Enhanced Heterogeneous Cloud: Transparent Acceleration and Elasticity

Jessica Vandebon*, José G. F. Coutinho*, Wayne Luk*, Eriko Nurvitadhi† and Mishali Naik†
*Imperial College London, United Kingdom
{jessica.vandebon17, gabriel.figueiredo, w.luk}@imperial.ac.uk
†Intel Corporation, San Jose, USA
{eriko.nurvitadhi, mishali.naik}@intel.com

***Abstract*—This paper presents ORIAN, a fully-managed Platform-as-a-Service (PaaS) for deploying high-level applications onto large-scale heterogeneous cloud infrastructures. We aim to make specialised, accelerator resources in the cloud accessible to software developers by extending the traditional homogeneous PaaS execution model to support automatic runtime management of heterogeneous compute resources such as CPUs and FPGAs. In particular, we focus on two mechanisms: *transparent acceleration*, which automatically maps jobs to the most suitable resource configuration, and *heterogeneous elasticity*, which performs automatic vertical (type) and horizontal (quantity) scaling of provisioned resources to guarantee QoS (Quality of Service) objectives while minimising cost. We develop a prototype to validate our approach, targeting a hardware platform with combined computational capacity of 28 FPGAs and 36 CPU cores, and evaluate it using case studies in three application domains: machine learning, bioinformatics, and physics. Our transparent acceleration decisions achieve on average 96% of the maximum manually identified static configuration throughput for large workloads, while removing the burden of determining configuration from the user; an elastic ORIAN resource group provides a 2.3 times cost reduction compared to an over-provisioned group for non-uniform, peaked job sequences while guaranteeing QoS objectives; and our malleable architecture extends to support a new, more suitable resource type, automatically reducing the cost by half while maintaining throughput, and achieving a 23% throughput increase while fulfilling resource constraints.**

***Index Terms*—FPGA, PaaS, heterogeneous clouds, transparent acceleration, heterogeneous elasticity**

I. Introduction

In the last decade, cloud computing has shaped the information technology landscape, with increasing numbers of businesses, governments, and researchers offloading their computation needs to third-party data centres to drastically reduce their operating costs. In order to serve computationally intensive workloads, including High Performance Computing (HPC) and Artificial Intelligence (AI), cloud providers offer specialised accelerator resources, such as GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays), to tenants. However, this embrace of heterogeneity is mostly found in cloud IaaS (Infrastructure-as-a-Service) systems, such as Amazon EC2 [1], where tenants can request virtual machines (VMs) with access to hardware accelerators. In this context, tenants are still responsible for manually managing their provisioned resources.

To reduce the above effort, Platform-as-a-Service (PaaS) systems automatically manage cloud resources. PaaS offers a simplified cloud infrastructure model in which a pool of computation units, acting as autonomous *workers* (e.g. Dynos in Heroku [2] or EC2 instances in Amazon Elastic Beanstalk [3]), execute tasks from a stream of incoming requests. Within this model, tenants are still responsible for identifying the size (capacity) of the worker that can meet their performance and cost requirements. PaaS offers *transparent execution*, which ensures that tasks are automatically balanced across provisioned workers, and *elasticity*, which automatically adjusts the number of workers to match incoming traffic in order to satisfy performance objectives while avoiding over-provisioning.

While this PaaS execution model is well suited for CPU-based resources, it translates poorly to heterogeneous platforms that support hardware accelerators. This is because the PaaS model assumes that all incoming tasks can be satisfied by a single type of worker. With traditional PaaS, if a specific task type is more demanding, we simply increase the worker's capacity (e.g. amount of memory and/or number of CPU cores). However, in a heterogeneous compute environment, we observe that each type of compute resource works best for specific types of workloads, and this mapping is not obvious. For instance, smaller workloads 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. Data-types and numerical representations may also drastically affect their relative performance, for instance, FPGAs tend to excel with integer-based operations, while CPUs and GPUs are designed to work with double-precision operations. The key challenge is finding a general process that allows these runtime mapping decisions to be made effectively and transparently with the smallest overhead possible.

In this paper, we present an *enhanced* PaaS execution model designed to optimise applications for large-scale heterogeneous cloud platforms, while making them accessible to software developers with no expertise in specialised hardware accelerators. For this purpose, we focus specifically on the *runtime* mechanisms that manage heterogeneous compute resources automatically and support transparent heterogeneous execution and elasticity, which we consider to be the current bottleneck in leveraging hardware accelerators in the cloud. In this context,

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT.2019.00027

cloud applications are linked with our runtime system which manages libraries of optimised implementations targeting CPU and FPGA devices. Future work will include the management of arbitrary user code by introducing compile-time development tools, and supporting more accelerator types such as GPUs.

We believe that our work is novel in two ways. First, we extend traditional PaaS mechanisms to support hardware accelerators. In particular, we employ *multiple* worker types, as opposed to a single type of worker, to leverage different capacities and capabilities suited to different types of workload. We support *transparent acceleration*, with a *task scheduler* that analyses incoming tasks and maps them to the most profitable available worker, and we develop an *auto-scaler* that implements *heterogeneous elasticity* by adjusting the quantity and type of workers based on accrued historic data and subject to user-supplied rules. As far as we know, our proposed PaaS is unique in that it combines vertical (worker type) and horizontal (worker quantity) auto-scaling by design. Second, we develop an offline method that automatically identifies a fixed set of worker types that provide the best trade-offs in terms of performance and cost, significantly reducing the runtime overhead of the auto-scaler.

