Scheduling Hardware-Accelerated Cloud Functions

Jessica Vandebon¹ · Jose G. F. Coutinho¹ · Wayne Luk¹

Received: 30 September 2020 / Revised: 11 April 2021 / Accepted: 26 July 2021 / Published online: 27 October 2021
© The Author(s) 2021

Abstract

This paper presents a Function-as-a-Service (FaaS) approach for deploying managed cloud functions onto heterogeneous cloud infrastructures. Current FaaS systems, such as AWS Lambda, allow domain-specific functionality, such as AI, HPC and image processing, to be deployed in the cloud while abstracting users from infrastructure and platform concerns. Existing approaches, however, use a single type of resource configuration to execute all function requests. In this paper, we present a novel FaaS approach that allows cloud functions to be effectively executed across heterogeneous compute resources, including hardware accelerators such as GPUs and FPGAs. We implement heterogeneous scheduling to tailor resource selection to each request, taking into account performance and cost concerns. In this way, our approach makes use of different processor types and quantities (e.g. 2 CPU cores), uniquely suited to handle different types of workload, potentially providing improved performance at a reduced cost. We validate our approach in three application domains: machine learning, bio-informatics, and physics, and target a hardware platform with a combined computational capacity of 24 FPGAs and 12 CPU cores. Compared to traditional FaaS, our approach achieves a cost improvement for non-uniform traffic of up to 8.9 times, while maintaining performance objectives.

Keywords Cloud computing · Heterogeneous computing · FaaS · FPGA

1 Introduction

Cloud computing has become increasingly popular in the last decade, allowing businesses and researchers to offload computations to cloud data centres, drastically reducing their operating costs. For instance, IaaS (Infrastructure-as-a-Service) systems provide on-demand virtual instances of compute resources, such as servers, routers and storage, which can be operated in a similar fashion to their physical counterparts. Moreover, PaaS (Platform-as-a-Service) systems automatically manage provisioned resources, with services such as *load-balancing*, which maximises performance and availability; and *elasticity*, which dynamically scales resources according to demand in order to reduce tenancy costs.

Recently, a new cloud model called FaaS (Function-as-a-Service) has emerged to simplify resource pricing and management over both PaaS and IaaS. FaaS is triggered by web requests which execute *cloud functions*. These functions can be supplied by providers, or deployed by clients. With FaaS, clients pay per serviced request (Figure 1(b)), offloading resource management responsibilities to the provider. In contrast, IaaS and PaaS models charge tenants for all allocated resources throughout the period in which they have them, regardless of whether they are used or idle (Figure 1(a)).

Cloud functions are *stateless*, and are realised by containers, which provide an ephemeral and isolated runtime environment to execute computations. Note, however, that functions can still store, load, and share state by accessing external databases. FaaS has found its place in major cloud platforms (e.g. AWS Lambda [5], Microsoft Azure Functions [14], and Google Cloud Functions [10]), supporting real-time data processing (batch and stream processing), Internet of Things (IoT), and edge computing.

In this paper, we present SLATE (Heterogeneous cLOUD mAnagement for FunCTion-as-a-service SysTEms), a novel FaaS approach (Figure 1(c)) designed to leverage

✉ Jessica Vandebon
jessica.vandebon17@imperial.ac.uk

Jose G. F. Coutinho
gabriel.figueiredo@imperial.ac.uk

Wayne Luk
w.luk@imperial.ac.uk

¹ Imperial College London, London, United Kingdom

Figure 1 Comparison between (a) resource-oriented cloud models, (b) traditional FaaS, and (c) SLATE FaaS.

heterogeneous cloud compute resources, such as CPUs and FPGAs, in order to provide further performance and cost benefits over traditional homogeneous FaaS approaches.

In contrast to current FaaS offerings, SLATE is designed for scenarios where functions have very different computational requirements and performance objectives. Current solutions are restricted to a single type of resource configuration, which is instantiated to execute each submitted request. While this approach is well-suited for function requests that have similar computational requirements and thus can be served by a single type of resource configuration, it does not address performance and pricing concerns in cases where requests have very different computational requirements, for example, in domains such as High Performance Computing (HPC) and Artificial Intelligence (AI).

The work reported in this paper is a refinement of and extension to the approach presented in [17]. In particular, we combine SLATE's task scheduler and auto-scaler into a single heterogeneous scheduler component to improve the system's effectiveness. In addition, we provide an in-depth look at performance modelling techniques used in our approach.

The main contributions of this paper are as follows:

1. The SLATE FaaS architecture and management mechanisms;

2. The implementation of a simulated FaaS prototype with the above architecture;
3. An evaluation of our prototype targeting three application domains, namely machine learning, bio-informatics, and physics, on FPGA and CPU resources. We compare SLATE to current FaaS systems taking into account performance and cost.

2 Background

2.1 Motivation

Consider a scenario where an application employs two cloud functions that perform Machine Learning (ML) tasks, namely: *training* and *inference*. Training tasks involve sending large chunks of data at regular time intervals, while inference tasks are smaller and happen irregularly according to user demand. Both function types access an external database to store and load the ML model: training updates the model continuously, while inference uses the updated model. In this example, we have two distinct task types with specific performance requirements: training tasks process bulk data and are more computationally intensive, while inference tasks are smaller and have lighter computation requirements.

