Quality of Service Driven Runtime Resource Allocation in Reconfigurable HPC Architectures

Marcello Pogliani*, Gianluca C. Durelli*, Antonio Miele*, Tobias Becker†,
Peter Sanders†, Cristiana Bolchini* and Marco D. Santambrogio*
*Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy
{marcello.pogliani, gianlucacarlo.durelli, antonio.miele, cristiana.bolchini, marco.santambrogio}@polimi.it
†Maxeler Technologies Ltd., United Kingdom
{tbecker, psanders}@maxeler.com

***Abstract*—Heterogeneous System Architectures (HSA) are gaining importance in the High Performance Computing (HPC) domain due to increasing computational requirements coupled with energy consumption concerns, which conventional CPU architectures fail to effectively address. Systems based on Field Programmable Gate Array (FPGA) recently emerged as an effective alternative to Graphical Processing Units (GPUs) for demanding HPC applications, although they lack the abstractions available in conventional CPU-based systems. This work tackles the problem of runtime resource management of a system using FPGA-based co-processors to accelerate multi-programmed HPC workloads. We propose a novel resource manager able to dynamically vary the number of FPGAs allocated to each of the jobs running in a multi-accelerator system, with the goal of meeting a given Quality of Service metric for the running jobs measured in terms of deadline or throughput. We implement the proposed resource manager in a commercial HPC system, evaluating its behavior with representative workloads.**

I. Introduction

Despite the unprecedented increase in complexity and performance requirements of computing systems, improvements in silicon technologies and fabrication processes cannot guarantee the yearly doubling of system performance of the past decades (Moore's law [1]). With the rise of microprocessors' power densities, power consumption emerged as a hard limit to their evolution [2]. As single-threaded performance leveled off, a paradigm shift was needed to continue increasing performance: this led to the rise of multi- and many-core architectures, and to the shift from instruction-level parallelism to thread-level parallelism. However, "dark silicon" [3] limits multi-core designs: as the power budget constrains how many cores can be integrated on a chip, transistors are under-utilized, and the architectures cannot scale beyond a few hundreds of cores. Furthermore, costs related to electricity and computer room air conditioning are such a relevant part of the datacenter total cost of ownership that it is usual to characterize datacenter costs in terms of dollar per watt [4], [5]. Even in the High Performance Computing (HPC) scenario, classically characterized for striving to maximize the computational performance alone, the trend is clearly going towards the optimization of energy-aware metrics (e.g., performance per watt). This is testified by projects such as the "Green 500" list [6], started in 2005 to rank supercomputers in terms of energy efficiency.

In this context, Heterogeneous System Architectures (HSA) are emerging as a common paradigm to provide high performance solutions for consumer and HPC systems. Trends and projections have shown that using heterogeneous architectures is the most feasible way to achieve exascale performance in HPC systems, keeping the power budget manageable [7]. HSAs exploit different processing elements for different types of tasks: typically, a standard multi-core CPU, which runs the control-intensive part of the applications, is coupled with highly efficient and specialized accelerators that run the computational intensive and performance-critical parts. A popular accelerator is the Graphical Processing Unit (GPU), an architecture originally conceived for graphic rendering, which can be used for general purpose computation through specialized frameworks such as OpenCL [8] and NVIDIA CUDA [9].

Besides GPUs, reconfigurable hardware – usually implemented with Field Programmable Gate Arrays (FPGA) – is emerging as a key player in the HPC context. Although GPUs are good at performing massively parallel floating-point computation, reconfigurable fabric allows to completely change the data-path if the computation requires so. Reconfigurable hardware is able to couple high performance with low power consumption, at the expense of ease of programming. HPC solutions featuring FPGA accelerators are available on the market; an example is represented by the Data Flow Engine (DFE) produced by Maxeler Technologies [10]. Customers can build their own computational cluster or buy compute time from cluster owners following the utility computing paradigm. A classical problem in the context of shared datacenters is that resource provisioning for running applications (virtual machines or, in general, jobs) is usually performed statically by the user. The end user has to determine upfront the resources needed by

This paper appeared at: 2016 IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC)
DOI: http://doi.org/10.1109/CSE-EUC-DCABES.2016.156

its own application, ending up over-provisioning resources. This scenario not only increases the renting costs for the user, but is not even beneficial for the datacenter owner: to achieve high efficiency (thus reducing the datacenter management costs), all the running machines in a cluster must have a high utilization. Current solutions featuring FPGA accelerators are no different and, for this reason, research is moving in this direction [11]; however, previous approaches in the reconfigurable computing domain do not consider the Quality of Service (QoS) delivered to customers, which is of utmost importance in a utility computing paradigm. Furthermore, unlike runtime resource management solutions, static resource partitioning cannot cope with possible failures of datacenter nodes or accelerators.