The main contributions of this paper are as follows:

1) A resource management architecture for a heterogeneous cloud PaaS system designed to support transparent acceleration and heterogeneous elasticity (Section II);
2) The implementation of a PaaS prototype with the above architecture (Section III);
3) An evaluation of our prototype on a platform with 28 FPGAs and 36 CPU cores targeting three application domains (machine learning, bioinformatics, and physics), showing the benefits of our transparent acceleration and elasticity on a heterogeneous cloud platform (Section IV).

Related work is described in Section V, while Section VI concludes this paper and presents future work.

II. The ORIAN Approach

A. Motivation

The traditional PaaS execution model targets mostly homogeneous, CPU-based workers. Our goal is to support new types of workloads in cloud computing platforms, such as HPC and AI, by leveraging heterogeneous compute resources, such as specialised hardware accelerators. We aim to make accelerators accessible to software developers that have no experience working with them. For this purpose, we extend the current PaaS execution model to support transparent acceleration and elasticity (auto-scaling) for heterogeneous resources.

There is no panacea when it comes to a compute device (e.g. CPU, FPGA, or GPU) that outperforms others for all workload types and sizes. This is demonstrated by the example in Fig. 1, showing performance models for a sequencing alignment application [4] executed on different FPGA and CPU configurations. Each configuration corresponds to a number of devices of a specific type, hence (8, FPGA) corresponds to a configuration employing eight FPGAs. In this example, each configuration achieves a different maximum throughput, and the throughput achieved by each varies with problem size. No single configuration achieves the highest throughput for every problem size. For example, while all FPGA configurations achieve a larger maximum throughput than any CPU configuration, all CPU configurations outperform the FPGA configurations until $\sim 10^2$ reads, at which point (2, FPGA) becomes more effective. Furthermore, although the (8, FPGA) configuration achieves the greatest maximum throughput, it is not until $\sim 10^6$ reads that it becomes more effective than the (4, FPGA) configuration. When other factors, such as cost, are also considered, decisions become more complex since the most performant configuration may not be the most cost effective. This lack of a universal 'best' configuration motivates the need for an automated process that maps tasks to the most suitable configuration at runtime.

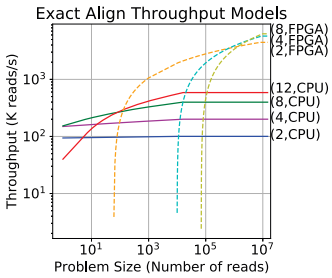


Fig. 1: Performance models for a sequencing alignment application executed on different CPU and FPGA resource configurations.

The above explains in part the current lack of support for heterogeneity in PaaS. In particular, the complexity of runtime management increases with heterogeneity, since in addition to balancing task requests horizontally across multiple workers to service as many requests as possible in parallel, it becomes necessary to scale requests vertically according to the device type that is best suited to service it (see Fig. 1). Another issue is that the computing landscape changes very rapidly every year, with new accelerators appearing in the market with more advanced architectures and runtime systems. This requires management logic to be flexible and generic, otherwise it needs to be constantly redesigned to support legacy and new devices. Effective management of heterogeneous resources requires knowledge about the suitability of each resource to different workloads, but acquiring and maintaining such knowledge is challenging, particularly as platforms grow.

In the following, we describe the key components of ORIAN, our proposed heterogeneous PaaS, and explain how we address the above challenges.

B. ORIAN PaaS Architecture

Fig. 2 illustrates the proposed ORIAN PaaS architecture. The architecture has two key components: the *Application Manager* and the *Resource Manager*. The *Application Manager* works on behalf of cloud tenants, providing an interface for submitting applications and requirements (objectives and allocation rules). Once submitted, the *Application Manager* creates an execution environment for running the application (a CPU-based VM). To leverage hardware accelerators, applications explicitly

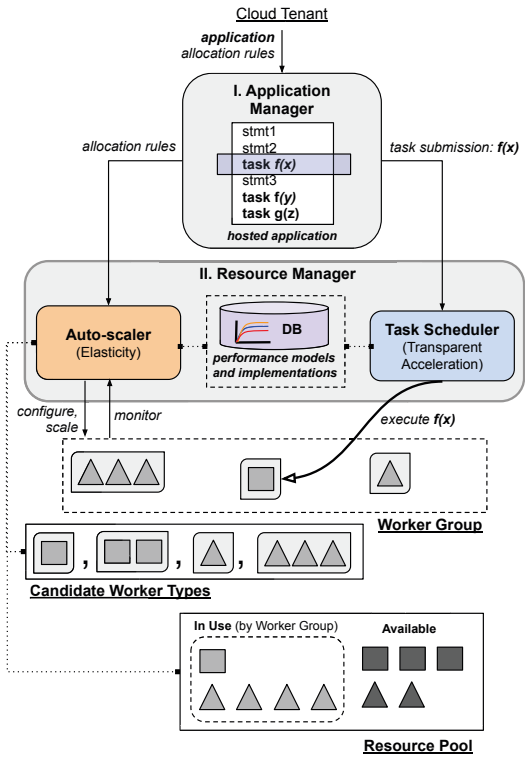


Fig. 2: The ORIAN heterogeneous PaaS architecture.

instruct ORIAN to execute *managed tasks*. Managed tasks (*tasks* for short) are calls to predefined library functions belonging to a specific application domain, abstracting users from implementation and resource management details. To execute a task, users need only to specify the function name and supply the corresponding input data. In turn, the *Application Manager* submits tasks to the *Resource Manager* for execution.