Current FaaS solutions are not designed to support such scenarios efficiently, where tasks have very different computation requirements. In particular, clients must identify a single resource configuration (e.g. a 4 core CPU with 512MB of RAM) to service every incoming request for a given function. Every time a request is submitted, the FaaS platform uses a replica of the same resource configuration instance to execute that task, and clients pay per request serviced. So, in the case where we have heterogeneous traffic with both small (low compute requirements) and large (high compute-intensive) tasks, the following arrangements apply with current FaaS solutions:

- a) clients may ensure they have a configuration large enough to service both types of tasks, however this leads to over-provisioning and thus over-paying for smaller tasks;
- b) if a resource configuration is heterogeneous (for example, includes both a CPU and an FPGA), clients need to manually load-balance traffic to distribute task workloads to the appropriate resource, for instance, sending smaller tasks to the CPU and larger tasks to the FPGA;
- c) clients may try to identify the cheapest resource configuration that meets performance requirements for each type of task and deploy separate function services, however this requires expertise.

2.2 Related Work

Table 1 summarises current cloud FaaS approaches. The three key commercial FaaS offerings (AWS Lambda [5], Microsoft Azure Functions [14], and Google Cloud Functions [10]) are limited to CPU-based function types, where users select memory capacity and the number of CPU cores to service each request. Therefore, there is no scheduling required. At the time of this writing, none of the commercial FaaS vendors allow configurations with accelerators.

Open-source frameworks like OpenFaaS [2], OpenWhisk [6] and Kubeless [12] allow a more flexible environment than their commercial counterparts, enabling users to build their own FaaS systems. Developers are able to implement and deploy their own function types and control certain resource management mechanisms, with some support for accelerators (e.g. virtualised GPU nodes). However, although these tools enable greater flexibility, and inclusion of arbitrary instances with accelerators, they are also limited by a single instance type.

Our approach, SLATE, builds on the mechanisms of traditional FaaS approaches with added support for heterogeneous scheduling. Individual requests are mapped automatically for execution onto the most effective instance type from a pool of candidates derived offline using performance modelling. SLATE is loosely based on the heterogeneous PaaS system ORIAN [16], modified to employ FaaS execution and cost models. SLATE's bespoke FaaS approach is able to fully harness the benefits of powerful heterogeneous platforms with a mix of CPU and accelerator resources.

2.3 Challenges

In general, when considering heterogeneous computation, there is no single resource configuration that works

best for all types of workloads, and the best configuration for each scenario is not obvious. For instance, smaller jobs may perform faster on CPUs since data movement and offload overheads would dominate otherwise, while sufficiently large streaming and data-parallel workloads may perform better on FPGAs and GPUs, respectively. Moreover, data-types and numerical representations may also drastically affect relative performance. For instance, FPGAs tend to excel with integer-based operations, while CPUs and GPUs are designed to work with double-precision operations. Thus, management techniques based solely on replicating a single resource configuration, as currently found in traditional FaaS systems, do poorly to leverage the benefits of heterogeneous computation.

The lack of support for heterogeneity in cloud computing in general can be attributed to the complexity of scheduling heterogeneous resources at runtime. In particular, it would be beneficial to be able to map a request to a device that is best suited to service it. With new accelerators appearing in the market every year, the management strategy needs to be flexible and generic to support legacy and new devices. Knowledge about the suitability of each resource to different workloads is necessary, but acquiring and maintaining such knowledge is challenging, particularly as platforms grow.

SLATE addresses these challenges and supports the following key novel features:

1. Heterogeneous scheduling to map each individual request onto the most suitable device selected from a pool of candidate instance types;
2. Offline performance modelling to characterise function performance on supported heterogeneous targets in order to inform scheduling decisions at runtime;
3. Seamless and transparent accelerator support, enabling high-level applications that invoke SLATE functions to be entirely resource-oblivious.

Table 1 Comparison between different FaaS approaches.

Product	Type	Scheduling	Resource Configuration	Accelerator Support
AWS Lambda [5]	Commercial	None	User selected	No
Google Functions [10]	Commercial	None	User selected	No
Azure Functions [14]	Commercial	None	User selected	No
OpenFaaS [2]	Open Source	None	User selected	Yes
OpenWhisk [6]	Open Source	None	User selected	Yes
Kubeless [12]	Open Source	None	User selected	Yes
SLATE	Open Source	Heterogeneous	Automatically selected	Yes

3 The SLATE Approach

3.1 Definitions

To explain the details of our approach, we first present the definitions used throughout the remainder of this paper.

A cloud *function* is a computation available for execution by the FaaS system. A function *request* defines a task that is submitted to the FaaS system to be executed. For instance, the *matmul*(A, B) request triggers a matrix multiplication task in SLATE where A and B are $N \times N$ matrices.

Requests are resource-oblivious, which means that they do not specify which compute resources to employ for task execution. Each request is serviced by a *function instance*, which is a set of resources automatically allocated by the FaaS system to execute the corresponding task. Each instance has an associated *function type*, (N, PE, f, D) , where N and PE specify the resource configuration (quantity and type of processing element), f specifies the cloud function, and D specifies the input domain on which the instance operates.

For simplicity, our current SLATE model is limited to instances which combine one or more processing elements ($N > 1$) of the same type to acquire more computational power (e.g. 3 FPGAs). However, our model can be extended to support other instance types, including instances that mix different types of processing elements.