In this work, we propose a runtime management controller for HPC systems that accelerate computational intensive kernels of performance-critical applications with reconfigurable hardware. The controller aims at dispatching the available reconfigurable processing resources among the running applications in order to guarantee application-specific QoS requirements, specified in terms of deadlines or desired throughput. The controller features two different scheduling policies devoted to resource dispatching. We implemented the controller in a Maxeler HPC system featuring 8 FPGA-based accelerators (Maxeler's DFEs) and validated it considering different instances of a financial application presenting strict and varying time requirements.

This paper is organized as follows. Section II describes the problem; Section III introduces the proposed techniques; Section IV evaluates this work on a Maxeler HPC system using representative workloads. Finally, Section V presents an overview of related work, and Section VI closes the paper.

II. Context and Problem Definition

In this paper, we target a system with multiple accelerators shared among a number of applications. We refer to a Maxeler Technologies MPC-X[1], which contains up to 8 DFEs shared by multiple CPU nodes over an Infiniband network. DFEs are FPGA-based accelerators aimed at large-scale processing, and provide massive parallelism through deep pipelining.

The resource manager supports highly parallelizable applications structured as a series of mutually independent operations; they have a computational intensive and highly data parallel loop, with as few dependencies as possible, running for many iterations. This structure is extremely simple, but representative of scientific computation kernels that benefit from hardware acceleration: it is found in HPC applications such as option pricing, image and video processing, and computation of the correlation matrix (e.g., for brain network analysis).

[1]https://www.maxeler.com/products/mpc-xseries/

A. Execution model

From the point of view of an application willing to use DFE resources, the basic execution model concerns the so-called *actions*, i.e., the atomic units of computation. An action is a single invocation of DFE functionalities with its own application-specific data streams and parameters sent to the DFE. Usually, an action consists in setting scalar values in the DFE, streaming data, running the computation, and finally receiving back streamed data and scalar values. Actions are considered atomic because it is not possible to preempt or cancel them once they have been dispatched, mainly due to the large context-switching overheads involved. Many applications can be split into several actions; if a computation does not depend on transient data stored inside the DFE memory, each action can run on an arbitrary DFE, and actions belonging to the same applications can run in parallel over multiple devices.

The abstraction that supports the decoupling between applications and physical DFEs in the Maxeler framework is called a *DFE group*. A DFE group is presented to an application as a single virtualized DFE backed by a pool of one or more physical devices. All DFEs within a group are configured with the same bitstream (i.e., in Maxeler's terminology, the same maxfile); then, actions submitted synchronously or asynchronously to a group are scheduled onto physical DFEs through a queuing mechanism implemented in the MPC-X device to ensure high DFE utilization.

Applications that do not depend on transient data can submit actions to a group without knowing what is the physical DFE running the computation. Groups can be shared between multiple applications that use the same configuration bitstream and, if they are created as *dynamic* groups, they can be resized at runtime by adding or removing DFEs. It is worth noting that shrinking the group, i.e., removing a DFE, requires waiting until one DFE has finished its current action. Using groups, applications are not aware of the physical resources they are using; this renders the resource allocation process more flexible, and paves the way for dynamic runtime resource management. In fact, although the group abstraction, by itself, does not solve any resource management problem and is just a thin virtualization layer, in the following we will build upon DFE groups to perform automated DFE allocation.

B. Problem Statement

The problem we tackle is to manage allocation for jobs. A job $J = \{a_1, \ldots, a_n\}$ is defined as a set of stateless and independent actions $a_1, \ldots, a_n$ that require the same configuration bitstream. The output of the resource manager is a time-varying well-formed allocation. Given a set of jobs $\mathcal{J}$ and a number N of resources, a well-formed allocation θ is a function $\theta : \mathcal{J} \to \mathbb{N}$ where

$$\sum_{j \in \mathcal{J}} \theta(j) \leq N$$

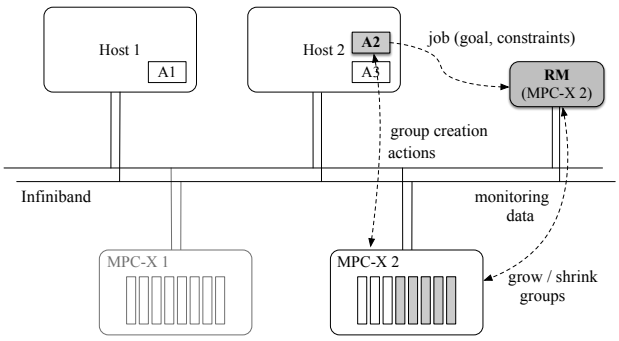

Figure 1. The proposed resource manager in a typical MPC-X deployment.