The *Resource Manager* works on behalf of the cloud provider to manage provisioned resources and task execution. It contains two main components: a *Task Scheduler* and an *Auto-scaler*. Both components operate on a heterogeneous *Worker Group*, consisting of currently provisioned worker instances. We define a *worker* as a resource configuration instance comprised by a computing device, such as a CPU or an FPGA, and a specific quantity (i.e. (3, CPU)). In our current definition, we assume that accelerator workers (e.g. (n, FPGA)) use a CPU to drive input and output data to accelerators and back, but the CPU computation is negligible since DMA is used. Our model could be easily extended to more complex workers using CPUs and accelerators in parallel (e.g. ([n,m], [CPU, FPGA])).

A key challenge in our approach is identifying the most profitable worker types within a budget of computing devices. For instance, as illustrated in Fig. 3, with a budget of 3 FPGAs, we can employ three workers of type (1, FPGA), or one (1, FPGA) and one (2, FPGA), or even a single (3, FPGA) to execute any given task. Identifying which type of worker to use and how many of them to employ may have implications in terms of latency (how fast a single request is processed) as

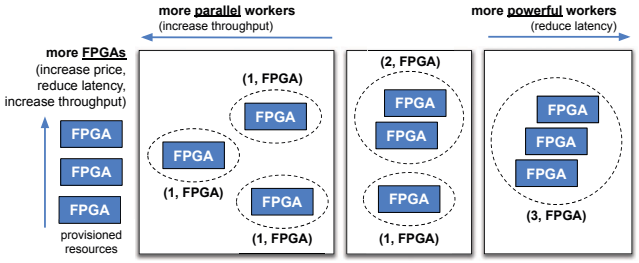


Fig. 3: Identifying the right quantity and type of workers from provisioned resources allows the proposed platform to optimise the application in terms of latency, throughput, and cost.

well as throughput (how fast a group of requests are serviced per unit of time).

Once the worker types have been automatically identified by ORIAN, the *Auto-scaler* is responsible for setting up the initial *Worker Group* with the appropriate quantity and type of workers, and scaling this group if necessary (elasticity) according to the user allocation rules. The *Auto-scaler* is constrained by the *Resource Pool* which defines the budget of computing devices that can be provisioned during the life-cycle of the application. The *Resource Pool* is defined by user-supplied allocation rules, which specify the minimum and maximum number of devices of each type (e.g. CPU or FPGA) and performance (QoS) objectives (see Section II-D). The *Task Scheduler*, on the other hand, is responsible for transparent acceleration, mapping each submitted task to the most suitable available worker in the *Worker Group*.

To guide mapping and scaling decisions, the *Resource Manager* employs a database of task implementations and corresponding models characterising the performance of each implementation for different workloads. In particular, we employ a library-based system where each supported function may have multiple implementations in the database targeting different workers. By maintaining a library of curated, optimised, and reusable accelerator functions, we enable non-device experts to access accelerators without requiring a high-level programming framework. Supporting user-defined accelerator functions and their compilation process is beyond the scope of this paper, but we plan to introduce them in future research.

Note that, as will be explained in Section III and illustrated in Fig. 8, our resource management architecture can be organised as a hierarchy of smaller *Resource Manager* instances, each targeting a specific cloud resource, and governed by the same ORIAN API. Having a uniform interface allows us to build a complex resource management system from simpler ones, and to match the system with the underlying hardware infrastructure. For simplicity, in this section we only consider a single *Resource Manager* instance. More details about our proposed resource management process, including transparent acceleration and heterogeneous elasticity, are described below.

C. Transparent Acceleration

To support transparent acceleration, our platform operates in two modes: offline and online. When offline, implementations are profiled, and performance models are empirically derived

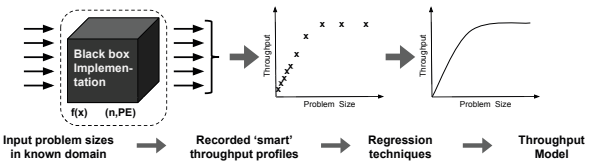


Fig. 4: ORIAN offline performance modelling.

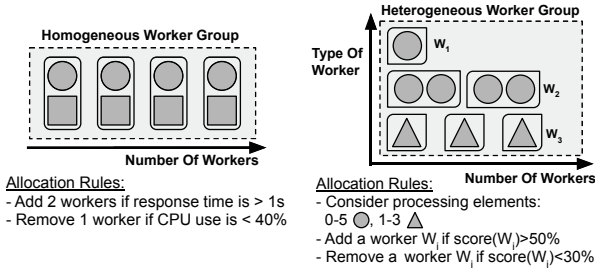
for each. When online, these models are used by the *Task Scheduler* to automatically select the worker in the *Worker Group* expected to minimise execution time and/or cost for a submitted task (*f(x)*). In particular, the *Task Scheduler* selects the worker instance expected to minimise execution time if no performance objective is specified. Otherwise, the *Task Scheduler* selects the cheapest worker that meets the specified objective. For this purpose, a cost model for the available resources needs to be defined by the cloud provider. This model can be based on a number of factors, including power consumption and supply and demand considerations, and can be customised and updated at runtime.

Performance Modelling. Access to performance models is vital for ORIAN to make transparent mapping decisions at runtime. In order to automate the modelling process, we use a generic method to collect samples for an arbitrary, black-box function. The method is based on the following two assumptions: **(1)** the throughput will eventually saturate (stop changing) as we increase problem size, and **(2)** the implementation domain is known (i.e. the minimum, maximum, and granularity of valid problem sizes). The modelling process is illustrated in Fig. 4. Profiles are collected starting at the minimum problem size. Increments between sampled problem sizes are increased as the change in the throughput between subsequent samples decreases (analogous to a negative second derivative of a continuous function). Fewer profiles are collected as problem sizes approach saturation (throughput values change less). This ensures enough profiles are collected to derive an accurate model without performing an exhaustive search. Once profiles have been collected, samples are *cleaned* using common data-processing techniques to remove outliers, and models are derived using least-squares regression.