3.2 Pricing

The pricing model of SLATE, which is based on existing FaaS systems, consists of two costs charged to the client:

1. a *request cost*, which is a fixed rate per request submission, and
2. an *execution cost*, which depends on the duration and resources used (e.g. memory and CPU) to execute a task.

The key idea behind the *request cost* is to charge clients based on the minimum set of resources that the system guarantees to be available at all times. To compute the *execution cost*, we simply multiply the cost of the instance's resources with the task execution duration.

The actual pricing model employed in FaaS, as well as in PaaS and IaaS counterparts, is determined by the cloud provider. This may dynamically change due to supply and demand considerations, as well as each compute resource's operating costs, including energy consumption and maintenance. In addition, cloud providers may offer discount prices at off-peak hours to avoid idle resources. Cloud computing

pricing is complex and is out of the scope of this paper. In our evaluation in Section 7, we consider the standard pricing of a popular FaaS vendor at the time of writing for comparison.

3.3 User-Defined Functions

In this paper, we focus on the case where functions are pre-defined in the FaaS system. An additional mechanism is required for clients to deploy user-defined functions, which involves supplying:

- a function specification that describes: (1) its domain (valid inputs), and (2) all valid resource configurations
- a containerised implementation of the function for each valid configuration (e.g. using one or more Docker containers to realise a micro-service architecture)
- a SLATE API implementation to support function execution

Given the above three items, the SLATE system can schedule and execute a user-defined function on the most appropriate resource configuration. As we shall see next, performance modelling is automatically handled by our system.

3.4 Stages

SLATE comprises three stages, as illustrated in Figure 2, namely:

- I. **Performance modelling:** This stage is performed offline before SLATE is ready to service requests. The aim of performance modelling is to enable a reasonable estimate of the time to execute function $f(x)$ on a particular resource configuration (say a 2-core CPU) for a specific problem size x . Since we cannot profile every possible problem size, we perform a statistical analysis to find a model that best fits observed data. Once SLATE has the performance models for all cloud functions, it is ready for configuration.
- II. **Configuration:** Before submitting requests, clients must configure their FaaS environment. In particular, clients must list all the functions that they wish to execute, and how fast each function should run (performance requirement). SLATE will then automatically identify, based on the performance models, the most cost efficient resource configurations (candidate function types) that meet the timing constraints for each function.

Figure 2 The three stages of the SLATE FaaS System.

III. Execution: Once the configuration stage is complete, the FaaS system is ready to accept function requests. For each request, SLATE automatically maps the task to a function instance using the candidate types determined during configuration. A new instance is spawned if none are currently available. Once task execution is complete, the instances involved become idle and can either be deployed to handle other incoming tasks, or be released to allow other clients to allocate these resources.

In the following three sections, we cover each of these stages in more detail.

4 Performance Modelling

To service each incoming request, the SLATE FaaS system must decide which function type is best suited to execute a particular task. That is, it must meet the performance requirements defined in the configuration stage and also be cost efficient. In order to minimise the decision-making overhead, we generate models that characterise the performance of every function exposed by our FaaS system prior to runtime execution. More specifically, we generate a performance model for every function running on a particular resource configuration. For instance, a matrix multiplication can have

three possible targets: 1 CPU, 12 CPUs, and 1 GPU; each would each have their own performance model. Our performance modelling approach includes two distinct steps: profiling and model generation, which we explain next.

4.1 Profiling

One key design feature of our profiling approach is to treat target functions as black-boxes. This allows our performance modelling process to be automated and generic, and thus it can be seamlessly employed whenever a new function implementation is introduced. With our method, each sample profile is identified by three elements: the associated function, the target configuration, and the problem size. For instance, the associated function can be a matrix multiplication, the target configuration can be 12 CPU threads, and the problem size can be $10^6 \times 10^6$ matrices. To collect enough profiles to derive an accurate model and to speed up this process, a ‘*smart*’ profiling method was developed. The method is based on the following two assumptions:

1. **Saturation.** The observed function throughput (amount of work done per unit of time) will eventually saturate (stop changing), whether trends increase or decrease to saturation (see Figure 6 for examples of increasing and decreasing saturating models)

2. **Domain.** The valid function domain is known (i.e. minimum, maximum, and valid granularity of supported input problem sizes) as well as all valid resource configurations (see requirements in Section 3.3)

Based on these assumptions, our smart profiling method collects samples by starting at the minimum problem size, maintaining an ordered list of appropriate problem sizes until reaching the maximum. Increments between sampled problem sizes in the list are increased as the change in the throughput between subsequent samples decreases. This is analogous to a negative second derivative of a continuous function. Fewer profiles are collected as problem sizes approach saturation and throughput values change less. For each configuration and problem size, a minimum of three samples are collected. Figure 5(a) shows an example of collected profiles.

4.2 Model Generation

Once profiles have been collected, our approach derives models to predict throughput for each function implementation. To do so, sampled profiles are *cleaned* to remove outliers, then regression techniques are used to derive mathematical functions that accurately model the samples. As with profiling, we employ a generic technique to generate models for arbitrary functions, treating implementations as black boxes. This approach enables model generation for implementations without requiring source code access, whereas many other performance modelling techniques require such access to extract application features [19] [8] [18] [9].

