Resource Elastic Virtualization for FPGAs using OpenCL

DOI:
10.1109/FPL.2018.00028

Document Version
Accepted author manuscript

Published in:
28th International Conference on Field Programmable Logic and Application (FPL)

Resource Elastic Virtualization for FPGAs using OpenCL

Anuj Vaishnav, Khoa Dang Pham, Dirk Koch and James Garside
School of Computer Science, The University of Manchester, Manchester, UK
Email: {anuj.vaishnav, khoa.pham, dirk.koch, james.garside}@manchester.ac.uk

***Abstract*—FPGAs are rising in popularity for acceleration in all kinds of systems. However, even in cloud environments, FPGA devices are typically still used exclusively by one application only. To overcome this, and as an approach to manage FPGA resources with OS functionality, this paper introduces the concept of resource elastic virtualization which allows shrinking and growing of accelerators in the spatial domain with the help of partial reconfiguration. With this, we can serve multiple applications simultaneously on the same FPGA and optimize the resource utilization and consequently the overall system performance. We demonstrate how an implementation of resource elasticity can be realized for OpenCL accelerators along with how it can achieve 2.3x better FPGA utilization and 49% better performance on average while simultaneously lowering waiting time for tasks.**

I. Introduction

Modern SRAM-based FPGAs provide device capacities well beyond a million LUTs which, in many cases, is enough to perform multiple tasks in parallel. Furthermore, High-Level Synthesis (HLS) has become mature enough to allow widespread deployment of FPGAs for general purpose acceleration and recent cloud service installations such as Microsoft Azure, Amazon F1 Instances, and Alibaba Cloud underline this development. However, while these developments are very impressive, FPGA run-time management and hardware virtualization techniques have not progressed at the same pace.

In this paper, we introduce and evaluate the concepts behind *resource elastic FPGA virtualization* which manages FPGA resources in the spatial domain (i.e. how many resources to allocate for a hardware task) as well as in the time domain (i.e. when to occupy a certain FPGA region and for how long). The important novelty of this paper is that applications can readjust resources transparently to the calling application, to maximize FPGA utilization and consequently to optimize overall performance.

This paper assumes that the FPGA provides its resources in slots as shown in Figure 1. In the figure, and for the rest of the paper, we use one dimensional tiling of the FPGA resources into adjacent slots. However, the methodology proposed in this paper could also be applied to systems that tile resources in two dimensions, if needed.

In our approach to virtualization, modules may occupy one or more adjacent slots. Modules may have more than one physical implementation called an *implementation alternative* for using different numbers of resources (e.g., for trading resources for throughput). We also assume that the operation of a long acceleration job can be preempted and that the time to reach the next possible *preemption point* is known or is at least bound.

In the software world, context switching between tasks for virtualization can be performed almost instantaneously. In the FPGA case, however, configuration time of hardware modules is very expensive (typically in the range of many milliseconds). Moreover, extra time is needed to deal with internal states of hardware tasks (which is mainly defined by internal flip-flop values and other memory elements). Therefore, it often makes sense to continue operation before changing the resource allocation for the corresponding hardware task if this saves time/effort to deal with the internal state of the hardware module. For example, if we want to preempt a hardware task that implements a two-dimensional convolution window (as used for several image filters and for Convolutional Neural Network (CNN) classifiers), then it makes sense to only preempt a module at the end of a frame. In this case, the relatively large line buffers that normally keep a history for the convolution window contain no state information needing preservation and, consequently, can be discarded.

In most cases, it is relatively easy to define such dedicated preemption points. In the case of OpenCL (the de-facto standard for describing accelerators in a high-level manner) this is directly dictated by the programming model where a larger compute problem is decomposed into small chunks (called *work-groups*). Here each work-group follows a run-to-completion model, where a completed work-group (and the corresponding hardware module) does not contain any internal state that needs to be considered further. In this paper, we will introduce and demonstrate resource elastic virtualization following OpenCL (see Section IV). However, the concept of resource elasticity can be applied in a much broader context and there are further approaches that implement the idea of preemption points where the state is minimal. In [1], the concept is automated within an HLS compilation framework where a compiler performs live variable analysis to identify operational states with a minimum of memory elements to store.

In a nutshell, assuming a system analogue to the one shown in Figure 1 and a set of somehow preemptable modules, resource elastic virtualization aims at maximizing the FPGA resource allocation transparently to the application which calls the hardware acceleration. This means that if a new arriving task needs FPGA resources, we have to somehow shrink the running module layout to accommodate this task and, if a task terminates, we will expand the remaining tasks such that the FPGA is kept highly utilized for delivering overall better performance. From an application's point of view, only a generic accelerator call is exposed and the entire resource allocation and hardware accelerator management (including FPGA configuration) will be carried out by an FPGA virtualization layer.

