Elastic Management of Reconfigurable Accelerators

Paul Grigoras, Max Tottenham, Xinyu Niu, Jose G. F. Coutinho and Wayne Luk
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ, UK
Email: paul.grigoras09@imperial.ac.uk

***Abstract*—This paper presents a runtime system for reconfigurable accelerators that supports elastic management: it enables effective sharing of accelerator resources across multiple applications. For each application, this runtime system allocates an appropriate amount of resources to satisfy its quality-of-service requirements, while minimising the overall execution time for a collection of applications. The effectiveness of this runtime system is due to a set of scheduling algorithms and strategies customised for different types of workloads. We demonstrate our approach by implementing a dynamic Monte Carlo design for pricing bond options.**

I. Introduction

FPGA based dataflow engines (DFEs) can achieve increased performance and energy efficiency for a number of applications [1]. However, despite the orders of magnitude increase in performance and energy efficiency, FPGA based acceleration is neither considered mainstream nor used extensively for high performance computing in industry. We believe this is due to:

- Large technical barrier – application developers coming from more traditional software engineering background have a hard time adapting to FPGA development;
- Slow development cycle – with compilation of circuits taking as much as several days for the largest available commercial chips, the development cycle is much slower when compared to traditional software;
- Limited runtime management – FPGAs and other hardware accelerators are largely invisible to the operating system, and only minimal management functionality is provided;
- Large initial investment – FPGA based solutions require a large initial investment;
- Reduced utilisation – given that, compared to CPUs, FPGAs are less suited for general purpose tasks, peak utilisation may be significantly larger than the mean leading to the risk of FPGA-based devices being underutilised.

Cloud computing can offer a solution to the above challenges and improve adoption of FPGA-based acceleration by (a) providing higher level APIs allowing users to compose applications, (b) improving development speed by providing pre-compiled implementations, (c) providing runtime systems that manage heterogeneous resources, (d) reducing initial investment and commitment and (e) maximising resource utilisation by allowing resources to be shared by multiple tenants.

A key enabler for cloud computing is the ability to provide and manage *elasticity* within the cloud environment. We refer to *elasticity* as the ability to provision and release resources at runtime based on computational demands, providing cloud tenants with the illusion of unlimited resources, and enabling effective sharing of the underlying hardware resources.

Elasticity underpins the dynamic of two conflicting needs: (1) the need to satisfy the user's application requirements as fast and as cost effective as possible, and (2) the cloud provider's need to maximise resource utilisation and increase profit. An elastic system is self-adaptive, and contains two types of components:

- an *elasticity manager* which translates the needs of an application and the feedback from resource managers into a set of provisioned resources that can meet the required computational demand;
- a *resource manager* which makes low-level decisions on how to best allocate provisioned resources to maximise overall throughput and quality of service (QoS). In addition, this component is responsible for providing monitoring information to assist the elasticity manager.

In this paper, we focus on the challenges of enabling elasticity for reconfigurable accelerators in multi-tenant environments. In our approach, multi-tenancy is handled by a scheduler component, capable of sharing a number of available physical DFEs between a much larger number of competing tenants, with different Job-Level Objectives (JLOs). We define a JLO as an objective associated to a job that needs to be satisfied by the scheduler component. An example of a JLO is, for instance, the desired maximum execution time for a submitted job. The goal of the runtime system is to satisfy these JLOs (within bounds of available resources) with a minimum allocation of resources.

The structure of this paper is as follows. First, we present the design of *Hydrogen*, a novel elastic management system for reconfigurable accelerators that operates on a multi-tenant cloud environment (Section III); Second, we show how various scheduling strategies can easily be used with Hydrogen in order to achieve load balancing, reduce average waiting based on priority and realise *elasticity* by allocating, at runtime, DFEs based on the task demand (Section IV); Third, we describe our current implementation of *Hydrogen* (Section V); and finally, we evaluate our system using a dynamic implementation of a Monte Carlo bond option pricing application (Section VI).

II. Background and Related Work

Elasticity has long been recognised as a key feature of cloud computing [2]–[5], to allow resources to be time-shared between cloud tenants. In [6], the authors refer to elasticity as the dynamic provisioning (and de-provisioning) of resources for applications based on workload and service level objectives (SLOs). In [7], elasticity is more rigorously defined as "*the degree*" to which the available resources (provisioned by the system to a specific task) match the current demand, and a number of metrics are proposed to measure the elasticity of a system.

The idea of elasticity has also been studied in the context of reconfigurable computing. For instance, a different concept of elasticity named *elastic computing* has been introduced [8] which supports portable designs (the ability to run one function on multiple platforms) but augmented with a selection model which can choose the best available implementation based on application properties, and pre-compiled performance models. The authors extend their work in [9] which introduces a heuristic for generating parallel implementations for heterogeneous computers.