We allocate resources to meet explicit goals, following a goal-oriented approach. As different applications running on the same HPC cluster may have different performance requirements, goals are job-specific. Each job is described by the following metadata:

- A *goal*, which takes the form of a desired deadline or a desired throughput.
- The job *size*, i.e., the number of actions (if the goal is a deadline).
- An optional *constraint* on the minimum and maximum number of DFEs assigned to the job while it is running; the resource manager will enforce this constraint.

In this framework, the resource manager tackles the following problem: *Given a set of jobs $\mathcal{J}$, and N resources (i.e., DFEs), find a time-varying well-formed allocation θ that satisfies the constraints, so that the goals are met if possible*. Our resource manager leverages DFE groups by creating a dynamic group for each job, and modifying at runtime the group sizes to vary the allocation of DFEs to the jobs.

III. Design and Implementation

The proposed resource manager is a stand-alone user-space application, deployed in an environment composed of one or more high performance switched Infiniband networks, multiple CPU nodes, and one or more MPC-X devices. As shown in Figure 1, it runs on a generic CPU node, is tied to a specific MPC-X, and exchanges data with CPU nodes and the MPC-X over an IP network. A specifically designed API allows applications to register to the resource manager, specify goals, and submit jobs; the resource manager connects with the Maxeler management software to retrieve monitoring data and issue group resize commands. While designing the system, we took into account three important requirements:

- To provide a simple way to plug in new resource management policies and algorithms.
- To provide a generic infrastructure to handle requests from applications.
- To provide an abstraction layer over the retrieval of monitoring data and the change of resource allocation on the MPC-X.

Having the resource manager mediate every DFE-related operation would result in massive performance overhead; thus, the resource manager is a passive component and the managed applications directly perform operations such as loading bitstreams and submitting actions.

We developed two DFE resource management policies, each one targeted for a different workload scenario: the earliest deadline first policy, and the throughput-based policy. These two policies are discussed in the following sections.

A. Earliest Deadline First

The *earliest deadline first* (EDF) policy, applicable to jobs specifying deadlines as a goal, assigns all the allocable resources to the job with the earliest deadline among the ones active in the system. The allocable resources are the DFEs in excess to the ones needed to cope with any metadata-specified constraints. The events that can trigger a change of allocation are the retirement or the submission of a job. In particular, a job submission changes the allocation if the new job is due before the deadline of the currently running job. The EDF policy supports only jobs specifying goals in terms of deadlines.

This policy attains good results in case of batch jobs, when all the results of the computation are needed only after the job deadline, provided that the system is able to meet all the deadlines. The main drawback is a lack of predictability of the response time: as the exact time a job is scheduled depends on the deadlines of other jobs, it is difficult to control the response time when submitting a job.

If the duration of a single action is small with respect to the whole job, the resource manager effectively preempts a job when it changes the allocation. In case of a workload where all the jobs are instances of the same application and use the same bitstream, the problem is similar to classical real-time scheduling with preemption: EDF minimizes the maximum lateness of the jobs. It is optimal in the sense that if there exists a schedule for a set of jobs fulfilling all the deadlines, then also the EDF schedule fulfills all the deadlines. To show that EDF is optimal in this context, we start from the fact that a job is composed of a finite number of discrete actions which can be independently assigned to any resource. Let us consider a set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ having respectively deadlines $d_1 \leq d_2 \leq \cdots \leq d_n$. If a schedule s of $\mathcal{J}$ exists so that all the deadlines $d_1, \ldots, d_n$ are met, then J_1 completes before d_1, i.e., all the actions $a_i \in J_1$ complete before d_1. The EDF policy schedules the jobs in the order of their deadlines: applying the EDF schedule to $\mathcal{J}$, the first actions to be completed are all the ones of J_1. As we consider independent actions, if there exists a schedule where J_1 completes at a time instant $t \leq d_1$, *a fortiori* J_1 completes at a time instant $t' \leq t \leq d_1$

using the EDF schedule[2]. Let us consider a fictitious job J_{12} composed of the actions $\{a : a \in J_1 \vee a \in J_2\}$. Assuming that the schedule s fulfills all the deadlines, then J_{12} completes before d_2 because J_1 completes before d_1, J_2 completes before d_2, and $d_1 \leq d_2$. In this case, the EDF schedule runs the actions belonging to J_{12} before anything else; thus d_2 is met under EDF for the same considerations done for the job J_1. Reasoning in this fashion for the other jobs of $\mathcal{J}$, we can conclude that if there exists a schedule s able to fulfill all the deadlines, also the EDF schedule does so, thus EDF is optimal. This is true assuming that the jobs are known to the system at the beginning, but similar considerations are true also in case of job arrivals spread in time. This reasoning is not valid, instead, in presence of a non-negligible reconfiguration delay.