Sample Cleaning. First, samples with the same function, target configuration, and problem size are grouped and

averaged, such that there is one throughput value for each problem size. The first cleaning step is to remove outlier samples within each group. The throughput average and standard deviation for each group are calculated. If the standard deviation is larger than 10% of the average, the sample with the greatest average distance (absolute difference) to all other samples in the group is removed. The average and standard deviation are re-calculated and this process is repeated until the standard deviation is smaller than 10% of the average. If only one sample remains in a group, the data point is removed.

Next, trend outliers are identified and removed. The series of average throughputs for each function and configuration is traversed starting from the smallest problem size, comparing it to the next. If the throughputs differ by more than 200% while problem sizes differ by less than 10 times, the next throughput sample is removed. Figure 5(a) and (b) show sample profiles before and after cleaning. Note that the 10% and 200% threshold values have been empirically derived, and can be changed.

Function fitting. Once profile samples have been averaged and cleaned, a mathematical function is derived to model the throughput trend for each function implementation. Performance models are piece-wise, with a function for the pre-saturation region and a constant throughput value for the saturation region, as follows:

$$tp(x) = \begin{cases} model_{pre_sat}(x, c_1, c_2, c_3, \dots) & x_{min} \leq x \leq x_{sat} \\ tp_{sat} & x_{sat} < x \leq x_{max} \end{cases}$$

where x is the input problem size, x_{min} and x_{max} are the supported domain limits, x_{sat} is the problem size at which

Figure 3 The SLATE FaaS configuration stage.

saturation occurs. tp_{sat} is the saturated throughput value, and $model_{pre-sat}$ is a mathematical function of x defined by a set of coefficients ($c_1, c_2, c_3, \dots$).

The steps to automatically generate the performance model from accrued profiling data are as follows. First, the constant saturated throughput value (tp_{sat}) is determined. Starting from the largest problem size and moving backwards, samples are iteratively averaged until the throughput changes by more than 5%. This average is tp_{sat} .

Next, least squares regression is used to derive model coefficients ($c_1, c_2, c_3, \dots$) for $model_{pre-sat}$. Any function that models the profile trend shape can be used. To automate this process, our current performance modelling process identifies a suitable pre-saturation model from two function types:

- saturating log: $c_1 \times \log(c_2 + c_3x + c_4x^2 + \dots)$ (see Figure 6(a), (b), (d), (e))
- decaying exponential: $e^{(c_1 + c_2x + c_3x^2 + \dots)}$ (see Figure 6(c), (f))

Each of the above model types is tested, and the one with the lowest fitting error is selected. Various optimisation tools, such as SciPy optimize [15], can be used to determine the coefficients. Finally, the saturation problem size, x_{sat} , is determined as the intersection between the derived pre-saturation model and the constant saturation value.

Once the performance models are generated, they are stored and ready to be used by the configuration stage, as explained in Section 5.

5 Configuration

Before requests can be serviced at runtime, SLATE must be configured. This stage is a novel aspect of our approach, critical to reduce runtime decision-making complexity and overhead. Upon completion of the configuration phase, a bespoke SLATE system is initialised for a given application. This system is tailored to the client's performance requirements, and only considers the most cost effective function instance types. Figure 3 depicts the key configuration steps of SLATE:

- (1) **client submits the application manifest.** The application manifest includes a list of functions in the application and the client's requirements. For each function, the client must specify the valid domain and performance objective (i.e. a maximum execution time target for that function with any input size in the domain). An example is included in box (1) of Figure 3.
- (2) **determine candidate function types.** The list of candidate function types contains all types considered by the scheduler during execution. Determining this list is a crucial aspect of SLATE, since it prunes the

search space by restricting each tailored FaaS system to a set of relevant candidate types. This removes significant decision-making overhead at execution time. Each function may have multiple implementations with different resource targets N and PE (denoting, respectively, resource quantity and type of processing element), and thus different possible instance types. As explained in Section 4, performance models for each implementation are derived offline, and these are used to predict execution time while determining candidate types. To identify candidates, we generate a graph for each function and corresponding performance objective in the manifest, plotting predicted execution times for all implementations and various inputs in the specified domain. A range of inputs spanning the function domain submitted in the application manifest are considered. See the example in Figure 4. The *best* target resource, (N, PE) , for each input is determined, such that it meets its performance objective (time to complete the task) with the minimum execution cost. Candidate types are identified for each function based on these 'best' configurations for each range of inputs: $(N, PE, f, (min, max))$. To ensure our candidate types and corresponding input ranges enable effective decisions at runtime, the segmentation process is iterative. After a first segmentation pass (Figure 4), we repeat the process for each pair of neighbouring subdomains for a more fine-grained segmentation. This improves our robustness against selecting unrepresentative samples in the first pass by ensuring barriers between sub-domains are not arbitrary. In practice this two pass system is found to be effective, but it can be extended to perform more iterative passes to further improve robustness.

Figure 4 During the configuration phase, clients can establish the timing objective for each function. SLATE automatically segments the input domain and identifies the candidate instance types to be used during execution.