In this work we look more specifically at supporting elasticity on dataflow engines (DFEs). Dataflow machines emulated on FPGAs can achieve orders of magnitude improved performance [1]. A dataflow design is usually statically scheduled into a deep hazard-free pipeline through which data is streamed. This leads to ideal throughput rate of one result per clock cycle and eliminates the fetch-decode-execute cycle of von Neumann architectures.

Scheduling has been studied extensively and given its classification as an NP hard problem [10], many heuristic based solutions have been developed, which result in good empirical performance.

Classical Monolithic Schedulers such as the O(1) scheduler [11] and the Completely Fair Scheduler (CFS) [12] are commonly used in modern operating systems (e.g. the Linux kernel) and in cloud compute clusters.

Two Level Schedulers are designed to scale to enormous loads and cluster sizes and as such their performance with a very small number of nodes and medium length job size suffers. Hadoop On Demand [13] and Mesos [14] can provision multiple Hadoop clusters on one physical cluster, each having its own scheduler to allow for better resource segregation. However these schedulers suffer from having insufficient information about the physical network and as such can lead to poorer resource allocations [15].

Omega [15] is an example of *distributed/shared state scheduler*. Each node in the network contains global state information, as such each node has information about every other node in the network. While this may be deemed effective for large compute clusters it does mean that there is a significant amount of overhead at each compute node. The goal for our system was to remove the burden of scheduling from the compute resources, allowing them to expend more cycles doing useful work, and fewer cycles determining schedules.

In the context of reconfigurable computing, scheduling has been used to allow the execution of large designs in time-multiplexed FPGAs [16], [17]. Time-multiplexed FPGAs could provide a more efficient way of implementing time-slicing. Efficient preemption schemes for FPGAs have been studied extensively and a number of solutions have been proposed that rely either on customised FPGA architectures or partial reconfiguration [18]–[20]. However tool support for both time-multiplexed FPGAs and preemption is limited and for this reason we have decided to adopt the simpler (albeit less efficient option) of using a software based time-sharing approach as discussed in Section III-C.

III. System Architecture

In this section we describe *Hydrogen*, a lightweight architecture that can efficiently manage DFEs to meet the demand of multiple applications. Hydrogen has been designed to support a cloud computing *platform* as described for example in the HARNESS project [21]. Figure 1 provides an overview of *Hydrogen*. Each *Hydrogen* instance manages a pool of resources that is shared across multiple applications. In particular, applications submit jobs to a *Hydrogen* instance, and the resource manager associated to each instance allocates resources such as to satisfy their JLO. Based on information provided by the resource managers, the *Elasticity Manager* is able to provision or release specific reconfigurable resources based on the observed demand, as measured by a JLO_{metric}, thus achieving *elasticity*.

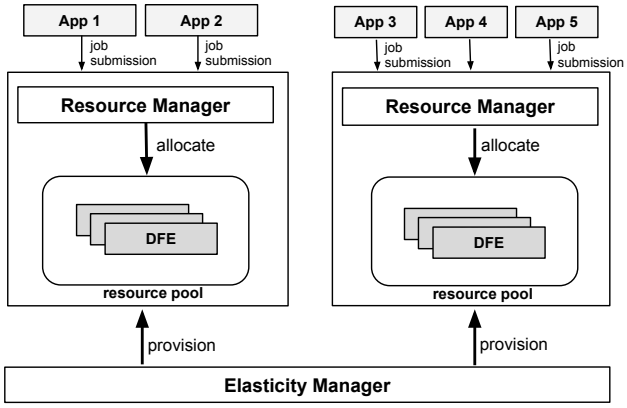

Fig. 1. Overall architecture diagram of *Hydrogen*.

A. System Characterisation

In this context, we view *jobs* as requests for a particular computation (e.g. solve a system of equations, perform a convolution etc.) which are made by clients via Remote Procedure Call (RPC) services (for which the API is provided in the client interface). The API functions correspond to available high performance implementations for available accelerators. This achieves virtualisation of resources behind a single interface and enables *Hydrogen* to be applied not only to manage DFEs but, potentially, heterogeneous accelerators in general. From a

TABLE I
JOB AND RESOURCE CHARACTERISATION USED IN HYDROGEN.

Item	Parameter	Description
job	min	Min resources to start computation
	max	Max resources a job can benefit from
	cost_function	Total execution time of the job
	priority	Determines job precedence over resources
resource	id	The unique identifier for the resource
	idle_time	The time the resource spends in idle state
	utilisation	$\frac{total\_time - idle\_time}{total\_time}$

user application perspective, a single heterogeneous resource (the best available as deemed by the system) is presented to them. Programmers only have to manage wrapping up their computation into jobs which the elastic framework can manage.