Dynamic Reconfiguration. While our transparent acceleration approach is designed to support arbitrary accelerator types, dynamic reconfiguration must be taken into account when considering FPGAs. Allocated FPGAs are reconfigured whenever they need to execute a new implementation during the life-cycle of an application. Each time there is a context switch (from implementation X to Y), there is a reconfiguration cost involved. Since we target HPC-based applications, FPGAs are often assigned to perform computationally intensive large tasks, therefore the sub-second reconfiguration cost is negligible compared to the computation time.

D. Heterogeneous Elasticity

Elasticity refers to automatic scaling of allocated compute resources to match incoming workload. By automatically adapting to traffic in order to maintain the *minimum cost* resource set required to satisfy QoS objectives, elastic systems



(a) Traditional PaaS Elasticity (b) ORIAN Heterogeneous Elasticity

Fig. 5: ORIAN vs. Traditional PaaS Elasticity.

aim to reduce tenant costs. In the absence of elasticity, a tenant may provision N workers required to satisfy a QoS objective at peak traffic hours, but they are forced to pay for these N workers at off-peak times when fewer workers would be sufficient. Elasticity is a key PaaS feature to support non-uniform and/or unpredictable traffic without forcing tenants to over-provision. ORIAN's heterogeneous elasticity supports automatic scaling of heterogeneous worker groups.

As depicted in Fig. 5, traditional PaaS platforms scale resources by replicating a single worker type, increasing or reducing compute capacity linearly. Instead, our proposed heterogeneous elasticity system supports multiple worker types, in which we identify the appropriate quantity and type of workers according to incoming workload. To support our heterogeneous *Auto-scaler*, we rely on user-supplied resource-agnostic *allocation rules*:

(1) The minimum and maximum number of processing elements that can be allocated (e.g. 1-3 CPUs, 0-2 FPGAs)
(2) To increase and decrease action thresholds (e.g. add a new worker if score $>$ 50%, remove a worker if score $<$ 30%)
(3) The window size, N (e.g. auto-scaling event every N jobs)
(4) A QoS objective (e.g. complete a job in 1.2s)

Configuration. First, ORIAN determines the *Resource Pool* defining the budget of cloud resources that constrains scaling decisions. Specifically, the minimum and maximum number of each processing element (allocation rule (1)) are used to set the quantity and types of resources in the pool (e.g. 3 CPUs, 2 FPGAs). Next, a list of *Candidate Worker Types*, $\boxed{(n, PE)}$, is derived using the *Resource Pool*, performance models, and QoS objectives (rule (4)). To determine this list, a graph of

Fig. 6: The most effective workers for different problem sizes and a fixed time objective. This example considers 4 workers (W_1, W_2, W_3, W_4), and concludes that 3 of them should be candidate workers (W_1, W_2, W_3). Note that the 'best' worker for each size is that with the minimum cost $\times$ compute time product. Although W_4 is always the most performant (fastest compute time), it is never the most cost effective for the given objective, and thus it is not considered.

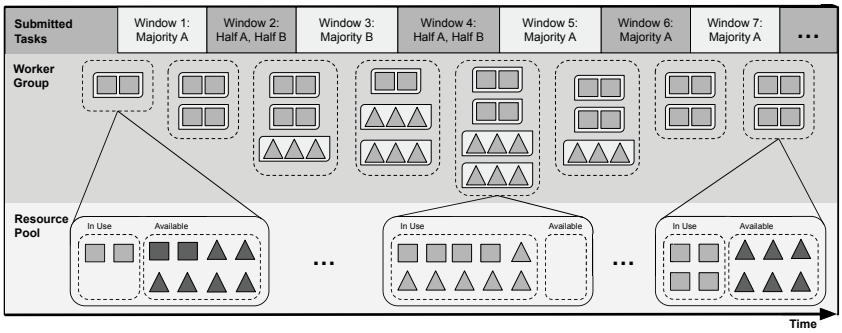


Fig. 7: An example of a dynamically changing worker group and resource pool, described in Section II-D.

predicted execution times for all possible worker types that can be selected from the *Resource Pool* is produced to identify the most effective worker types (see Fig. 6). The resulting list of worker types is therefore designed to cover all problem sizes for a particular managed task and *Resource Pool*, while meeting the given objective with the minimum cost.

Worker Group Scaling. At runtime, while the *Task Scheduler* selects the best worker instance from the *Worker Group* to execute a task, the *Auto-scaler* determines the best candidate worker types for each task using performance models and the QoS objective. In particular, the *Auto-scaler* maintains a score for every candidate worker type, incremented whenever it is deemed to be the best. Periodically, as specified by the window size (rule (3)), the *Auto-scaler* compares the score for each candidate worker type to the increase and decrease thresholds (rule (2)), and adds or removes workers to/from the *Worker Group* as appropriate, subject to availability in the *Resource Pool*. We currently define the window size between auto-scaling events by a number of jobs, but this could be extended to support time windows as well.