B. Throughput-Based Policy

In streaming applications, such as real-time video encoding, intermediate results are constantly needed: the important metric is not the job response time, but the throughput, i.e., the rate at which new results become available. Using the EDF scheduler in this scenario, the latency of the first intermediate results is unpredictable: the scheduler can run the application immediately at maximum speed, or even make the application wait for a long time before starting its execution (if other applications have a stricter deadline). For this scenario, we developed a different heuristic that exploits a feedback loop to keep the job *throughput* in a certain range.

Unlike the EDF policy, where the scheduling events are confined to submission and retirement of a job, the resource manager for the throughput-based policy runs periodically, querying the MPC-X device for its status, running the decision algorithm, and issuing group resize commands according to the new allocation. This periodic nature is the main drawback: a short control period, while allowing for a fine grained control, may cause too many reconfiguration events and hurt the system performance; for this reason, in the remainder of this discussion, we consider a workload composed of multiple instances of the *same* application, such as an hardware accelerated video-encoding cluster where different users have different QoS requirements.

This policy uses a target throughput as a goal g; when a job j specifies its goal as a deadline[3], it is translated into a time-varying throughput $g_j(k)$:

$$g_j(k) = \frac{\text{total number of actions}_j - \text{actions}_j(k)}{\text{deadline}_j - \text{time}(k)}$$

Here, actions(k) is the total number of actions completed at the control step k, and time(k) is the current time at k; unlike EDF, the job metadata must specify the total number of actions. We remark that using a time-varying goal and computing the throughput over a time window provides more precise results than averaging the throughput from the beginning of the job, although a requested globally averaged throughput is a more straightforward interpretation for a deadline.

[2]The schedule s executes before the time instant t at least all the actions of J_1, plus possibly others.

[3]Although, as discussed, the EDF scheduler is optimal for deadline-based batch applications, the throughput scheduler is useful for workloads featuring a mix of deadline-based and throughput-based requirements.

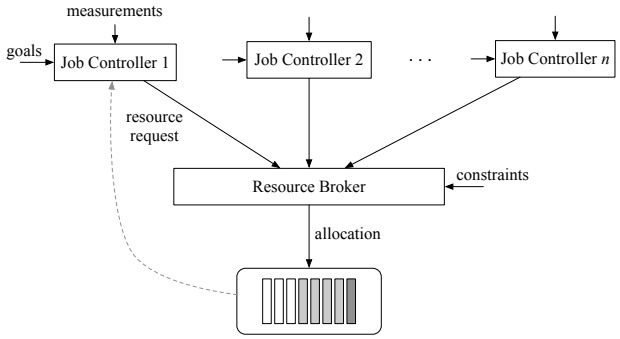

Figure 2. High-level structure of the resource management algorithm.

Figure 2 depicts the high level structure of the throughput-based resource management algorithm. To achieve modularity, we follow an approach already in use for other resource management problems [12]: we separate the algorithm in two layers having a clear interface between them, the *Job Controllers* and the *Resource Broker*. The algorithms used in the job controllers and in the resource broker are based upon heuristics; nevertheless, the split in two levels makes it simple to exploit other techniques for the per-job controller (e.g., formal control theory) keeping low the complexity of the controllers, and leaving to the resource broker the task of coping with constraints. A two layer algorithm, where the job controller request is independent from the other jobs of the workload, can produce less precise results than a single, integrated algorithm which takes into account all the jobs at the same time. Despite this, we believe that the advantages of a simple and easy to adapt structure outweighs its shortcomings.