At this stage, the proposed system only supports DFE based implementations, but there is no inherent restriction on the type of accelerators which can be used. Accelerators will be referred to as *resources* and it is the job of the runtime to manage elastic applications and assign resources to them. The runtime can dynamically select scheduling algorithms to efficiently distribute the load over the resource cluster (Section VI).

B. Components

The *Hydrogen* architecture allows the resource pool to grow and shrink at runtime. This framework has an overall goal or *strategy*, which includes minimising the total completion time of all incoming jobs or minimising job start latency. The main components of the proposed design are:

- An *implementation library* containing efficient dataflow designs, associated performance metrics (both measured and estimated), and scalability information such as how many DFEs can the implementation scale and what is the required topology;
- A *scheduling library* which contains algorithms for producing an out of order execution schedule that maximises customisable objectives;
- An *elasticity manager* which is responsible for deciding on an algorithm from the *scheduling library* that will minimise the objective function from the current *Strategy*, and adjusting the resource pool size;
- A *resource manager* which allocates jobs to provisioned resources based on the scheduling algorithm.

This approach has a number of benefits. First, it greatly improves ease of use of (heterogeneous) accelerators; in particular for DFEs this is a major concern when considering the increased compilation time (hours, even days) for a high performance implementation. Second it enables effective sharing of implementations in a portable and transparent way to the users. Finally it enables effective sharing of multiple DFE accelerators; this is important since DFEs are on the one hand extremely expensive but on the other hand they offer increased performance and energy efficiency and can often remain unused for long periods of time.

The framework operation comprises the following main stages:

1. *Initialisation* - *Hydrogen* is provisioned with a number of *physical* reconfigurable accelerators (e.g. by the cloud platform)
2. *Job Service* - wait for incoming jobs and on receipt:
   a. insert the job into the Ready-Queue
   b. generate an execution schedule
   c. wait for resources to become available:
      i. dispatch job for execution on DFE
      ii. remove it from the Ready-Queue
      iii. insert it into the Run-Queue
      iv. wait for results (all resources)
      v. return resources to the Resource Pool
      vi. send results to client
3. *Adapt* - based on the observed JLO metrics:
   - Adjust the size of the resource pool
   - Send a request to the cloud platform for more or less resources

C. Job Preemption

Supporting *preemption* is one of the challenges of using DFE based resources in a cloud environment. On one hand preemption is required to ensure fairness with an online, weighted scheduling approach. Without preemption long running jobs may "starve" shorter (and potentially higher priority jobs) from completing and the runtime system would have no means of ensuring their JLO is met. On the other hand, expensive additional hardware support is required to support preemption since unlike CPUs, FPGAs have not been traditionally designed to support rapid preemption.

In Hydrogen we propose to circumvent this issue by using *time-sharing*. In this approach, we would allow dynamic designs to cooperate and willingly give over resources (if requested by the scheduler) after achieving one "unit of work" which is entirely algorithm dependent. For example in an iterative seismic imaging algorithm (such as Reverse Time Migration [22]) the unit of work may be consider one iteration of the outer loop. In case of sorting this could be sorting one subset of the entire input array and so on. One limitation of this approach could be the potential *fragility* of the solution, since one misbehaving implementation can slow down the entire system. In our approach we rely on the fact that dynamic designs are not provided by end-users, but by qualified developers.

With this approach there would be a trade-off between the overhead introduced by the "package mechanism" (i.e. splitting jobs into quanta sized units) and the latency in processing requests. Another issue is that this approach could negate performance benefits achieved by iterative applications [22] which rely heavily on the ability to use on-board DRAM to store large amounts of intermediary results between iterations (this is common in stencil computation) or time invariant matrices (common in iterative solvers for linear equations). The bandwidth to DRAM is significantly larger than on PCI-Express,

so forcing applications to send workpackets through the slow PCIe connection would result in a significant performance degradation. An alternative solution to this problem could be to provide direct network access to the DFE implementations. Recent generations of DFEs (such as the MPC-N box [23]) support direct connection to *multiple* 10GBit Ethernet streams thus potentially allowing *Hydrogen* to bypass the slow PCIe connection, resulting in improved latency and bandwidth. In some applications, spare on-chip resources could be used for the TCP interface.

D. Runtime Reconfiguration

Another challenge of time-sharing DFEs is the overhead of runtime reconfiguration. Unlike CPU cores, in which context switching can incur an overhead of about 30μs, the time to reconfigure a DFE is about 1s. Moreover, in the current generation of Maxeler DFEs, *all DRAM data is lost* after reconfiguration since to achieve reconfiguration, the entire card is reset. This means all data (including potentially large, time-invariant matrices for example in the case of sparse iterative solvers) will have to be re-transmitted to the DFE.

To minimise this overhead, the scheduler needs to keep track of the configuration associated to each provisioned DFEs across multiple runs, and allow an allocation strategy that minimises the number of required reconfigurations at the beginning of every run. For the context of this paper, we do not minimise the runtime reconfiguration overhead and our hardware experiments reflect its full cost.