The contributions of this paper include:

- Concepts of resource elastic FPGA virtualization (Section III)
- Implementing FPGA virtualization using OpenCL for

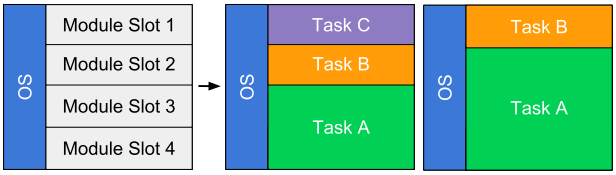

Fig. 1: Physical FPGA layout featuring four partial reconfiguration regions (depicted as module slots). Hardware modules may take one or more adjacent slots and the communication is provided through a hardware operating system (OS) infrastructure.

resource elasticity (Section IV)
- Evaluation of resource elastic schedulers using simulation (Section V)
- Case study on resource elastic schedulers (Section VI).

II. Related Work

FPGA resource virtualization has been examined several times before [2] and the different approaches can be classified in three major directions: overlays, preemptive module execution and virtual FPGAs as described below.

A. Overlays

Overlays or intermediate fabrics [3] provide a layer of programmability that abstracts the low-level details of an FPGA. This approach is commonly used to enhance design productivity by providing a software-like compilation flow rather than a hardware CAD tool flow. Because of this abstraction, overlays can also be used for virtualization.

One direction of virtualization using overlays is targeting portability across different FPGAs (and FPGA vendors) and has been shown for coarse-grained ALU-based arrays (CGRAs) [4], [5] as well as for fine-grained LUT-based overlays [6] or even hybrid systems [7]. This method of virtualization shares some ideas of a Java virtual machine where the same bytecode can be executed on different hardware targets.

Another way to use overlays for virtualization is to apply context switching techniques on the overlay. In its simplest fashion, a softcore processor running an OS would implement this technique. However, the extra layer of programmability of an overlay comes at a substantial cost in performance.

B. Preemptive Hardware Multitasking

The idea of preemptive hardware multitasking is adopted from software systems which virtualize CPUs in the *time domain* where a scheduler allocates CPU time slots to a set of tasks. Similarly, in the FPGA case, systems were built that can stop a running module on the FPGA, capture its state and use the FPGA resources for another hardware task. The preempted task may then be resumed later in a fully transparent manner.

Unfortunately, the state of a hardware module is difficult to capture in a general case and solutions: 1) require restrictions on the preemptive hardware modules, 2) are very target FPGA specific, and 3) are costly in time (e.g., when using configuration readback techniques [8]–[10]) and/or in resources (e.g., when state access is implemented inside the module's user logic [11]). Hardware design techniques including multicycle paths, multicycle I/O transactions, pipeline registers in primitives such as in multipliers and memory blocks, latches or optimization such as retiming are not sufficiently applicable (if at all) with preemptable hardware, hence making this approach impractical for many real-world applications.

C. Virtual FPGAs

The idea of virtual FPGAs is to divide a large FPGA into smaller logical units that can be allocated at run-time. While some resource allocators allow picking a variable number of virtual FPGAs with respect to the currently available resources, the resource allocation is commonly not changed until the allocated resources are released upon task completion [12].

In summary, overlays and preemptive hardware multitasking commonly incorporate too much overhead and virtual FPGAs do not provide enough flexibility to support a holistic solution to FPGA virtualization. Moreover, FPGA virtualization in the time domain alone is, for most FPGA virtualization systems, a pitfall because it omits the spatial programming model which is commonly used along with FPGAs. In other words, we believe that an FPGA should ideally first be virtualized in the *space domain* and (only if needed) in the *time domain*.

III. Space-Time Virtualization: Resource-elastic Hardware

Consider a mostly compute bound system where some randomly arriving and terminating tasks run in parallel. To understand virtualization, let us start from a software scenario where the system would run on a single CPU. In this case, a scheduler would allocate *time slots* to the currently running tasks following a scheduling policy that keeps the *CPU utilization* high with useful work and that implements some means of fairness (or quality of service).