***Job Controllers*:** The job controllers work independently and in the context of a single job j, computing its "ideal" resource request. They take as an input the job metadata, the current allocation and the completed actions, producing two synthetic values. The first value is a performance metric p_j, which indicates how the performance of j are far from its goal: if j is performing better than its goal, $p_j > 1$, otherwise $p_j < 1$; the resource broker uses p_j to decide which jobs to penalize if some of the requests cannot be satisfied. The second output value is the resource request r_j, expressed as the number of desired DFEs. If the size of actions belonging to the same job is approximately constant and the jobs scale linearly with the number of assigned DFEs, we can compute the resources needed to enforce the required throughput as follows. Let $g_j(k)$ be the desired throughput

of j at the control step k, and $t_j(k)$ the throughput of j computed as in (1), where actions(k) is the total number of completed actions and time(k) is the current time at the step k. The duration of an action can be estimated, as shown in (2), by the ratio between the allocation and the throughput over the last time window, smoothing the resulting value through an exponential moving average filter with a constant smoothing factor $\alpha \in [0, 1]$; by varying the value of α, the algorithm can react quicker or slower to a change of the action duration at the expense of being able to filter outliers out (e.g., particularly slow actions, measurement errors). We then compute the resource request $r_j(k)$, and $p_j(k)$ value as the ratio between the current throughput and the goal, as reported in (3) and (4) respectively.

$$t_j(k) = \frac{\text{actions}(k) - \text{actions}(k-1)}{\text{time}(k) - \text{time}(k-1)} \quad (1)$$

$$\hat{a}_j(k) = \alpha \cdot \frac{\text{allocation}_j(k-1)}{t_j(k)} + (1-\alpha) \cdot \hat{a}_j(k-1) \quad (2)$$

$$r_j(k) = \lceil \hat{a}_j(k) \cdot g_j(k) \rceil \quad (3)$$

$$p_j(k) = \frac{t_j(k)}{g_j(k)} \quad (4)$$

Resource Broker: The resource broker (described in Algorithm 1) computes the allocation according to the values computed by the job controllers and the system constraints. As input parameters, the resource broker takes a list of currently running jobs, and the number of available DFEs (*max*). The value of *max* is the number of available physical DFEs minus any DFEs reserved for jobs initialized but not yet started, and any blacklisted DFE. It is worth noting that a DFE is blacklisted when it fails repeatedly a reconfiguration process. If the resource request fits the MPC-X, the resource broker works as in Algorithm 2: it divides the excess DFEs among the jobs that declared a deadline as a goal: throughput-bound applications do not usually gain in having *more* resources than what they need, due to other synchronization constraints in other parts of the computation pipeline the DFE application is part of. If the resource request does not fit the available DFEs, the resource broker acts as described in Algorithm 3: it penalizes some jobs assigning less resources than required according to a heuristic.

In order to make the job submission process simple for the users, we do not require previous profiling data: when a job enters the system, it is not possible to predict its resource request. However, the throughput-based algorithm targets long-running jobs: for this reason any bad choice in the initial allocation, due to the lack of profiling data, has a small impact on the job performance. In particular, when a job j enters the system and there are enough unallocated DFEs to cope with its constraints, we assign to j the unallocated resources; otherwise, we deallocate DFEs from active jobs ensuring the resource request of each active job is less or equal than the resources it is allocated. If it is not possible, we reject j.

Algorithm 1 Resource Broker

```
function ResourceBroker(jobs, max)
    request ← 0
    for all j ∈ jobs do
        j.allocation ← 0
        request ← request + r_j
    end for
    if request ≤ max then
        AllocateFit(jobs, max, request)
    else
        AllocateNotFit(jobs, max, request)
    end if
end function
```

Algorithm 2 Request fitting in the MPC-X

```
function AllocateFit(jobs, max, request)
    slack ← max − request
    for all j ∈ jobs do
        j.allocation ← r_j
    end for
    dbound ← {j ∈ jobs : j.goal = DEADLINE }
    Sort(dbound) by p
    for all j ∈ dbound do                          ▷ Paused jobs
        if j.allocation = 0 and slack > 0 then
            j.allocation ← 1
            slack ← slack − 1
        end if
    end for
    assigned ← true                                ▷ Other jobs
    while slack > 0 and assigned do
        assigned ← false
        for all j ∈ dbound do
            if slack > 0 then
                j.allocation ← j.allocation +1
                slack ← slack − 1
                assigned ← true
            end if
        end for
    end while
end function
```

IV. Results

We implemented our resource manager on a system composed of a Maxeler Technologies MPC-X2000 device with six *Maia* DFEs (Altera Stratix V D8 FPGA, 48 GB RAM for each DFE), and a CPU node with two Intel®Xeon®E5-2670 (8 cores, 2.6 GHz, 20 MB LLC, Intel HyperThreading) and 64 GB of DDR3 RAM, running the CentOS 6.4 GNU/Linux distribution, and MaxelerOS 2014.2. The MPC-X is connected to the CPU node via two switched Infiniband

Algorithm 3 Request not fitting in the MPC-X

```
function AllocateNotFit(jobs, max, request)
    available ← []
    for all j ∈ jobs do
        prio ← p_j
        for i ← 1 ... r_j do
            available ← available + ⟨prio, j⟩
            prio ← prio · 0.75
        end for
    end for
    Sort(available) by the first element
    allocated ← 0
    for all e ∈ available do
        if allocated < max then
            e[1].allocation ← e[1].allocation +1
            allocated ← allocated +1
        end if
    end for
end function
```

networks; the HCAs are Mellanox ConnectX VPI PCI-Express 2.0, capable of Quad Data Rate (QDR) transfers.