IV. Scheduling Strategies

An important component of *Hydrogen* is the scheduler which enables multi-tenancy and supports elasticity. Using standard terminology [24], the scheduler is solving the problem of allocating a set of jobs R to multiple machines (DFEs) running in parallel (P_m). A job r_j can run on one or more machines in $M_j \subseteq P_m$ as soon as the job is available. Switching machines from one subset to another incurs a penalty and the objective function we wish to minimize is the total weighted completion time of each job $\Sigma w_j C_j$. Specific to DFEs we include in our analysis the reconfiguration time (when a currently unavailable function is requested by a client, or when resources need to scale to match user demand). This can be thought of as an added constant Δs.

We use a set of strategies to score allocations produced by the scheduling algorithms. We use these scores to determine the final scheduling allocation. Additionally we also bound the impact of running several scheduling algorithms by providing a window size on which they operate. It is simple to add and remove scheduling algorithms for comparison in the elastic framework. *Hydrogen* also allows for more detailed cost functions to be developed on a per job basis. For this paper, we simplified the cost functions for every job to:

$$f(j) = \frac{\text{j.defaultTime}}{\text{j.allocatedResources}} \quad (1)$$

The *defaultTime* is a parameter which governs what the default runtime for a job would be given its allocation to one resource. Therefore the cost function of Equation 1 assumes a linear scaling of performance with the number of *allocatedResources* for a job. This assumption is realistic only for applications which do not incur a penalty (e.g. induced by additional communication overhead) when extending to multiple resources, such as some Monte Carlo applications. In practice, a more generic and accurate performance model for a specific implementation could be determined either via machine learning (e.g. linear regression) or could be estimated by high-level analysis of the original design. Machine learning could work well in the wider context of cloud computing, since performance models could be improved based on a wealth of available data such as measured execution times in correlation with other parameters (system load, application specific parameters, data size etc.). High-level analysis itself could also be a promising approach, since the performance of some dataflow designs can be estimated accurately (e.g. for stencil computation).

A. Strategies

Scheduling can either be fixed with one algorithm or run in *Managed Mode*, whereby the Scheduler uses information about the first *Window Size* jobs in its ready queue to make a scheduling decision. Scheduling happens in O(w) time, as the decision is bounded exactly by the window size. As the window size initially is fixed this means that scheduling decisions happen in O(1) time. Algorithm 1 shows the algorithm for the managed mode and Algorithm 2 shows some of the algorithms we have used with our approach.

Algorithm 1 Managed mode algorithm.

```
1: function MANAGER(queue)
2:     for Alg ∈ SchedulingAlgorithms do
3:         allocations[a] ← Alg(queue, WindowSize)
4:     end for
5:     for alloc ∈ allocations do
6:         scores[alloc] ← score(alloc)
7:     end for
8:     SelectedSchedule ← selectMaxScore(scores)
9:     ElasticityManager(SelectedSchedule)
10: end function
```

The scoring process is determined according to the framework's current strategy. If maximising fairness then the scheduler puts a higher priority on minimising the number of late jobs weighted by their priorities, i.e. a high priority late job incurs a bigger penalty than a low priority late job. When focusing on the total completion time the scheduler focuses on maximising the total number of completed jobs whilst minimising the makespan, the time taken between the first job starting and the last job finishing, of the allocation.

Previous work has shown that it is possible to determine acceptable schedules whilst maintaining a bound on the number of jobs considered in the calculation [25]. There are a few trade-offs to this approach. First, the larger the window size the longer the computation takes to allocate jobs. The smaller the window size the quicker a job gets allocated, but the quality of

Algorithm 2 Scheduling algorithms used with our approach.

```
1: function FCFSMAX(ReadyQueue)
2:     for job ∈ ReadyQueue do
3:         if availableResources > job.max then
4:             allocate(job, job.max)
5:         end if
6:     end for
7: end function
8:
9: function FCFSMIN(ReadyQueue)
10:    for job ∈ ReadyQueue do
11:        if availableResources > job.min then
12:            allocate(job, job.min)
13:        end if
14:    end for
15: end function
16:
17: function FCFSAMAP(ReadyQueue)
18:    for i ← (0..windowSize)) do
19:        job ← ReadyQueue[i]
20:        if availableResources > job.min then
21:            allocate(job, job.min, job.max)
22:        end if
23:    end for
24: end function
25:
26: function SJTF(ReadyQueue)
27:    repeat
28:        minJob = q.getMinJobByCost()
29:        if availableResources > minJob.min then
30:            allocate(minJob, minJob.min)
31:        end if
32:    until availableResource > minJob.min()
33: end function
```

the allocation may suffer. Second in order to use this scheduler effectively, a library or API must be created in order to wrap the tasks into Jobs submitted to the framework, additionally it relies on good performance profiles and cost functions for each Job. If Job parameters are unknown then this approach may not be optimal. However in the cloud environment, it can be quick to build up cost models of commonly used jobs using historical data.