Consider the example in Fig. 7, illustrating how a *Worker Group* scales with variations in traffic. Two candidate workers and task types are considered: *A* tasks suited to $\boxed{(2,\square)}$ workers, and *B* tasks suited to $\boxed{(3,\triangle)}$ workers. The increase and decrease thresholds are both set to 50% for simplicity, and the task sequence shows a spike in type *B* traffic between mostly *A* tasks. The worker group begins with one $\boxed{(2,\square)}$. In the first window, the majority of tasks are type *A*, and therefore the $\boxed{(2,\square)}$ score is greater than the 50% increase threshold. As such, a new $\boxed{(2,\square)}$ is introduced. In Window 2, there are 50% type *A* and 50% type *B* tasks. Both worker types will have scores that meet the increase threshold (50%), so the *Auto-scaler* attempts to introduce a new worker of each type. However, since there are no available $\square$s in the resource pool, only a $\boxed{(3,\triangle)}$ is added. In Window 3, there are majority *B* tasks. The score for $\boxed{(3,\triangle)}$ is greater than the 50% increase threshold, so a new $\boxed{(3,\triangle)}$ is introduced, while the score for $\boxed{(2,\square)}$ is less than the 50% decrease threshold, so a $\boxed{(2,\square)}$ is removed. In Window 4, both worker types have scores that meet the increase threshold (50%). A $\boxed{(2,\square)}$ is introduced but a new $\boxed{(3,\triangle)}$ cannot be since there are no available $\triangle$s in the *Resource Pool*. Note that the thresholds define how aggressive the scaling is in response to incoming traffic variations.

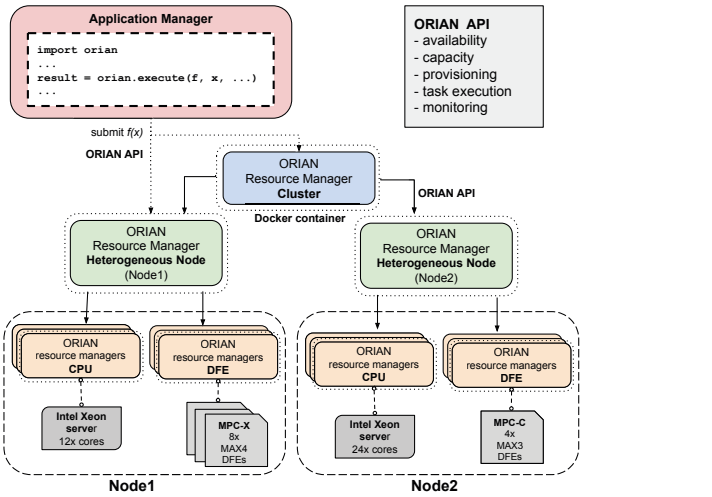
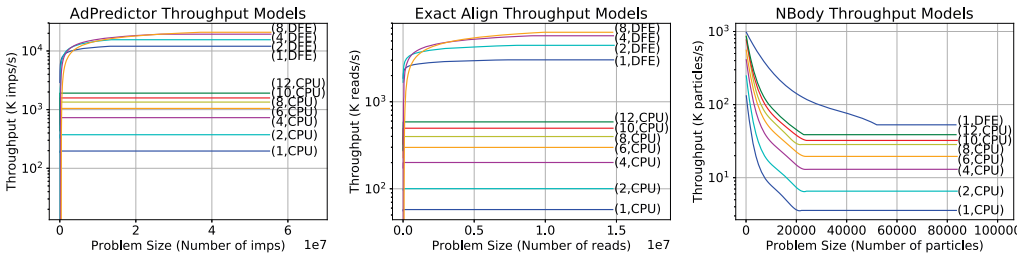


Fig. 8: Our current ORIAN prototype contains four types of resource managers: cluster, heterogeneous node, DFE, and CPU.

III. Implementation

We validate our PaaS approach by developing a prototype (Fig. 8) that targets a hardware infrastructure with two heterogeneous nodes: *Node1* has access to 12 CPU cores and 24 Max4 DFEs [5], while *Node2* has access to 24 CPU cores and 4 Max3 DFEs [6]. A DFE (Dataflow Engine) is a complete compute device system developed by Maxeler [7], which contains an FPGA as the computation fabric, RAM for bulk storage, the necessary logic to connect the device to a CPU host, interfaces to other buses and interconnects, and circuitry to service the device. A DFE provides an abstraction for executing dataflow programs, which operate on streams of data and deep pipelined designs. DFEs have been successfully used for accelerating HPC applications.

Our prototype currently employs four ORIAN Resource Managers that respectively handle: a cluster, a heterogeneous node, DFEs, and multi-core CPUs. We employ a micro-service architecture of hierarchically arranged modular resource managers organised to reflect the levels of complexity of a hardware infrastructure. Each manager operates in an isolated Docker [8] container to ensure isolation of services, and managers communicate with one another via the ORIAN API. Because resource managers share the same interface, managers can be introduced, replaced, or updated while allowing the rest



(a) AdPredictor (b) Exact Align (c) NBody

Fig. 9: Performance models for each case-study function implementation.

TABLE I: Case-study implementation lines of code (LOC) and problem size definition.

Case Study	CPU LOC	DFE LOC	Problem Size Definition
AdPredictor	51	131	# of ad impressions
Exact Align	96	323	# of reads to align
Nbody	24	289	# of particles to simulate

of the system to work as designed, enabling the architecture to grow with the underlying hardware platform.

Our prototype currently supports Python user applications, which interface with the *Application Manager*. In turn, the *Application Manager* connects directly to any *Resource Manager* in the hierarchy using the ORIAN API, accessing the resources that lie underneath (i.e. if connected to the cluster manager, the user application can access CPU cores and DFEs on all available heterogeneous nodes, while if directly connected to a heterogeneous node manager, the user application can access CPU cores and DFEs only on that node). Auto-scaling and task scheduling decisions are made in the top-level *Resource Manager*, and tasks are forwarded through the hierarchy for execution on the selected compute resources.