For the evaluation, we chose as a benchmark a simplified option pricing application based on the Black and Scholes formula. This type of option pricing is often performed repetitively in various types of financial risk analytics. For example, Value at Risk (VaR) computes the expected threshold loss on a portfolio of financial assets for a specified probability. Computing VaR requires the repeated pricing of options for an entire portfolio over a range of scenarios, representing a very large workload that can be easily parallelized. The processing kernels are suitable for implementation on co-processors, with low memory cost and large computation/data transfer ratio.

A. Deadline Goals

We ran our system with multiple instances of our benchmark application without considering the reconfiguration overhead. Figure 3 summarizes the execution times of the benchmark application for a static, fixed allocation, and using the resource manager. To obtain those results, we ran the benchmark using a set of static allocations, partitioning the available DFEs among three jobs in various ways (i.e., the 3-3-2 scenario means that the workload is composed of three applications, two of them using a group with 3 DFEs, and one using a group with 2 DFEs). Then, we set the deadlines for the throughput-based algorithm in order to obtain a behavior as close as possible to the one of the static allocation, and we compared the results to the ones obtained using the EDF policy. While the EDF allocation is reported in all the plots for the sake of clarity, it is the same result throughout all the table rows, because its behavior depends only on the ordering between the deadlines and not

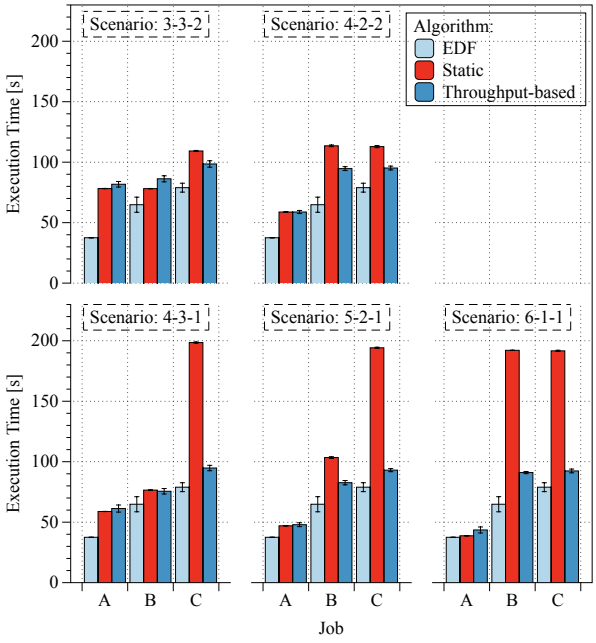

Figure 3. Execution time for a workload composed of three instances of the same application managed through different policies.

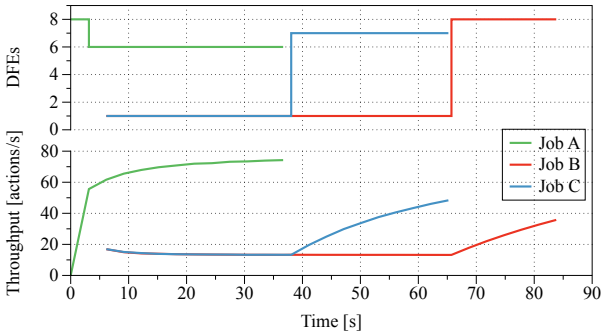

Figure 4. Execution trace of a workload with three instances of the same applications, managed by the EDF algorithm.

on the value of the deadlines themselves. The plots show that using a dynamic allocation improves the utilization of the DFEs: when the job with the earliest deadline finishes, the freed DFEs can be reallocated to the other jobs; furthermore, the performance penalty with respect to the static allocation is low both using both EDF and the throughput-based algorithm. As expected, in this case, EDF leads to better results than the throughput-based policy.

Figure 4 and 5 depict the dynamic behavior of both the developed policies; in the example we consider, deadlines are arbitrarily set to 50, 109, and 195 seconds respectively with a constraint of at least 1 DFE assigned to each job[4]. The top half of the plots shows the allocation; in case of

[4]Those deadlines are set to match the scenario "5-2-1" of Figure 3.