B. Elasticity

The objective of *Hydrogen* is to achieve elasticity by reducing the number of resources allocated at all times while maintaining JLOs (i.e. their target execution times). In our approach elasticity is achieved by adding more resources of a particular type when, based on an execution schedule, we *estimate* that at least one job would fail to meet its JLO based on the number of resources it has been allocated as shown in Algorithm 3. Similarly, when all jobs meet their JLO and at least one job exceeds its JLO (by a term β, a simple means of specifying a preference for scaling up to meet user demands rather than scaling down to reduce costs for the cloud provider) the framework will decide to remove resources of that specific type.

The jloMetric indicates whether the JLO has been met and to what degree: a positive jloMetric means that all jobs for a specific resource meet their JLO; a negative jloMetric means that not all jobs meet their JLO; the magnitude of jloMetric should show *by how much* do jobs miss their JLO; so a higher jloMetric could mean *Hydrogen* should provision / deprovision more than one resource. We consider the jloMetric from the moment jobs enter the queue (i.e. ignoring some of the network transfer). Therefore the Elasticity Manager can decide whether to provision or deprovision resources based on the $jloPos$ and $jloNeg$ metrics which are the sum of per job positive and negative JLOs as shown below.

Algorithm 3 Elasticity manager loop.

```
1: function ELASTICITYMANAGER(Allocations)
2:     jloPos ← 0, jloNeg ← 0
3:     for (job, resources) ∈ Allocations do
4:         jloMetric ← job.getJlo(resources)
5:         if jloMetric < 0 then
6:             jloNeg ← jloNeg + jloMetric
7:         else
8:             jloPos ← jloPos + jloMetric
9:         end if
10:    end for
11:    if jloNeg < 0 then
12:        scheduler.provision(job.resourceType)
13:    else
14:        if jloPos > β then
15:            scheduler.deprovision(job.resourceType)
16:        end if
17:    end if
18: end function
```

$$jloPos = \max(\sum_{j_i>0} j_i - \beta, 0),\ jloNeg = \sum_{j_i<0} j_i$$

V. IMPLEMENTATION

We report our ongoing effort to implement the proposed design on top of Maxeler's Dataflow Engine software stack. The important components of our current *Hydrogen* implementation are:

- *Scheduler* - a resource manager that achieves load balancing, and devises an out of order job execution schedule to satisfy objectives as best as possible (described in detail in Section IV)
- *Elasticity Manager* - aggregates information about individual jobs in a particular execution schedule and decides how to minimally resize the resource pool to best meet execution demands
- *Dispatcher* - thin layer that has direct access to the DFEs (and other computer resources) it manages
- Library of dynamic implementations
- *Client Interface* - the interface through which clients submit compute jobs to *Hydrogen*

Figure 2 provides an overview of how these components integrate.

A. Dispatcher

MaxelerOS manages the communication between the CPU and the DFEs on the MaxNode. The dispatcher is a thin layer

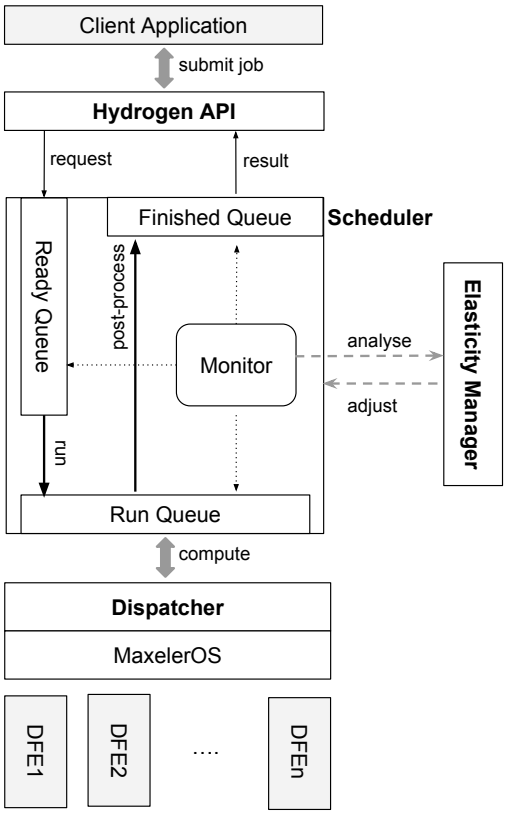

Fig. 2. Implementation level diagram of a *Hydrogen* instance.

on top of MaxelerOS. It accepts incoming requests and runs them on available resources using a bitstream implementation library which it manages directly. On incoming requests, the dispatcher will load the required bitstream onto the specified DFE and start execution.

