

[11] GCP Functions. `https://cloud.google.com/functions/`. Accessed: 2020-06-29.

[12] Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`. Accessed: 2020-06-29.

[13] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *2008 grid computing environments workshop*, pages 1–10. Ieee, 2008.

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[15] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10–10):95, 2010.

[16] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC 19, pages 25–36. Association for Computing Machinery, 2019.

[17] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, number 130–136. Citeseer, 2015.

[18] David Kirk et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[19] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[22] Tensor Processing Units. `https://cloud.google.com/tpu/`. Accessed: 2020-06-29.

[23] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Heterodoop: A mapreduce programming system for accelerator clusters. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 235–246, 2015.

[24] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[25] Richard D Hornung and Jeffrey A Keasler. The RAJA portability layer: overview and status. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[26] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[27] Chris Gregg, Michael Boyer, Kim Hazelwood, and Kevin Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *Workshop on Applications for Multi-and Many-Core Processors (A4MMC)*, pages 1–12, 2011.

[28] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[29] Amazon Greengrass. `https://aws.amazon.com/greengrass/`. Accessed: 2020-06-29.

[30] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[31] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with OpenLambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[32] Apache OpenWhisk. `http://openwhisk.apache.org/`. Accessed: 2020-06-29.

[33] IBM Cloud Functions. `https://ibm.com/cloud/functions`. Accessed: 2020-06-29.

[34] Kubeless. `https://kubeless.io/`. Accessed: 2020-06-29.

[35] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188. IEEE, 2018.

[36] Firecracker. `https://firecracker-microvm.github.io/`. Accessed: 2020-06-29.

[37] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 57–70, 2018.

[38] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[39] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards High-Performance Serverless Computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.

[40] Amoghvarsha Suresh and Anshul Gandhi. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 19–24, 2019.

[41] Chavit Denninnart, James Gentry, and Mohsen Amini Salehi. Improving robustness of heterogeneous serverless computing systems via probabilistic task pruning. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 6–15. IEEE, 2019.

[42] Bryce Allen, John Bresnahan, Lisa Childers, Ian Foster, Gopi Kandaswamy, Raj Kettimuthu, Jack Kordas, Mike Link, Stuart Martin, Karl Pickett, et al. Globus online: Radical simplification of data movement via SaaS. *Preprint CI-PP-5-0611, Computation Institute, The University of Chicago*, page 43, 2011.

[43] Kyle Chard, Steven Tuecke, and Ian Foster. Efficient and secure transfer, synchronization, and sharing of big data. *IEEE Cloud Computing*, 1(3):46–55, 2014.

[44] Allcock, William E. and Allen, Benjamin S. and Ananthakrishnan, Rachana and Blaiszik, Ben and Chard, Kyle and Chard, Ryan and Foster, Ian and Lacinski, Lukasz and Papka, Michael E. and Wagner, Rick. Petrel: A programmatically accessible research data service. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC 19, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Zhengchun Liu, Prasanna Balaprakash, Rajkumar Kettimuthu, and Ian Foster. Explaining wide area data transfer performance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 167–178, 2017.

[46] Kevin Harms, Ti Leggett, Ben Allen, Susan Coghlan, Mark Fahey, Carissa Holohan, Gordon McPheeters, and Paul Rich. Theta: Rapid installation and acceptance of an XC40 KNL system. *Concurrency and Computation: Practice and Experience*, 30(1):e4336, 2018.

[47] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[48] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[49] Amazon Spot Instances. `https://aws.amazon.com/ec2/spot/`. Accessed: 2020-06-29.