In contrast to this, when using an FPGA, we should keep device *resource utilization* high with useful work. Because tasks with different resource requirements may arrive and terminate over time, this implies that modules are fractionable in *resource slots* that can be allocated by a run-time system. In other words, for virtualizing FPGAs, it is desirable to have a certain level of resource elasticity in the system which we define as:

> ***Hardware Resource Elasticity:*** *the capability of a hardware accelerator to change its resource allocation transparently to the task that is using it. Resource elasticity allows for trading resources for throughput at run-time.*

To support resource elasticity, a scheduler has to perform space domain multiplexing (SDM) by considering various implementation alternatives and/or a variable number of accelerator instances. To demonstrate how this can change utilization and performance, let us assume the scenario illustrated in Figure 2. Because of the ability to change the allocation and to choose the best implementation on an event, we can see that a resource elastic scheduler would allow maximizing the utilization compared to conventional scheduling which only considers fixed-size accelerators. With *event* we refer to a time where scheduling and resource allocation decisions may be taken. In this paper, events include: 1) arrival of a new task, 2) completion of a task and 3) reaching a preemption point, as shown in Figure 2. That example reveals that resource elasticity includes some reconfiguration overhead but results in better resource utilization and consecutively in faster execution (e.g. Task B) and/or higher throughput (Task A).

A resource elastic scheduler must perform three essential trade-offs which are commonly not considered by normal time domain schedulers:

1) Multiple instances vs Different sized modules

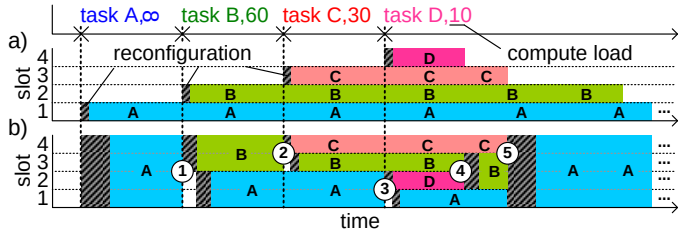

Fig. 2: Resource allocation for kernels (tasks) A, B, C and D in time when using a) Normal fixed module scheduling and b) Resource Elastic scheduling on a 4 slot FPGA. The circled events highlight cases where resources are needed to accommodate new arriving tasks (①, ②, ③) or cases where tasks complete (④, ⑤).

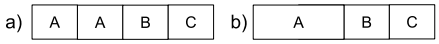

Fig. 3: Logical slot configuration example where a) shows using multiple instances of a single slot module, while b) shows using different sized module, which may have super-linear speed compared to single slot module.

2) Run to completion vs Changing module layout
3) Collocated change vs Distributed change

Trade-off 1 arises at run-time when a module can utilize additional slots for better performance. However, the penalty of pausing the currently running module and performing partial reconfiguration for changing to a different sized module may not be the best option given the work remaining and the possible speed-up achievable with a different sized module. Figure 3 shows available possible module layout alternatives which a space domain scheduler would have to choose from, depending on the aim.

Trade-off 2 considers the case when the work left for the kernel is not enough to amortize the partial reconfiguration overhead. This holds regardless if we use reconfiguration for shrinking and expansion of modules or for defragmenting the module layout. In this case, a decision must be made on whether changing the module layout to achieve better system-level performance at the cost of performance-sacrifice for certain modules is beneficial or not.

Moreover, the decision of selecting a multi-instance option over a different module size may also depend on the location of available free slots which is expressed by Trade-off 3. For example, consider the scenario as shown in Figure 4 where the only available resource slot is at one end of the FPGA. In this scenario, the possible options for maximizing resource utilization for Task A are to either replicate A or to defragment the FPGA such that available slots are collocated to use a larger implementation alternative for A.

Note, the scenarios used in the above description of trade-offs are relatively simple as the focus is just on kernel A. The complexity of the possible situation increases as we consider multiple distinct kernels where it may be necessary to sacrifice more performance for a particular kernel to achieve higher overall performance at system level. Further, the aim of the scheduler may not be performance but fairness or energy etc. which would change policies.

Similar decisions have to be taken by traditional operating systems and different objectives result in different policies (e.g., an embedded real-time operating system follows more different objectives than a Linux OS). From a distance, Figure 1 gives the impression that resource elastic FPGA virtualization is similar to multicore scheduling as done by many

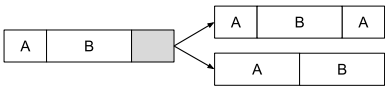

Fig. 4: Example scenario where there are two possible alternatives. 1) To replicate A or 2) To perform defragmentation to use different sized module for A.

software operating systems. However, FPGA virtualization has to consider several very different aspects such that modules may occupy multiple adjacent resources (slots) simultaneously (causing fragmentation issues), that modules may have implementation alternatives that may work on different problem sizes, and that context switching is significantly expensive.

Moreover, if the available slots are not sufficient to accommodate all tasks even when selecting smallest implementation alternatives, our virtualization approach uses time division multiplexing (TDM) as kind of a fallback approach. Analogous to a virtual memory subsystem using disk swapping for providing more than the physically available RAM at a performance penalty, TDM transparently allows the provision of more FPGA slots at some penalty for reconfiguration.