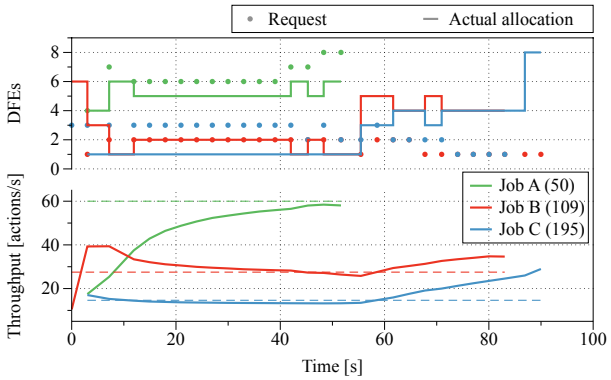

Figure 5. Execution trace of a workload with three instances of the same applications, managed by the throughput-based algorithm; deadlines set to 50, 109, and 195 seconds.

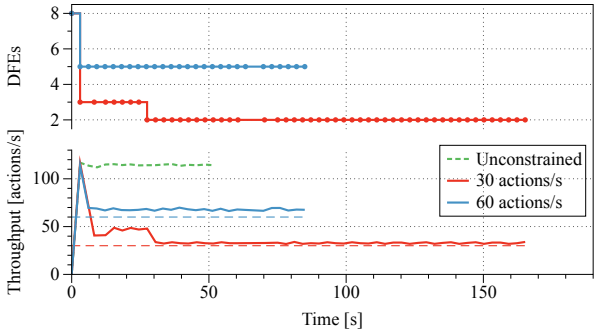

Figure 6. Execution trace of the benchmark application when controlling the throughput of applications running in isolation.

the throughput-based policy, the dots are the request performed by the job controller before the resource brokerage phase, while the lines are the allocation after the resource broker intervention. Although the throughput-based policy internally enforces a throughput over a time window, we chose to represent the global throughput (average throughput from the beginning of the job), as it is representative of goals expressed in the form of a deadline, and allows to easily understand whether the deadline was met. The bottom half of the plot shows the global throughput; the dashed lines in Figure 5 are the deadline goals converted to a global throughput (if the deadline is 200s and the job size is 3000, we draw a dashed line at $^{3000}/_{200} = 60$ actions/s). It is also possible to notice the difficulty to predict when the job will actually start its execution with the EDF algorithm.

B. Throughput Goals

Figure 6 shows the throughput-based policy behavior when the goal is specified as a throughput. The plot shows the throughput over a time window equal to the control period, running a single instance of the benchmark application. The dashed line refers to an unconstrained execution; for the blue and red lines, we set the goals to 30 and

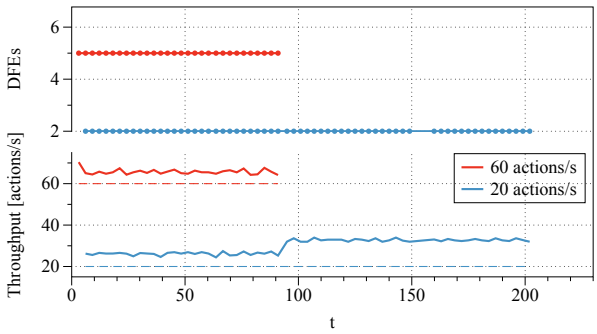

Figure 7. Execution trace of the benchmark application when controlling the throughput of applications executed in co-location.

60 actions/s respectively. The MPC-X allows to assign to a job only an integer number of DFEs: for a goal of 60 actions/s, the throughput is noticeably higher than requested, as using less DFEs would make the system to underperform. Figure 7 shows the behavior of two co-located applications with different goals, 60 and 20 actions/s. After the first job finishes, the throughput of the second job increases without changing the allocation: when both applications run together, the interconnection bandwidth saturates; as soon as the first job ends, the bandwidth is not saturated anymore. Furthermore, changing the allocation to 1 DFE would reduce the throughput to less than the required goal.

V. Related Work

Commodity operating systems do not cope well with HSAs: they manage devices such as GPUs or FPGA-based systems as I/O devices, without higher-level abstractions. Research has recently moved towards building better system-level abstractions for HSA; two fundamental research challenges are resource virtualization and elasticity.