The dispatcher allows selection between the MaxelerOS managed pool of devices and direct access to resources on request. The first method provides some mechanism to share DFEs at a node level. However this incurs an overhead (use of synchronisation primitives required to enable non-destructive sharing of resources) which is not justified in the context of *Hydrogen* (which already knows the status of devices and can therefore prevent inferences between applications at the system level). An additional advantage of using the scheduler in our approach is the possibility to easily changing the scheduling algorithms to more advanced algorithms (whereas the MaxelerOS scheduler uses only a simple FIFO based approach).

B. Dynamic Implementations

To evaluate our approach we manually designed a number of dynamic implementations. By *dynamic implementation* we understand that the design can run efficiently on an arbitrary number of DFEs as provisioned by *Hydrogen*. Such a design takes the number and ids of DFEs to run on as parameters. Then, using OpenMP, all DFEs are loaded and the corresponding bitstreams start streaming data in parallel. Finally merging of the results is done on CPUs (as required by the particular application) and the overall result is returned to the dispatcher which then transmits it to the scheduler.

C. Client Interface

The client interface works on behalf of the client sending requests to the scheduler. It is responsible for data marshalling and un-marshalling (packing service name, client identification, parameters and job data) and keeping track of client identification information (e.g. user id, client priority, etc.)

Clients can submit jobs through the Hydrogen API, which provides a set of services which correspond to accelerated reconfigurable implementations. This works through a simple Remote Procedure Call (RPC) mechanism implemented on top of the Boost Asynchronous IO library [26] and provides custom made marshalling and un-marshalling functions for various types of messages. A message contains: (a) a unique identifier which identifies the type of computation to be performed, (b) data that the computation operates on, (c) additional (scalar) parameters required for the computation, (d) data and parameter count (to enable un-marashalling), (e) client identification. To reduce the message size, the transmission format is binary (instead of an XML or ASCII based format).

VI. Evaluation

We evaluate our approach on a dynamic Monte Carlo design for bond options pricing implemented as described in section V-B. Monte Carlo simulations are widely used in the finance industry to model interest rate to price fixed income products. In the past two decades, the field has evolved from modelling a single instantaneous interest rate [27] to modelling the dynamics of an entire forward rate curve [28]. A forward rate curve is modelled as:

$$df(t,T) = \sigma(t,T)\int_t^T \sigma(t,u)dudt + \sigma(t,T)^T dW(t) \quad (2)$$

where $f(t,T)$ is the forward rate at time T started from time t; $\sigma(t,T)^T$ is the forward volatility column vector; $W(t)$ is a random variable under standard normal distribution. For each Monte Carlo path, a random $W(t)$ is used to construct a forward rate curve. The generated forward curves are used to value fixed income financial products.

A bond option is a financial instrument which provides the owner of the option with the right to buy or sell a bond at a fixed price K in the future. A *call option* allows owners to buy asset, while a *put option* allows owners to sell asset.

The payoff of the bond option at time T $v(t,T)$ can be expressed as:

$$v(t,T) = \max(exp(-\int_t^T f(t,u)du) - K, 0) \quad (3)$$

To accelerate the bond option pricing process, the Monte Carlo paths and the payoff functions are implemented in FPGAs. The design uses OpenMP and the Maxler API to

TABLE II
SCALABILITY OF A MONTE CARLO APPLICATION ON THE PROPOSED ELASTIC FRAMEWORK. EXECUTION TIMES ARE USED AS PART OF THE PERFORMANCE MODEL FOR DECIDING RESOURCE ALLOCATION.

Paths (1E6)	FPGAs	Time (s)	Speedup	Predicted	Error (%)
1	1	5.91	1.00	5.91	0.00
1	2	3.16	1.87	2.96	6.52
1	3	2.16	2.73	1.97	8.96
1	4	1.67	3.53	1.48	11.63
2	1	11.78	1.00	11.78	0.00
2	2	6.14	1.92	5.89	4.09
2	3	4.25	2.77	3.93	7.64
2	4	3.35	3.52	2.95	12.06
3	1	17.71	1.00	17.71	0.00
3	2	9.11	1.94	8.86	2.80
3	3	6.43	2.75	5.90	8.22
3	3	5.00	3.54	4.43	11.46

operate on multiple FPGAs in a map-reduce fashion: the workload is distributed across a variable number of FPGAs (up to 4 Xilinx Virtex 6 devices, depending on the demand and the *jloMetric*) and then reduced on the CPU of the Maxeler system. The Monte Carlo paths and the payoff functions are implemented on the FPGA. We use a LUT Optimised uniform random number generator [29] to feed the Monte Carlo paths. Design parameters such as volatility and strike price are passed from CPUs, and the evaluated payoff results are averaged in CPUs to price the target derivative.