In this section, we highlighted the specific issues of resource elastic FPGA virtualization; in the following section, we will provide a concrete implementation. In this paper, we introduce space-time virtualization for one FPGA in a single operating system scenario. By moving the virtualization management layer from the OS into a hypervisor, the mechanisms proposed here could be used for virtualizing entire operating systems.

IV. FPGA Virtualization for OpenCL

We selected the OpenCL execution model [13] for our case study as it is an industry standard for High-Level Synthesis (HLS) and heterogeneous computing in general. It is described in the following sub-section while the design details of our run-time resource manager for performing resource elasticity will be revealed in Section IV-B.

A. Execution Model

An OpenCL application is made up of a host program and kernels. A *host program* is responsible for submitting computation commands and manipulating memory objects while executing on the host machine. *Kernels* are the compute-heavy functions and are executed on accelerators which are also called *compute units*. When a host program issues a command for kernel execution, an abstract index space is defined, known as NDRange. A kernel is executed exactly one time for each point in the NDRange index and this unit of work is called a *work-item*. There is no execution order defined by the OpenCL standard for work-items. However, synchronization can be achieved with barriers for local memory. Work-items are collated into work-groups and executed on computing units altogether. This provides a coarsely grained decomposition of an NDRange and allows work-items to share and synchronize data inside a work-group. Note that there is no order of execution or sharing of data defined by the OpenCL standard among work-groups. This property allows context switches at work-group granularity since work-groups only share global memory without any synchronization restrictions and because all the results are written back to global memory outside the accelerators at the end of their execution. Further, the independence of work-groups allows launching multiple

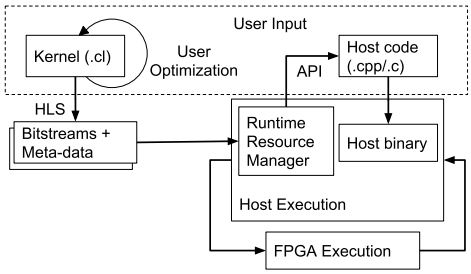

Fig. 5: Design flow for the OpenCL kernel development (design-time) and execution (run-time) with resource elasticity.

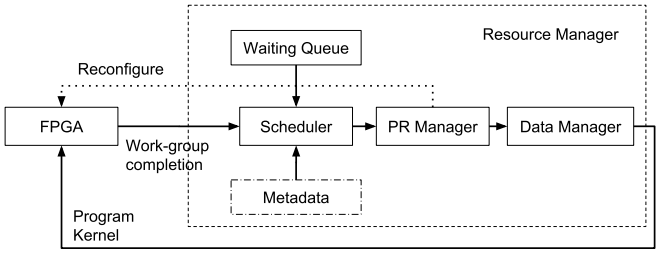

Fig. 6: Resource manager architecture for virtualizing the resource footprint of the OpenCL kernels at run-time.

instances of the same accelerator for scheduling different work-groups concurrently.

Furthermore, the implementation alternatives for OpenCL can be synthesized by using Vivado HLS (or another HLS tool) with different optimization pragmas to implement area vs performance trade-offs. For example, a larger core for matrix multiplication may benefit from better reuse of operands and from more logic for arithmetic. This allows a resource elastic system to benefit from more FPGA resources at run-time.

The complete design flow used for the development and execution of the OpenCL kernel with resource elasticity is shown in Figure 5. The input required by the programmer is an OpenCL kernel with optionally different optimization levels and host code. The only difference compared to the standard Xilinx OpenCL flow for FPGAs [14] is the generation of different kernel versions from the user viewpoint which can be performed easily by selecting the relevant HLS optimization options.

Note that OpenCL could be swapped with any other High-Level Language supported by HLS tools given similar execution semantics with minor modifications. Moreover, it is not a requirement that each module is implemented with different resource/performance variants as we also allow implementing resource elasticity by instantiating an accelerator multiple times. Our resource manager (see Section IV-B) can arbitrarily handle multiple accelerator instances of different size.

B. Resource Manager Design

We have implemented a resource manager that consists of four different components: Waiting Queue, Scheduler, PR Manager and Data Manager, as shown in Figure 6. The Waiting Queue keeps track of kernels waiting for execution and contributes to the implementation of a Round Robin policy for TDM which is used if the demand for accelerators exceeds the available resources (typically given in terms of slots). The scheduler performs Space Domain Multiplexing (SDM) i.e. it decides which modules to shrink or to grow and in what manner given the meta-data (profiling information and available bitstreams) of kernels. Further, the scheduler is also responsible for identifying which module should be replaced when performing TDM. The PR Manager performs the partial reconfiguration requests issued by the scheduler. While the Data Manager keeps track of the work-group execution for each kernel; it is also responsible for programming the accelerators for the next work-group. Note that the whole flow only triggers when an execution command is issued from a host program or if a kernel finishes its work-group execution.