- (3) **determine minimum instance group and cost per request.** By default, the minimum instance group contains one instance of each candidate function type. This way, there will be at least one instance of each candidate type readily available, avoiding the overhead of spawning instances from zero. Note that clients can increase the number of instances in this group for any candidate type. For instance, Figure 3 illustrates a minimum instance group defined by the client, where each of the five candidate function types has a pre-allocated number of instances (1, 2, 1, 3 and 1, respectively). The minimum instance group defines the request cost, which is a fixed cost added to the total cost of executing a task. The request cost is proportional to the size of the minimum instance group, and clients have the option to accept this cost, before proceeding to the next step. Clients may update the performance objectives (going back to step 2) to select more cost efficient candidate types.
- (4) **initialise SLATE FaaS system.** Once the client accepts the minimum group cost, a bespoke SLATE FaaS system is initialised: the minimum group function instances are deployed and a *scheduler* is initialised with access to the instance group and the candidate type list.
 - The *instance group* contains all the allocated instances. Instances in the group are either idle and can be immediately employed by the task scheduler, or are busy executing a task.
 - The *scheduler* is responsible for mapping each request forwarded from the gateway to a suitable instance in the group. An instance of type (N, PE, f, D) is suitable to execute a request $f(x)$ if $x \in D$. For example, a function instance with type $(2, GPU, matmul, (1000, 100000))$ can execute a matrix multiplication function using two GPUs, accepting $N \times N$ input matrices with $1000 \leq N \leq 100000$. To execute a task, the scheduler selects a suitable instance from the group, forwards the request to that instance for execution, and marks the instance as *busy* for the duration of execution. If there are no available (i.e. not *busy*) suitable instances in the group, the scheduler immediately spawns a new one. The scheduler also monitors and maintains a log of the time and instance selected for every request. This log is checked periodically to determine each instance's *idle time*, i.e. the time since the last request for that instance type. If an instance's idle time is greater than the *idle time threshold* (default 10s), and the instance is not currently busy, it is removed from the group. While releasing instances from the group has no bearing on the cost for the client, it allows other clients to allocate these resources.

6 Execution

Once the configuration stage is complete, SLATE is ready to accept incoming requests. As illustrated in Figure 2, the key components of SLATE during execution are:

- The *gateway* serves as a single point of entry to the underlying FaaS resource management platform. To execute functions, function requests are submitted to the *gateway*. The requests are forwarded to the *scheduler* for execution, monitoring, and scaling.

7 Evaluation

In this section, we evaluate our approach using our SLATE FaaS simulator, covering performance modelling and run-time mapping decisions.

Case-Studies. For our evaluation, we target three case-study functions: (1) AdPredictor [11], an advertisement click prediction model (machine learning); (2) Exact Align [7], a sequence alignment process (bioinformatics); and (3)

Figure 5 Profiling, Cleaning, and Performance Modelling For Exact Align DFE Implementations.

N-body Simulation [13], particle simulation (physics). These are examples of HPC applications that are not well-supported by current managed cloud platforms (PaaS and FaaS).

Platform. We have optimised multi-CPU and multi-FPGA implementations, targeting a Intel i780 CPU platform with 12 cores and 24 Max4 Dataflow Engines (DFEs) [1] with Intel Stratix V FPGAs. A DFE is a complete compute device system developed by Maxeler [3], which contains an FPGA as the computation fabric, RAM for bulk storage, logic to connect the device to a CPU host, and all necessary interfaces, interconnects, and circuitry. Our CPU implementations are programmed in C++, while the DFE implementations are written in MaxJ, a domain-specific language based on Java for developing dataflow programs.

Pricing. We use the following pricing model for our evaluation: 1 CPU-s costs \$0.00002 and 1 DFE-s costs \$0.00008. 1 CPU-s corresponds to a one second execution on a CPU, while 1 DFE-s corresponds to a one second execution on a DFE. Each request costs $$(1\% * \text{min_group_cost})$.

So, in a scenario with 1 million requests, each executed on (2, DFE) targets for 100ms, and a minimum group containing one (2, DFE) instance and one (1, CPU) instance, the cost would be:

- Request cost: $10^6 \times 1\% \times (2 \times \$0.00008 + 1 \times \$0.00002) = \1.80
- Execution cost: $10^6 \times 0.1s \times (2 \times \$0.00008) = \$16.00$

- Total cost: \$17.80

Our prices are roughly based on the FaaS pricing for AWS Lambda [5], where each request costs \$0.0000003, and the average execution cost is \$0.00002 per second for one CPU instance with 1 GB of RAM. Our DFE cost is based on AWS EC2 FPGA-optimised instances compared to general-purpose CPU instances, where a `f1.2xlarge` instance costs roughly 4 times more than an `m4.2xlarge` instance [4].

Performance Models. Using the techniques explained in Section 4, performance models are derived for each case-study for CPU and DFE targets. Graphs of the derived performance models are included in Figures 5 and 6. In order to evaluate the accuracy of the models, observed compute times for each problem size and configuration were compared to model-predicted times, and the average percent errors for each were recorded for each target implementation. In general, the average errors are reasonably small, mostly less than 1% and with a maximum of 7.94% for Exact Align executed on 4 DFEs. This can be attributed to high variance due to a lack of determinism in the observed compute times for the 4 DFE implementation, but remains small enough to make sufficiently accurate predictions.

Although the average errors are observed to be very small, it was noticed that errors varied greatly between different problem size ranges. For each implementation, sample problem sizes were split into four quartiles, and the

Figure 6 Performance Models.