IV. Evaluation

A. Case Study Applications

Our current prototype supports three functions in different domains: (i) AdPredictor [9] (advertisement click prediction: machine learning), (ii) Exact Align [4] (sequence alignment: bioinformatics), and (iii) NBody (particle simulation: physics). These case studies are examples of HPC applications that are not currently well supported by cloud platforms. Various optimised multi-CPU and multi-DFE implementations for each are available. The CPU implementations are programmed in C++, while the DFE implementations are programmed in MaxJ, a Java DSL used to describe dataflow programs. The LOC (lines of code) for each application's CPU and DFE implementations are included in Table I. With ORIAN, all implementation code for each device as well as the logic to manage their resources are replaced by a single task invocation.

Performance models for each available implementation are derived using the method explained in Section II-C. These models are presented in Fig. 9. To evaluate the accuracy of the derived models, observed compute times for each configuration were recorded and compared to predicted times. The models are observed to be accurate, with an average error across all functions and workers of 3.5% in the saturation regions.

TABLE II: ORIAN and maximum static config. throughput discrepancy.

Case Study	Average Discrepancy		Threshold
	Small Workloads	*Large* Workloads	
AdPredictor	12%	5%	1 million impressions
Exact Align	51%	4%	1 million reads
Nbody	22%	3%	30 thousand particles
Average	28%	4%	n/a

B. Transparent Acceleration

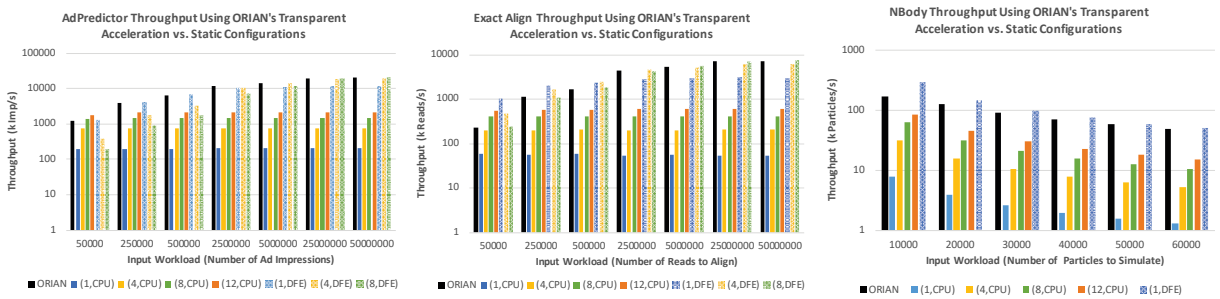
To evaluate transparent acceleration, we compare the throughput achieved by ORIAN-managed configuration selections to various static configurations for problem sizes spanning the domain of each task. For these experiments, we consider only resources from *Node1*, hence the *Application Manager* is configured to connect directly to the *Heterogeneous Node Resource Manager* (see Fig. 8). The complexity of the algorithm used to make decisions is $O(worker\_group\_size)$, so the overhead is constant for a given worker group. In our experiments, decision making takes ∼20ms.

Observed throughputs for ORIAN-managed configuration selections and static configurations are displayed in Fig. 10. ORIAN's performance is close to the maximum throughput in all cases. The average discrepancies between ORIAN-managed configuration throughputs and the maximum static configuration throughputs are included in Table II, where the threshold between *small* and *large* workloads is noted in the table for each case study.

Not surprisingly, the decision-making overhead is more noticeable for smaller workloads, but has less effect on larger workloads that take longer to execute. For large workloads, the average discrepancy between the ORIAN-selected configuration throughput and the maximum static throughput is very low, only 4%. For small workloads, the discrepancy is higher: 28% average across all applications, and up to 51% for Exact Align. However, small workloads complete in sub-second times, and therefore even with a 51% discrepancy, the order of magnitude of compute times for such workloads is unchanged.

C. Heterogeneous Elasticity

To evaluate heterogeneous elasticity, we compare the total cost and objective hit rate of an elastic versus an over-provisioned worker group. For our experiments, we consider the Exact Align application on a single heterogeneous node (*Node1*). We assign a cost of 3 units to 1 DFE and 1 unit to 1 CPU core (this pricing model is arbitrary, and in practice depends on factors such as supply and demand). Fig. 11



(a) AdPredictor (b) Exact Align (c) NBody

Fig. 10: ORIAN transparent acceleration vs. static configuration throughput for each application.

shows the graph derived using our performance models and a latency objective of 8s for all problem sizes. From this graph, ORIAN determines two candidate worker types: **CPU1**: $\boxed{(1,CPU)}$ and **DFE1**: $\boxed{(1,DFE)}$. We choose problem sizes SizeS=$\{500...5000\}$ and SizeL=$\{50{,}000...5{,}000{,}000\}$ suited to **CPU1** and **DFE1** workers, respectively. Note that although a **CPU1** worker costs a third as much as a **DFE1** worker and may meet the objective for a SizeL task, the cost $\times$ compute time product will be less with a **DFE1**. For example, for problem size 50,000 **CPU1** will complete execution in $\sim$1s, costing $\sim$1 unit overall, whereas **DFE1** will complete execution in $\sim$0.01s costing $\sim$0.03 units.

We consider the following task sequences and worker groups:

Task Sequences

- Uniform S: 60 SizeS tasks
- Uniform L: 60 SizeL tasks
- Non-Uniform SLS (peak): 20 SizeS, 20 SizeL, 20 SizeS tasks
- Non-Uniform LSL (dip): 20 SizeL, 20 SizeS, 20 SizeL tasks

Worker Groups

- Over-provisioned: 3x**CPU1** and 2x**DFE1** workers (fixed)
- Elastic: (1..3)x**CPU1** and (0..2)x**DFE1** workers
  (allocation rules: 1–3 CPU, 0–2 DFE; window: 5 jobs;
  increase threshold: 50%; decrease threshold: 30%)

The total cost and objective hit rate for each group and sequence are included in Table III. Total cost is the product of the time to run the task sequence and the average cost of the resources used in the *Worker Group* throughout.