This model shares some ideas of a cooperative operating system that can guarantee real-time behaviour in absence of a global (interrupt) timer. Because the execution of a work-group requires that all data is available and because execution time of work-items is commonly not data dependent (best practice), an upper execution time can be defined, which allows application of resource elasticity under real-time constraints.

Upon the scheduler wake up call, the resource manager retrieves data from all accelerators which have completed their work-group execution and goes through the following stages:

1) Kernel Selection: At first, the resource manager selects the set of kernels which need to be considered for the allocation of resources. This mainly depends on three cases based on the waiting queue size and FPGA state. i) When the FPGA is empty and the waiting queue is non-empty, it extracts the maximum number of kernels which can be run on the FPGA concurrently from the queue, based on their minimum size modules. ii) When the FPGA is non-empty but the waiting queue is empty, the selected kernels are the kernels currently running on the FPGA. iii) When the waiting queue and the FPGA are non-empty, it performs Round Robin scheduling in the time domain by closing all the kernels (removing them from the FPGA) which have completed their work-group execution and inserts them back to the waiting queue. After this, it shrinks currently running kernels by employing a heuristic for recovering the maximum possible slots. We implemented the heuristic to be fair to all the waiting kernels by allocating resources to them as soon as possible. It is possible to replace this heuristic with another to cater for other scheduling aims such as performance or energy. Note that at this stage of scheduling we may have kernels which have one or more accelerators currently running and which cannot be moved around. In this case, we select those kernels plus the maximum number of kernels from the queue which can be executed together, given the free slots available.

2) Generation of Layout Options: Once the kernels are selected, we compute the module layout (i.e. the exact position and size for each accelerator) for the selected kernels. This problem has a large search space as each kernel may have different implementation alternatives available (i.e. different accelerator sizes) which may also be replicated, hence increasing the number of possible combinations. However, this has three constraints which substantially prune the search space:

i. *Availability* only of certain sized modules rather than all implementation alternatives (e.g., a kernel may only have modules which may take at least 2 slots, making it infeasible to be used when only one free slot is available).
ii. *Run-time constraints*: executing kernels cannot relocate, so may constraint available free space.
iii. *Over allocation* i.e. we cannot allocate more slots than the FPGA can provide.

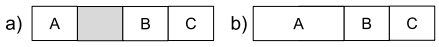

Fig. 7: Logical slot module layout example where a) is completely fair but wastes one of the slots, while b) is not absolutely fair but presents an acceptable trade-off with better utilization.

Currently, we perform this computation by enumerating all possible combinations and removing infeasible module layouts based on the three constraints. Note, since the number of slots on an FPGA tends to be small, the enumeration space will be small as well. However, one may possibly employ heuristics for better run-time at the risk of losing the optimal solution.

3) Resource Allocation: after layout generation, the resource allocator evaluates module layouts based on fairness with the adapted version of the Jain index [15] which accounts for the utilization of slots. Note that with *fairness* we refer to the resource allocation for a set of kernels only. Currently, we do not take the time domain into consideration for fairness. The adapted version of the Jain index is given by Equation 1, where x is the set of kernels in a module layout, n is the number of kernels and x_k denotes the slot allocation for kernel k in the module layout. We added the term $u(x)$ which states the number of slots utilized by the module layout and which can be calculated by Equation 2. We need to consider utilization ($u(x)$) for fairness because absolute fairness may come at the cost of resource wastage. Consider the example shown in the Figure 7: if we were to choose module layout a) we would be completely fair in allocation between kernels by giving each kernel a single slot, however, we would be wasting one of the available slots. If, instead, we choose module layout b) we would not be completely fair but would have an acceptable trade-off for maximizing resource utilization and performance.

The module layout with maximum score is selected by the resource allocator which is formalized by Equation 3. Where $r(x, n)$ is the fairness score of the module layout x and L_f is the set of all feasible module layouts.