Figure 7 Maximum error per data set quartile for (1,DFE) and (2,CPU) performance models.

maximum error for each problem size quartile was recorded. Examples of these quartile maximums for all (1, DFE) and (2, CPU) implementations are included in Figure 7, where a negative value indicates the model *overestimates compute time*. In general, errors were observed to be smallest in saturation regions (Q4) and largest for the smallest pre-saturation problem sizes (Q1). For each of our simulation experiments, an upper bound on predicted execution time is considered by assuming the maximum error for the quartile in which the specified problem size resides.

Configuration. To validate our heterogeneous FaaS approach, we compare SLATE heterogeneous function groups to homogeneous function groups in terms of performance and cost. Homogeneous function groups represent existing state-of-the-practice (SOP) FaaS approaches, which map all requests to an instance of the same type. We implement our own homogeneous function groups for both CPU and DFE targets, since the current SOP does not target DFE instances, and comparing heterogeneous SLATE functions to SOP CPU functions would not be fair for computations suited to FPGAs.

Before we run our experiments, we configure a SLATE system for each case-study application (Section 5). Using

Table 2 Heterogeneous and Homogeneous Function Groups.

	Functions	Candidate Types	Cost of 10^6 request
SLATE	Exact Align (Obj=5s)	(1, CPU, align, s)	\$2.60
		(1, DFE, align, m)	
		(2, DFE, align, l)	
	AdPredictor (Obj=200ms)	(1, CPU, adp, s)	\$1.00
		(1, DFE, adp, l)	
Homog. CPU	Exact Align	(1, CPU, align, {s, m, l})	\$0.20
	AdPredictor	(1, CPU, adp, {s, l})	\$0.20
Homog. DFE	Exact Align	(2, DFE, align, {s, m, l})	\$1.60
	AdPredictor	(1, DFE, adp, {s, l})	\$0.80
	N-Body Sim.	(1, DFE, nbody, {s, l})	\$0.80

our performance models, we generate the graphs in Figure 8 to identify candidate types for each function's input domain according to performance requirements (see Table 2).

As explained in Section 5, SLATE automatically segments each function's domain to classify inputs corresponding to the instance type they are suited to. For instance, with an objective of 5s for every Exact Align task, SLATE identifies three sub-domains (task types) and the function instance types that suit them, namely: *s* (small) tasks are suited to (1, CPU, align, s) functions, *m* (medium) tasks are suited to (1, DFE, align, m) functions, and *l* (large) tasks are suited to (2, DFE, align, l) functions.

Employing the candidate function types identified, we run simulation experiments using the function groups outlined in Table 2. For each case-study, we consider:

- A heterogeneous SLATE function group: with heterogeneous candidate types as determined in Figure 8.
- A homogeneous CPU function group: suited to *s* traffic.
- A homogeneous DFE function group: suited to *l* traffic.

Figure 8 Determining candidate function types for each case-study: Exact Align ((1, CPU, align, s), (1, DFE, align, m), (2, DFE, align, l)), AdPredictor ((1, CPU, adp, s), (1, DFE, adp, l)), and N-body Simulation ((1, DFE, nbody, {s, l})).

Table 3 Speedup And Execution Cost Decrease of SLATE Compared to Homogeneous Functions For Different Tasks.

Task	type	SLATE Speedup (Cost Improvement)	
		Homog. CPU	Homog. DFE
align(1500)	<i>s</i>	1.0× (1.0×)	1.3× (10.4×)
align(2000)	<i>s</i>	1.0× (1.0×)	1.0× (7.8×)
align(15000000)	<i>l</i>	76.3× (9.5×)	1.0× (1.0×)
align(20000000)	<i>l</i>	76.3× (9.5×)	1.0× (1.0×)
adp(5000)	<i>s</i>	1.0× (1.0×)	3.0× (11.8×)
adp(10000)	<i>s</i>	1.0× (1.0×)	1.5× (5.9×)
adp(500000)	<i>l</i>	31.5× (7.9×)	1.0× (1.0×)
adp(1000000)	<i>l</i>	37.8× (9.5×)	1.0× (1.0×)
nbody(4096)	<i>s</i>	-	1.0× (1.0×)
nbody(8192)	<i>s</i>	-	1.0× (1.0×)
nbody(32768)	<i>l</i>	-	1.0× (1.0×)
nbody(65536)	<i>l</i>	-	1.0× (1.0×)

For N-Body Simulation, since there is one function type which is best for all workloads, $(1, DFE, nbody, \{s, l\})$, we do not consider a homogeneous CPU function type for our experiments.

7.1 Performance Evaluation

To evaluate the performance of SLATE heterogeneous functions, we compare the execution time for an individual task using a SLATE-selected function instance to each homogeneous function instance in Table 2. The SLATE times take into consideration the overhead of the scheduler selecting an instance type (observed to be on the order of $1\mu s$). This overhead is practically negligible due to the configuration stage, which allows the system to perform one-to-one mapping decisions at runtime.

The speedup of execution using SLATE compared to employing homogeneous instances is shown in Table 3.

The corresponding improvements in cost are also recorded. For task types to which the homogeneous function instances are suited, SLATE achieves the same execution time and cost (i.e. 1.0 times speedup and cost decrease). That is, for *s* AdPredictor and Exact Align tasks executed on homogeneous CPU instances, *l* AdPredictor and Exact Align tasks executed on homogeneous DFE instances, and all N-Body Simulation tasks executed on homogeneous DFE instances. This is because SLATE selects the instance type best-suited to each task which is the same as the homogeneous instance type in these cases.