An important topic in *resource virtualization* is to grant virtualized operating systems access to hardware resources without impairing performance, and to solve contention on resources used by multiple operating systems. This topic, originally explored for GPU solutions [13], starts to be explored for FPGA-based systems; for instance, El-Araby et al. [14] tackled FPGA virtualization in the context of devices aimed at high performance applications. Research efforts have focused on extending cloud computing architectures to support heterogeneous processing elements. In fact, accelerators do not fit the cloud computing model as they are not virtualizable in a generic way and are difficult to share. Byma et al. [15] extended OpenStack to provision FPGAs like standard virtual machines; in this view, cloud platforms can offer to users virtualized FPGA resources loaded with precompiled or custom functionalities. The reconfigurable regions have a streaming interface connected to the datacenter network to allow direct communication with the virtual machines.

In a heterogeneous cloud, it becomes desirable to enable *elasticity*. Hydrogen [16] attempts to provide elasticity in multi-tenants environments for the same class of systems targeted by this work. It associates each job with a Job-Level Objective metric (e.g., execution time); each Hydrogen instance manages a pool of resources, and a resource manager allocates jobs to the resources according to a scheduling policy. The resource manager keeps a queue of ready jobs; when resources become available, it runs a scheduling algorithm to choose which jobs to run among the available ones, and how many resources to assign to each one of them. A component (elasticity manager) is associated with all the Hydrogen instances of the platform: it chooses the scheduling algorithm that the resource managers will use, and assigns resources to the Hydrogen instances (i.e., it grows or shrinks the resource pools). The elasticity manager is aware of the jobs' JLOs, and estimates whether the jobs will meet their objectives through a metric that tells by how much the jobs miss their JLOs. A key difference with our work is that we consider long-running jobs and perform online profiling, without needing the offline profiling used in Hydrogen. We also consider general, user-provided hardware implementations, while Hydrogen assumes a fixed repository (library) of high-performance commonly used kernels. Finally, the Maxeler platform used in this work is already harnessing the ideas of virtualization and elasticity [11]. However its runtime allocation mechanism aims to maximize DFE utilization without taking into account any QoS metrics: this results in a poor job for applications with different requirements sharing the same MPC-X.

VI. Conclusion

In this work we proposed runtime resource management techniques for HPC reconfigurable systems, implementing them on a commercial platform with promising results. We extended the MPC-X software layer with a component able to control at runtime applications that offload computations to the managed hardware accelerators. Unlike the state of the art, our system aims to control workloads with QoS-bound jobs, using metrics meaningful to the application. We considered two types of job-specific goals – deadline and throughput – showing that it is possible to control applications having a structure that is widespread among HPC ones. Resource management techniques are good enablers for resource sharing, even on devices conceived as single-application and single-user appliances: resource sharing can extend the suitability of HSAs to a wider user base.

Acknowledgments

This work was partially funded by the European Commission in the context of the FP7 SAVE project (#610996-SAVE).

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, pp. 114–117, Apr 1965.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Comm. of ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[3] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, July 2011.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines," *Synth. Lect. on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[5] C. L. Belady, "In the data center, power and cooling costs more than the it equipment it supports," *Electronics cooling*, 2007. [Online]. Available: http://www.electronics-cooling.com/2007/02/in-the-data-center-power-and-cooling-costs-more-than-the-it-equipment-it-supports/

[6] W. chun Feng and K. Cameron, "The Green500 list: Encouraging sustainable supercomputing," *IEEE Computer*, vol. 40, no. 12, pp. 50–55, Dec 2007.

[7] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the new normal for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, Nov 2013.

[8] *OpenCL – The open standard for parallel programming of heterogeneous systems*, Khronos OpenCL WG Std., 2011. [Online]. Available: https://www.khronos.org/opencl/

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, Mar. 2008.

[10] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 98–103, 2012.

[11] J. G. F. Coutinho *et al.*, "Harness project: Managing heterogeneous computing resources for a cloud platform," in *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2014.

[12] D. B. Bartolini, F. Sironi, D. Sciuto, and M. D. Santambrogio, "Automated fine-grained cpu provisioning for virtual machines," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 27:1–27:25, Jul. 2014.

[13] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, Jul. 2009.

[14] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, "Virtualizing and sharing reconfigurable resources in high-performance reconfigurable computing systems," in *Proc. of the 2nd Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Nov 2008, pp. 1–8.

[15] S. Byma *et al.*, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. of the 22nd IEEE Annual Int. Symp. on Field-Programmable Custom Computing Machines*, May 2014, pp. 109–116.

[16] P. Grigoras, M. Tottenham, X. Niu, J. G. F. Coutinho, and W. Luk, "Elastic management of reconfigurable accelerators," in *Proc. of the 12th IEEE Int. Symp. on Parallel and Distributed Processing with Applications*, Aug 2014.