$$r(x,n) = \frac{(\sum_{k=1}^{n} x_k)^2}{n \times \sum_{k=1}^{n} x_k^2} + u(x) \tag{1}$$

$$u(x) = \sum_{i=1}^{n} x_i \tag{2}$$

$$\arg\max a(x) = \{r(x,n) \mid x \in L_f\} \tag{3}$$

4) Resource Binding: after allocation of resources for each kernel, the resource binder identifies the best possible module and number of instances for it, in terms of performance considering the resource constraints based on the ranking function. We project the performance possible for each implementation alternative by calculating the time to completion of each kernel based on Equation 4, where x_k is the number of slots allocated to kernel k, $T_c(m)$ is the time to completion of module m, $P(m)$ is the time taken by partial reconfiguration for module m, and L_f^x is the possible combination of modules and number of instances for kernel k with resource constraint of x_k. The time to completion for a given module can be calculated by using the information given by the HLS tool, profiling and/or annotated by the programmer as meta-data. For instance, an OpenCL kernel synthesis can derive the information of the work-group execution time ($T_w(m, k)$) while a programmer can highlight work-group size (W_s). Here, the completion time is *estimated* based on Equation 5, where $W_l(k)$ is the number of work-items left for the execution of Kernel k and is known by Data Manager. The partial reconfiguration cost is specific to the FPGA used and generally scales linearly with the number of slots. It can be modelled by Equation 6, where $P(m)$ is the total latency for partial reconfiguration of module m, P_s is the configuration bitstream size of a single resource slot, N_m is the number of slots required by module m and P_t is the throughput of the configuration port (e.g., the Internal Configuration Access Port - ICAP) of the FPGA.

$$\arg\min p(x_k) = \{T_c(m) + P(m) \mid m \in L_f^x\} \tag{4}$$

$$T_c(m_k) \approx T_w(m_k) \times \frac{W_l(k)}{W_s(m_k)} \tag{5}$$

$$P(m) = \frac{P_s \times N_m}{P_t} \tag{6}$$

Our projection of performance takes into account the acceleration possible with the given module (by calculating its time to completion) and overhead of partial reconfiguration to help tackle the Trade-offs 1 and 2 as mentioned in Section III. The same technique is used to break the tie in favour of the first best performing configuration when ranking by the resource allocator. Thereafter, the resource binder requests PR Manager to load the new module layout.

5) Partial Reconfiguration Manager: upon receiving a reconfiguration request, our current PR manager performs partial reconfiguration for each instance one at a time based on the module layout selected by the resource allocator and resource binder. After reconfiguration, PR Manager instructs the Data Manager to provide input to the new accelerator.

6) Programming Kernels: Data Manager programs the kernels with new input data and also retrieves the output data when available. Further, it handles the case when a kernel change to a differently sized implementation alternative leads to a change in work-group size, such that the final outcome of the kernel still remains the same. It does this by calculating the next work-group indices such that no work items are left out at a new granularity, which may require certain work-items to be re-evaluated. However, this is safe to perform as kernels commonly write their output at a different memory location from where the input operands are stored and as each implementation alternative has the same functionality for each work-item.

This section revealed that resource elasticity fits the OpenCL programming model directly and our resource manager provides a reference implementation for the operating system services which can consider various resource elasticity trade-offs for running OpenCL accelerators. While other heuristics may be tailored to meet specific system requirements (e.g., performance, response time, power consumption), a run-time manager for a resource elastic system always has to provide the space-time mapping of resources, perform reconfiguration and has to manage the execution of kernels.

V. Experiments

To evaluate the characteristics of the scheduling algorithm, we conduct a series of simulation experiments to explore how different scheduling decisions impact our resource elastic virtualization approach. A working case study is presented in Section VI while the following subsections describe the simulation experiments and findings in detail.

A. Experimental setup

With simulation, we explored the effects of scheduling on a wide variety of applications by changing the characteristics of synthetic applications in terms of area requirements and completion time. We also used this to investigate scheduling effects on different sized FPGAs by varying the slots available for scheduling.

We tested our system with 1000 different compute-bound scheduling requirements, where randomly arriving and terminating tasks run in parallel. To model a compute-bound schedule, we generate a long running kernel of 4000 work-groups at the arrival time zero. While the other accompanying kernels in the schedule are generated in the range of [1, 12] for each experiment. Characteristics of each kernel are generated randomly (except for arrival time and number of work-groups of long-running kernel) i.e. base latency of work-groups is in range of [10, 100], arrival time is in [1, 10000], the minimum slot size of kernels is in [1, min(4, n-1)] where n is the total number of slots available on the FPGA and the maximum slot size of a kernel is [minimum slot size, min(4, n)]. The speedup achievable with different size kernel is in [3, 10] and the number of work-groups is in [50, 500]. The respective parameter selection is based on a uniform random distribution. We assume the I/O requirements of kernels are not on the critical path of the application. The partial reconfiguration cost is modelled linearly proportional to the number of slots with the cost of configuring a single slot being 5x the smallest latency possible for a work-group. Note that the given range of work-group latency and speed up available from configuring another kernel, it is likely that the cost of reconfiguration would not be recovered from a single work-group execution, as it would likely be the case in a real-world scenario. Furthermore, restricting the kernel size to a maximum of 4 slots allows us to study the scenario where a user runs the kernels of the given sizes on a much bigger FPGA. The scenario assumed here look similar to the example in Figure 2b showing quite a fine-grained schedule and consecutively relatively high configuration overhead. While the configuration cost is basically the number of slots to be reconfigured over time, the relative overhead depends on for how long the module runs after configuration. This means that the coarser the scheduling granularity is, the lower the relative configuration overhead. Further, it is important to note that our simulation workload shares some similarity with workload traces found in Google clusters [16], with its mix of long and short running kernels.