For uniform traffic, our experiments show that using an elastic group is either comparable or better than using an over-provisioned group. For the Uniform S sequence, using the elastic group is significantly ($\sim$2.5 times) less costly than using the over-provisioned group since the elastic group will

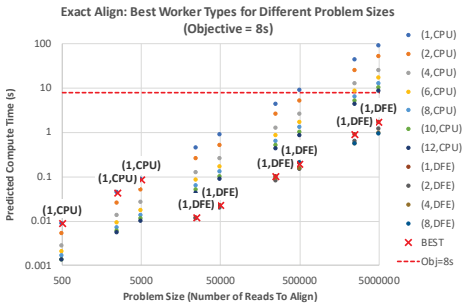


Fig. 11: Best worker types for Exact Align with a latency objective of 8s.

TABLE III: Total cost and objective hit rate for each worker group and task sequence (objective hit rate in parentheses).

Task Sequence	Over-provisioned	Elastic
Uniform S	6.28 (100%)	2.45 (100%)
Uniform L	2021.90 (77%)	2330.18 (83%)
Non-Uniform SLS	273.25 (90%)	119.01 (90%)
Non-Uniform LSL	371.58 (85%)	324.16 (87%)

never include the more costly **DFE1** workers. Both groups meet objectives 100% of the time for SizeS jobs. For the Uniform L sequence, the elastic group is comparable to the over-provisioned group both in terms of cost and objective hit rate. In practice, over-provisioning is used in the absence of elasticity in order to ensure support for maximum traffic (e.g. spikes at peak times). The Uniform L sequence represents constant maximum traffic, so the comparability of the results is due to the elastic group mimicking the over-provisioned group by including all workers to support the maximum traffic.

In general, elasticity is most desirable for unpredictable, non-uniform traffic with occasional peaks in traffic, represented by the SLS sequence. In our experiments, the elastic group is more cost efficient than the over-provisioned group for both non-uniform sequences, and in both cases the groups have comparable objective hit rates. However, the cost reduction provided by the elastic group is highest for peaked non-uniform traffic (SLS), as observed by the 2.3 times reduction in cost, when compared with the over-provisioned group. The SLS sequence represents a common scenario with mostly off-peak traffic, with occasional peak spikes, offering the most cost-effective solution when compared to over-provisioning.

D. Extending the Architecture

So far, in previous experiments, we considered only *Node1* resources. In this experiment, we include resources from *Node1* and *Node2* in order to increase both the type and quantity of available resources. For this purpose, we configure the *Application Manager* to connect to the *Cluster Resource Manager* which handles both heterogeneous nodes (Fig. 8). Applications are now able to access two types of DFEs: Max4 DFEs on *Node1* and Max3 DFEs on *Node2*. With more resources at hand, we increase the optimisation space. In this context, we consider two scenarios, S1 and S2, where S1 only considers *Node1*, and S2 considers both *Node1* and *Node2*.

With an arbitrary latency objective of 3s, ORIAN identifies two candidate worker types for each scenario, listed in Table IV

TABLE IV: Derived candidate workers and the cost of executing AdPredictor with 50 million impressions and time objective 3s.

Scenario	Candidate Workers	Cost
S1 (Node1)	$(1, DFE_{max4})$ for ≤ 25 million impressions $(4, DFE_{max4})$ for > 25 million impressions	10.4
S2 (Node1+2)	$(1, DFE_{max4})$ for ≤ 25 million impressions $(2, DFE_{max3})$ for > 25 million impressions	5.2

TABLE V: Derived candidate workers and throughput of executing AdPredictor with 50 million impressions and resource constraint ≤ 2 DFEs.

Scenario	Candidate Workers	Throughput (Kimp/s)
S1 (Node1)	$(1, DFE_{max4})$ for ≤ 25 million impressions $(2, DFE_{max4})$ for > 25 million impressions	15,549
S2 (Node1+2)	$(1, DFE_{max4})$ for ≤ 25 million impressions $(2, DFE_{max3})$ for > 25 million impressions	19,062

and derived from Fig. 12. Worker types $\boxed{(2, DFE_{max3})}$ and $\boxed{(4, DFE_{max4})}$ achieve similar throughput for large jobs (50 million impressions), therefore such jobs executed in S2 achieve the same performance as in S1 but at half the cost (see Table IV). With a resource constraint (instead of a time objective) of 2 DFEs maximum, the candidate workers identified by ORIAN are included in Table V. $\boxed{(2, DFE_{max4})}$ achieves a lower throughput than $\boxed{(2, DFE_{max3})}$, and thus large jobs in S2 execute 23% faster at the same cost as S1 (see Table V). Hence, the ability to easily extend the platform to include new types and numbers of resources allows the platform to automatically adjust and optimise performance and/or cost.

V. Related Work

Unlike ORIAN, existing cloud PaaS frameworks have limited or no support for heterogeneous resources. For instance, Google's PaaS AppEngine [10] and Microsoft Azure App Services [11] do not support GPU- or FPGA-enabled VMs. On the other hand, AWS Elastic Beanstalk [3] supports instances of any type, including those with hardware accelerators. However, accelerator-optimised instances still require resource-aware applications programmed for specialised resources.