On the other hand, for task types to which the homogeneous instances are not suited, there is a difference in execution time and SLATE is more cost effective. That is, for *s* AdPredictor and Exact Align tasks executed on homogeneous DFE instances, and *l* AdPredictor and Exact Align tasks executed on homogeneous CPU instances. In these cases, whether the execution time is greater or less than the SLATE-selected instance, execution is more costly. For instance, for *align*(2000), SLATE does not improve speed, but achieves a 7.8 times cost decrease.

In general, since the SLATE-selected instance is guaranteed to meet a timing objective for each task, it performs sufficiently well and is more cost effective overall.

Note that the execution times used for our experiments in this paper differ slightly from similar experiments in our previous work [17]. This is due to more rigorous data cleaning to remove outliers before averaging results, which particularly affect the more non-deterministic DFE implementations of each application. The trends in our results still support the benefits of our approach.

7.2 Cost Efficiency Evaluation

To evaluate the cost efficiency of SLATE, we compare the costs of executing sequences of 1 million tasks using SLATE functions to each homogeneous function group in Table 2, where the fixed cost for 1 million requests is included in the last column.

Figure 9 Examples of uniform, random, and spiked task sequences.

As previously mentioned, FaaS pricing models include an execution cost, based on the duration of the task, as well as a fixed cost per request. Since our approach automatically selects function instances that are the most cost effective for each task, the improvements in execution cost are implicit (Section 7.1). However, using our pricing model, heterogeneous function groups with multiple candidate workers typically have higher fixed request costs than homogeneous groups. Therefore, to fairly compare the cost efficiency of SLATE to homogeneous functions groups, we consider the total cost of executing sequences of multiple tasks.

For each function, we consider s or l task types and sequences with 1 million tasks. We evaluate SLATE's cost efficiency with three different types of task sequence: *uniform* traffic (1 million tasks of the same type), *random* traffic (a random sequence with 1 million tasks of either type), and *spiked* traffic (mostly one type with a spike of 100,000 of the other type). Examples of these traffic types are depicted in Figure 9 for NBody Simulation. The decrease in cost achieved by SLATE compared to each homogeneous function group is included in Table 4, where a value < 1 indicates a cost increase.

Since NBody simulation has a single resource type (1 DFE) suited to all traffic, it performs equally to SLATE in performance and cost in all scenarios.

In the case of the other applications, for sequences with uniform tasks, the homogeneous groups with resources to which that task type is suited are equally or more cost effective than SLATE. For example, uniform s AdPredictor and Exact Align sequences executed on homogeneous CPU instances are 5 times and 2 times less expensive than SLATE

respectively, while uniform l AdPredictor and Exact Align sequences are equal in cost to SLATE. In the cases where there is homogeneous s traffic, the significant reduction in fixed costs by using homogeneous instance groups leads to a reduction in overall cost of the sequence.

For task sequences with heterogeneous traffic (random or spiked), the comparisons of AdPredictor and Align are different with respect to SLATE. With AdPredictor, SLATE is more or equally cost effective than the homogeneous groups in all cases. SLATE costs up to 7.8 times less than homogeneous CPU functions for AdPredictor traffic with a spike of s tasks, and up to 2.8 times less than homogeneous DFE functions for AdPredictor traffic with a spike of l jobs. For non-uniform Exact Align sequences, SLATE is always more cost effective than the homogeneous CPU group (up to 9.5 times less costly for traffic with a spike of s tasks), but it is equal in cost to the homogeneous DFE group. This is because the difference in execution time and therefore cost of s and l Exact Align jobs is so large that the l jobs with higher execution time dominate the overall cost whether there are CPU resources available for the s jobs or not.

7.3 Discussion

Based on our evaluation, we expect that in scenarios with heterogeneous traffic comprising tasks that have different computational requirements, SLATE is likely to provide cost and performance benefits over homogeneous FaaS. This is demonstrated by our results with AdPredictor. However, in cases where there is predictable uniform traffic, it is better to use homogeneous functions with a resource configuration tuned to all traffic. For example, with N-Body Simulation, there is no benefit of using heterogeneous SLATE functions over using homogeneous DFE functions. Furthermore, in cases with heterogeneous task types for which one type significantly dominates in terms of execution time and cost over the other(s), SLATE may not provide cost benefits compared to a homogeneous resource group which is suited to the dominant traffic type (for example, Exact Align). In this case, it should be noted that while SLATE is not detrimental or advantageous in terms of cost, for clients without knowledge of the resource types best-suited to their traffic, automatic candidate identification may still be beneficial.

In a scenario with heterogeneous traffic, an expert client may manually determine function types best suited to each task type, and deploy separate homogeneous function groups for each type of task. While this might avoid increased fixed costs of heterogeneous SLATE groups, it requires significant effort and expertise to segment traffic into types and tailor instances to each. On the other hand, non-expert clients are unlikely to be able to tune instance types to each task type. Therefore, automatic identification of suitable candidate function types and segmentation of function domains accordingly

Table 4 Cost Improvement of SLATE Functions Compared to Homogeneous Functions For Different Task Sequences.