We ran the randomly generated schedule requirements on five different schedulers. The first three schedulers act as a baseline for our implementations and these are as follows:
1) Normal Scheduler (NS) which simply allocates the tasks at a First-Fit slot and runs them to completion. This is the most commonly used strategy [17]–[19] and is beneficial for FPGAs as the partial reconfiguration cost is quite high. 2) Conservative Cooperative Scheduler (CCS) is a time domain scheduler which performs a context switch when the task voluntarily relinquishes the control (in our case at the end of a work-group execution). However, since the standard cooperative schedulers do not take into account the availability of implementation alternatives, it may always use the minimum sized module to operate in conservative mode. This strategy aims at minimizing the number of reconfigurations required as it would leave maximum space available for new incoming kernels. On the contrary, we also compare against the version labelled 3) Aggressive Cooperative Scheduler (ACS), which employs the biggest module available for achieving maximum possible performance at the risk of higher reconfiguration count.

Our two implementations of resource elastic scheduling are 1) Standard Resource Elastic Scheduler (SRES) and 2) Performance driven Resource Elastic Scheduler (PRES). The implementation of SRES is discussed in Section IV-B. PRES follows the same implementation but resource allocation rating is performed with the aim of maximizing performance. Equation 7 captures the rating used for PRES by minimizing the total completion time for the online kernels, where $T_c(x_i)$ is completion time calculated using Equation 5, x is the module layout and L_f is set of all feasible module layouts.

$$\arg\min a(x) = \{\sum_{i=1}^{n} T_c(x_i) \mid x \in L_f\} \tag{7}$$

B. Results

The average completion time (the most relevant performance indicator) for each scheduler is shown in Figure 8a and the average waiting time for each kernel is shown in Figure 8b. Note we calculate wait time as $w_i = s_i - a_i$, where a_i is the arrival time of the kernel and s_i is the time when it begins its execution (after partial reconfiguration). The completion time is the lowest for NS from the baseline for small FPGAs as it does not have to pay higher reconfiguration penalties compared to other schedulers. Note that despite better performance, NS has poorest wait time from all the baseline schedulers, as it lets all kernels run to completion without interruption. On the other hand, resource elastic schedulers provide considerably lower completion time and similar wait time characteristics as cooperative schedulers (Figure 8b) even incorporating higher reconfiguration overhead. Specifically, if we compare the performance targeting versions of schedulers i.e. PRES and ACS, we can see that PRES can achieve higher performance as much as 39% to 64% than ACS. However, the performance advantage does not scale linearly with the number of resource slots available as they become so abundant that all kernels can run concurrently in their full sized modules for most of the experiments without higher reconfiguration cost when using ACS. Resource elastic schedulers achieve this performance advantage by considering all the possible implementation alternatives and employing them dynamically at run-time to maximize performance.

We measured the resource utilization as the total number of occupied slots after every scheduler wake up call as plotted in Figure 8c after normalization. We can see the employment of ACS leads to the highest utilization from baseline due to the heuristic of always using the largest module for each kernel. However, it does not tend towards the maximum