In addition, state-of-the-practice PaaS elasticity approaches are limited to homogeneous horizontal scaling. The auto-scalers used in each of the cloud platforms mentioned above are based on replication of homogeneous instances. For example, PaaS platform Heroku's auto-scaler [2] monitors incoming traffic and changes the number of Dynos in order to maintain a user-specified response time. ORIAN PaaS, on the other hand, combines vertical and horizontal scaling, automatically allocating (and releasing) heterogeneous compute resources - covering both device type and quantity - to match incoming traffic and QoS requirements.

Various state-of-the-art research projects focus on the provisioning, sharing, and programming of specialised heterogeneous cloud resources. For instance, the HARNESS project [12] offers a heterogeneous cloud platform, exposing compute, storage and communication devices as first-class cloud resources. The work in [13] proposes a modular hardware stack with interchangeable layers and an example heterogeneous communication layer for the orchestration of FPGA and CPU cloud clusters based on high-level user descriptions. Unlike ORIAN, neither project supports heterogeneous elasticity.

The work of [14] virtualises cloud FPGAs to enable sharing by mapping accelerators to regions of FPGAs managed

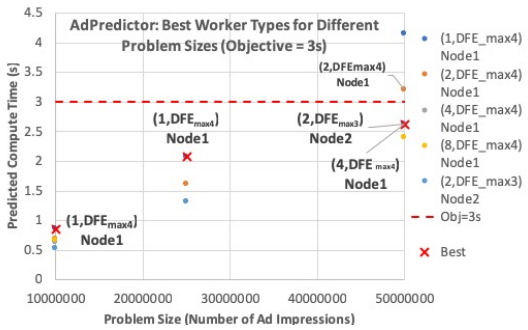


Fig. 12: Best worker types for AdPredictor with time objective of 3s.

by runtime managers on local processors. FPGAVirt [15] proposes a hardware/software co-design framework focused on abstraction, sharing, and isolation using an overlay dividing FPGAs into 'virtual functions' with a management service that maps submitted functions to available regions. ORIAN currently targets dedicated (non-shared) resources, but our work can be directly applied to virtualised resources if they offer appropriate support for performance isolation.

VI. Conclusion

We propose the ORIAN heterogeneous resource management architecture for a cloud PaaS system, extending the traditional homogeneous PaaS model to support heterogeneous worker types. ORIAN's key novelties are its support for *transparent acceleration* on large-scale heterogeneous infrastructures, allowing resource-oblivious applications to automatically tap on to specialised resources, and *heterogeneous elasticity*, the automatic scaling of heterogeneous cloud resources vertically and horizontally to match incoming workload and reduce cost. To evaluate our approach, we developed a prototype targeting three application domains (machine learning, bioinformatics, and physics), on a platform with a combined computational capacity of 28 FPGAs and 36 CPU cores. Our transparent acceleration decisions achieve on average 96% of the maximum manually identified static configuration throughput for different types of workloads, while removing the burden of determining configuration from the user. We also demonstrate that an elastic ORIAN resource group provides a 2.3 times cost reduction compared to an over-provisioned group for non-uniform, peaked job sequences while guaranteeing QoS objectives; and our malleable architecture extends to support a more suitable compute resource type reducing the cost by half while maintaining throughput, and achieving a 23% throughput increase while fulfilling resource constraints.

Current and future work includes the support of a compilation path to allow user-defined computations to be managed, as well as targeting other HPC application domains and hardware accelerators, such as GPUs and application-specific devices.

Acknowledgement

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1), Maxeler and Intel is gratefully acknowledged.

REFERENCES

[1] "Amazon EC2." [Online]. Available: https://aws.amazon.com/ec2/

[2] Heroku, "Scaling Your Dyno Formation." [Online]. Available: https://devcenter.heroku.com/articles/scaling

[3] "AWS Elastic Beanstalk." [Online]. Available: https://aws.amazon.com/elasticbeanstalk/

[4] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging FPGAs for Accelerating Short Read Alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 14, no. 3, pp. 668–677, May 2017.

[5] "Maxeler MPC-X Series." [Online]. Available: https://www.maxeler.com/products/mpc-xseries/

[6] "Maxeler MPC-C Series." [Online]. Available: https://www.maxeler.com/products/mpc-cseries/

[7] "Maxeler Technologies." [Online]. Available: https://www.maxeler.com/

[8] Docker, "Docker," https://www.docker.com/.

[9] T. Graepel, J. Q. n. Candela, T. Borchert, and R. Herbrich, "Web-scale Bayesian Click-through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine," in *ICML*. USA: Omnipress, 2010, pp. 13–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104326

[10] "Google App Engine." [Online]. Available: https://cloud.google.com/appengine/

[11] "Azure App Service." [Online]. Available: https://azure.microsoft.com/en-gb/services/app-service/

[12] J. Coutinho, M. Stillwell, K. Argyraki, G. Ioannidis, A. Iordache, C. Kleineweber, A. Koliousis, J. McGlone, G. Pierre, C. Ragusa, P. Sanders, T. Schütt, T. Yu, and A. Wolf, *The HARNESS Platform: A Hardware- and Network-Enhanced Software System for Cloud Computing*, 2017.

[13] N. Eskandari, N. Tarafdar, D. Ly-Ma, and P. Chow, "A Modular Heterogeneous Stack for Deploying FPGAs and CPUs in the Data Center," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 262–271.

[14] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized Execution Runtime for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017.

[15] J. Mbongue, F. Hategekimana, D. T. Kwadjo, D. Andrews, and C. Bobda, "FPGAVirt: A Novel Virtualization Framework for FPGAs in the Cloud," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018.