We implemented a design with 4 parallel processing elements which uses 20.25%, 13.59%, 9.40% and 6.75% of the maximum available lookup tables, flip flop, block ram and digital signal processors respectively. The corresponding execution times including the framework overhead are shown in Table II. The speedup scales linearly with the number of FPGAs, although some overhead is introduced by the framework and by the sequential reduction process.

To demonstrate the elasticity of the framework we evaluate it on a scenario in which a single client performs repeated requests of increasing and then decreasing sizes, successively increasing and decreasing the number of paths explored in the Monte Carlo simulation. As a jloMetric we used the difference between the target execution time and the expected time (predicted based on the results shown above). We include a condition to not scale down unless the jloMetric is positive and greater than 2.5 ($\beta = 2.5$). This is a very simple mechanism of specifying a preference for scaling up to meet client demands over scaling down (for example to reduce energy consumption). The framework cannot scale beyond its minimum (1) and maximum (4) available resources.

Table III shows how the framework could adapt in this scenario to increase the pool size when the jloMetric is negative (thus when failing to achieve a user specified objective) or decrease its pool size when the target objectives are achieved too easily.

We tested our framework design running in *Managed Mode* against four different standalone implementations of common job scheduling algorithms via discrete event simulation:

TABLE III
THE SYSTEM CAN ADJUST ITS RESOURCE POOL SIZE BASED ON THE USER SPECIFIED OBJECTIVE (THE TARGET EXECUTION TIME

Paths (1E6)	Target (s)	Expected (s)	jloMetric	Pool	Decision
1	5	5.91	-0.91	1	Scale Up
1	5	2.96	2.05	2	Preserve
2	5.5	5.89	-0.39	2	Scale Up
2	5.5	3.93	1.57	3	Preserve
3	9	17.71	-8.71	3	Scale Up
3	9	8.86	0.14	4	Preserve
2	5.5	2.95	2.55	4	Scale Down
2	5.5	3.93	1.57	3	Preserve
1	5	1.97	3.03	3	Scale Down
1	5	2.96	2.04	2	Preserve

- Variants of First Come First Serve: FCFSMax, FCFSMin, FCFSAMAP
- Shortest Job Time First (SJTF)

To evaluate the efficiency of the scheduler we implemented a discrete event simulator for the scheduling algorithms. This allows us to change parameters easily and investigate their impact on the overall performance. We observed the following metrics:

- Number of Jobs Completed (N_C)
- Average Wait Time (W_T) - the average time between when the job is issued by a process, and the time it is dispatched for processing by the framework.
- Average Service Time(S_T) - The service time is the time taken for a job to be processed. This can vary due to the different number of resources allocated to a job.
- Cluster Utilisation ($U_c = \frac{\Sigma U_r}{NoResources}$)
- Number Late Jobs (N_L) - A job is determined as late, if it has to wait for more than 1 second in the readyQueue before being dispatched.

Each experiment was repeated 10 times, outliers eliminated and we report the average values for each metric and results are shown in Table IV. Each of the scheduling algorithms on our framework were tested under two different scenarios. For both scenarios we used a *Window Size* of 30. We tested the following scenarios:

- **Scenario 1** - Each job is of the same type, the rate at which jobs are issued is relatively low, and the strategy the framework tries to minimize is set to Job latency;
- **Scenario 2** In this scenario there are two different job types, shorter lived service jobs (lower resolution simulation) which use fewer resources, and longer lived jobs (higher resolution simulation) which can utilise more resources; the rate of job dispatches is set relatively high and the strategy the framework uses is set to minimising completion time.

As can be seen in Table IV, the *Hydrogen* managed mode does a good job in achieving a high throughput and utilisation of the cluster. Our framework is clearly able to accelerate applications when load is low as seen by the average job service time. The average wait time is also kept to a minimum and the number of late jobs were exactly zero for all 10 runs.

TABLE IV
COMPARISON OF SCHEDULING ALGORITHMS AND THE MANAGED APPROACH BASED ON SIMULATION RESULTS.

Algorithm	W_T	S_T	U_c	N_C	N_L
FCFSMax	0.8415	0.2816	0.609	785.9	0.4
FCFSMin	0.0032	0.7496	0.490	796.8	0.0
FCFSAMAP	0.0204	0.3576	0.644	796.5	0.0
SJTF	0.0085	0.7499	0.477	778.0	0.0
Managed Mode	0.0123	0.3388	0.600	793.0	0.0
FCFSMax	15.988	0.2917	0.8624	1112.4	1065.6
FCFSMin	0.088	0.7486	0.8177	1330.0	1.6
FCFSAMAP	1.499	0.3341	0.979	1315.4	807.7
SJTF	0.682	0.7485	0.815	1328.5	287.1
Managed Mode	0.099	0.5302	0.916	1330.0	0

With an increased load, the FCFSMax algorithm provides good acceleration of jobs but does not utilise the resources as effectively as the other algorithms, this leads to fewer job completions and a large number of late jobs. The SJTF Algorithm fairs better but does not utilise the resources as well as the elastic framework.