Function	Traffic Type	SLATE Cost Improvement	
		Homog. CPU	Homog. DFE
Exact Align	uniform s	0.2×	1.7×
	uniform l	9.5×	1.0×
	random	9.2×	1.0×
	spike of s	9.5×	1.0×
	spike of l	8.9×	1.0×
AdPredictor	uniform s	0.5×	3.8×
	uniform l	7.9×	1.0×
	random	6.8×	1.4×
	spike of s	7.8×	1.0×
	spike of l	3.3×	2.8×
N-Body Sim.	uniform s	-	1.0×
	uniform l	-	1.0×
	random	-	1.0×
	spike of s	-	1.0×
	spike of l	-	1.0×

using SLATE is beneficial to both experts, by saving effort, and non-experts, by requiring less prior knowledge.

Finally, our simulation calculations do not currently take into account the overhead of initialisation and spawning new function instances (including dynamic reconfiguration), however we applied the same assumption to both heterogeneous and homogeneous groups in our evaluation. We intend to study the mechanisms for reducing this spawning overhead, for instance, by pre-allocating instances according to traffic patterns, in future work.

8 Conclusion

This paper proposes SLATE, a fully-managed Function-as-a-Service (FaaS) system for deploying managed cloud functions onto heterogeneous cloud infrastructures. SLATE extends the traditional, homogeneous FaaS execution model to support heterogeneous function types with different target resources, while abstracting and automating all resource management. In doing so, we aim to improve the accessibility of specialised accelerator resources to cloud tenants. We validate our SLATE approach with simulation, considering case study functions in three application domains (machine learning, bio-informatics, and physics), with implementations targeting FPGA and CPU resources. We compare SLATE heterogeneous functions to homogeneous CPU and FPGA function groups, achieving, respectively, a cost improvement for non-uniform task traffic of up to 8.9 and 2.8 times while maintaining user-supplied execution time objectives.

Current and future work includes developing a full SLATE prototype, and targeting other application domains and accelerator types, such as GPUs and application-specific devices.

Acknowledgements The support of Intel and the U.K. EPSRC (grants EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1 and EP/L00058X/1) is gratefully acknowledged.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Maxeler MPC-X Series. accessed Apr-2020. [Online]. Available: <https://www.maxeler.com/products/mpc-xseries/>
2. OpenFaaS Introduction: Serverless Functions Made Simple. accessed Apr-2020. [Online]. Available: <https://docs.openfaas.com/>
3. Maxeler Technologies. accessed Apr-2020. [Online]. Available: <https://www.maxeler.com/>
4. Amazon Web Services. Amazon EC2. accessed Apr-2020. [Online]. Available: <https://aws.amazon.com/ec2/>
5. Amazon Web Services. AWS Lambda: Serverless Compute. Accessed Apr-2020. [Online]. Available: <https://aws.amazon.com/lambda/>
6. Apache Software Foundation. Open Source Serverless Cloud Platform. accessed Apr-2020. [Online]. Available: <https://openwhisk.apache.org/>
7. Arram, J., Kaplan, T., Luk, W., & Jiang, P. (2017). Leveraging FPGAs for Accelerating Short Read Alignment. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(3), 668–677.
8. Bleuse, R., Hunold, S., Kedad-Sidhoum, S., Monna, F., Mounié G., & Trystram, D. (2017). Scheduling Independent Moldable Tasks on Multi-Cores with GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 28(9), 2689–2702.
9. Du, P., Sun, Z., Zhang, H., & Ma, H. (2019). Feature-Aware Task Scheduling on CPU-FPGA Heterogeneous Platforms. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 534–541.
10. Google Cloud Platform. Cloud Functions. accessed Apr-2020. [Online]. Available: <https://cloud.google.com/functions>
11. Graepel, T., Candela, J. Q., Borchert, T., & Herbrich, R. (2010). Web-scale Bayesian Click-through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *The 27th International Conference on International Conference on Machine Learning (ICML)*, 13–20.
12. Kubeless. The Kubernetes Native Serverless Framework. accessed Apr-2020. [Online]. Available: <https://kubeless.io/>
13. Maxeler. (2015). N-Body Particle Simulation. <https://github.com/maxeler/NBody>, accessed Jan-2020.
14. Microsoft Azure. Azure Functions: Serverless Compute. accessed Apr-2020. [Online]. Available: <https://azure.microsoft.com/en-gb/services/functions/>
15. SciPy.org. SciPy Optimize. accessed Sept-2020. [Online]. Available: <https://docs.scipy.org/>
16. Vandebron, J., Coutinho, J. G. F., Luk, W., Nurvitadhi, E., & Naik, M. (2019). Enhanced Heterogeneous Cloud: Transparent Acceleration and Elasticity. In *International Conference on Field-Programmable Technology (FPT)*, 162–170.
17. Vandebron, J., Coutinho, J. G. F., Luk, W., Nurvitadhi, E., & Naik, M. (2020). SLATE: Managing Heterogeneous Cloud Functions. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 141–148.
18. Wen, Y., Wang, Z., & O'Boyle, M. F. P. (2014). Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, 1–10.
19. Yasudo, R., Coutinho, J., Varbanescu, A., Luk, W., Amano, H., & Becker, T. (2018). Performance Estimation for Exascale Reconfigurable Dataflow Platforms. In *2018 International Conference on Field-Programmable Technology (FPT)*.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.