VII. CONCLUSION

We introduce *Hydrogen*, a framework for elastic management of reconfigurable accelerators, and describe its design and implementation. We highlight some of the challenges and issues such as lack of support for virtualisation or preemption and show how these could be overcome, in a multi-tenant environment by using a *scheduler* component and various scheduling policies. Future work opportunities include high-level support for generating dynamic designs (which can scale efficiently to take maximum advantage on *Hydrogen*), experimentation with more scheduling algorithms, different JLO metrics and more applications.

ACKNOWLEDGEMENT

This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 257906, 287804 and 318521, by the HiPEAC NoE, by the Maxeler University Program, by Altera, and by Xilinx.

REFERENCES

[1] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, "Maximum performance computing with dataflow engines," in *High-Performance Computing Using FPGAs*. Springer, 2013, pp. 747–774.

[2] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "Kingfisher: Cost-aware elasticity in the cloud," in *Proc. INFOCOM*, 2011, pp. 206–210.

[3] ——, "A cost-aware elasticity provisioning system for the cloud," in *Proc. ICDCS*, 2011, pp. 559–570.

[4] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Generation Computer Systems*, vol. 32, pp. 82–98, 2014.

[5] K. Salah, "A queueing model to achieve proper elasticity for cloud cluster jobs," in *Proc. CLOUD*, 2013, pp. 755–761.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[7] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proc. ICAC*, 2013.

[8] J. R. Wernsing and G. Stitt, "Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *Proc. LCTES*, 2010, pp. 115–124.

[9] J. R. Wernsing, G. Stitt, and J. Fowers, "The RACECAR heuristic for automatic function specialization on multi-core heterogeneous systems," in *Proc. CASES*, 2012, pp. 81–90.

[10] R. Rudek, "A note on proving the strong np-hardness of a scheduling problem with position dependent job processing times," *Optimization Letters*, vol. 7, no. 3, 2013.

[11] J. Aas, "Understanding the linux 2.6.8.1 cpu scheduler," *SGI*, vol. 22, p. 05, 2005.

[12] I. Molnar, "Completely fair scheduler design." [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[13] Apache Foundation. [Online]. Available: https://hadoop.apache.org/docs/r0.18.3/hod_admin_guide.html

[14] ——. [Online]. Available: http://mesos.apache.org/documentation/latest/mesos-architecture/

[15] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. EuroSys*, 2013, pp. 351–364.

[16] S. Trimberger, "Scheduling designs into a time-multiplexed FPGA," in *Proc. FPGA*, 1998, pp. 153–160.

[17] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. FCCM*, 1997, pp. 22–28.

[18] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Proc. FPL*, 2005, pp. 223–228.

[19] S. Jovanovic, C. Tanougast, and S. Weber, "A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems," in *Proc. AHS*. IEEE, 2007, pp. 358–364.

[20] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: concepts, overhead analysis, and implementation," in *Proc. FPGA*, 2007, pp. 188–196.

[21] J. Coutinho, O. Pell, E. ONeill, P. Sanders, J. McGlone, P. Grigoras, W. Luk, and C. Ragusa, "HARNESS Project: Managing Heterogeneous Computing Resources for a Cloud Platform," in *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2014, pp. 324–329.

[22] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell, "Exploiting Run-Time Reconfiguration in Stencil Computation," in *Proc. FPL*, 2012, pp. 173–180.

[23] Maxeler Technologies, "Maxeler MPC-N Series Dataflow Engines." [Online]. Available: http://www.maxeler.com/products/desktop/

[24] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2008.

[25] E. Shmueli and D. G. Feitelson, "Backfilling with lookahead to optimize the packing of parallel jobs," *Journal of Parallel and Distributed Computing, Elsevier Science*, pp. 1090–1107.

[26] Boost C++ Libraries, "Boost Asynchronuous I/O Library." [Online]. Available: http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html

[27] T. S. Y. Ho and S.-b. Lee, "Term Structure Movements and Pricing Interest Rate Contingent Claims," *Journal of Finance*, vol. 41, no. 5, pp. 1011–29, December 1986.

[28] D. Heath, R. Jarrow, and A. Morton, "Bond Pricing and the Term Structure of Interest Rates: A New Methodology for Contingent Claims Valuation," *Econometrica*, vol. 60, no. 1, pp. 77–105, January 1992.

[29] D. B. Thomas and W. Luk, "High quality uniform random number generation using lut optimised state-transition matrices," *VLSI Signal Processing*, vol. 47, no. 1, pp. 77–92, 2007.