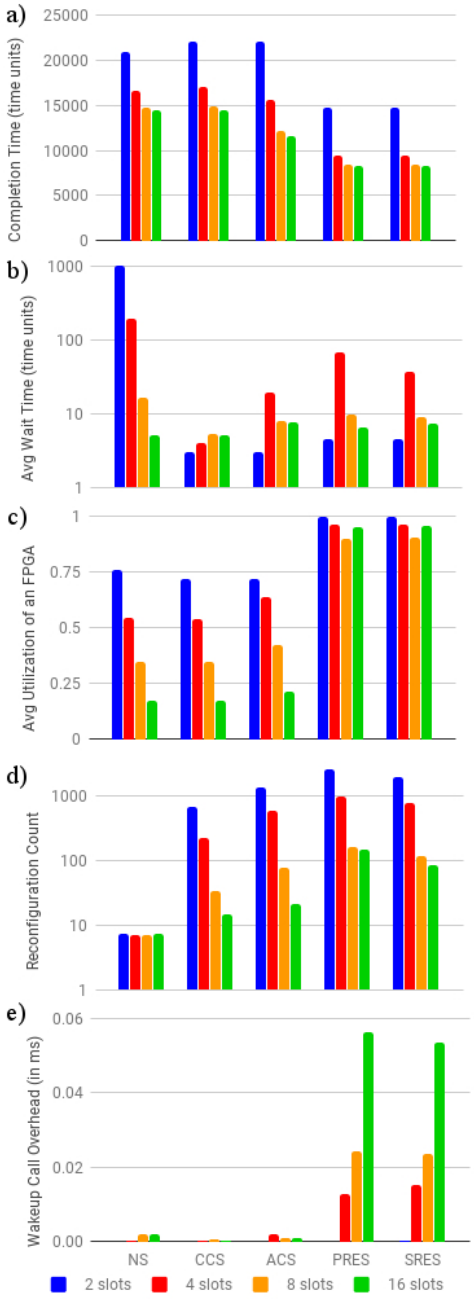

Fig. 8: Performance characteristics of the NS, CCS, ACS, SRES, and PRES schedulers on FPGAs with slot size 2, 4, 8 and 16 where a) is completion time, b) is average wait time, c) is average utilization of an FPGA, d) is the number of partial reconfigurations performed and e) is the average overhead of the scheduler's wake up call.

possible utilization as it cannot overcome the fragmentation repercussions of its heuristic. While on the other hand, the ability to adjust allocation of resources and replication of modules for higher performance helps our resource elastic schedulers to gain almost full utilization, which is about on average 2.3x higher utilization compared to ACS and about 2.7x better on average when compared to NS. The drop in utilization with respect to an increase in the number of slots available occurs because resource elastic schedulers tend to employ the biggest module possible for a kernel to maximize performance at the cost of fragmentation and the scenarios where kernels did not offer smallest module size for filling up the holes left in a particular module layout. Note that the higher utilization is achieved at the cost of a higher number of reconfiguration calls; the implication of this is captured in Figure 8a and 8b, as reconfiguration time is included in the total completion and wait times.

To quantify and contrast the overhead caused by the resource elastic scheduler, we measured the time taken for execution of single wake up calls on an x86 Intel Core i7-6850K running Ubuntu 16.04 LTS and the total number of partial reconfiguration calls performed for schedulers across all the experiments. The results of this are shown in Figure 8e and 8d, respectively. The computation overhead for a resource elastic scheduler is between 10x to 100x higher as compared to baseline schedulers. This is mostly due to our implementation which enumerates all the possible module layouts and uses expensive rating functions. However, the total execution time is still below tens of microseconds which is negligible compared to the partial reconfiguration cost which is in the range of milliseconds. Similarly, Figure 8d shows that the resource elastic schedulers require about 12x to 100x more configuration calls than NS and a similar number of configuration calls to ACS. However, despite this, the performance of the resource elastic scheduler is much better with lower waiting time for the kernels, as shown in Figure 8a and 8b. This is due to frequent partial reconfiguration for increasing resource utilization which, in turn, improves performance.

VI. Case Study

In our case study, we deploy resource elastic scheduling on a recent Xilinx Zynq UltraScale+ platform for the same baseline schedulers and resource elastic schedulers as analyzed in Section V. The platform on which we conducted the experiment is a TE8080 board featuring a XCZU9EG-FFVC900-2I-ES1 MPSoC device. Our system has been partitioned into a static part, which includes ARM cores and AXI interfaces, and the partially reconfigurable part, which is reserved for partially reconfigurable modules. In detail, the partially reconfigurable part has four regions with identical resource footprints which serve as slots, to host partially reconfigurable modules, as shown in Figure 9. In our design, the reconfigurable part occupies approximately 50% of the whole chip resources, and a slot takes around 3 ms for partial reconfiguration.

OpenCL applications are implemented in one or more reconfigurable slots. They can be relocated at run-time using a configuration controller based on the tool BitMan [20]. This allows reusing the bitstreams for different slots and reduces the total number of bitstreams which needs to be stored considerably, as compared to the standard Xilinx PR flow. The workload is offloaded to these kernels from the ARM cores which share the main memory with them.

We conducted the scheduling experiments on this platform with applications from communication, arithmetic and machine learning: CRC32, matrix multiplication (mmult), and Euclidean distance (e-dist) for k-means. The matrix multiplication kernel has two different physical implementations of slot sizes 1 and 4. Where the 4 slot version offers $5\times$ the performance of a 1 slot module due to better reuse of the data and a bigger work-group size. CRC32 and Euclidean distance kernels occupy 1-slot each.

The scheduling requirements for our case study models matrix multiplication as a long-running kernel which needs

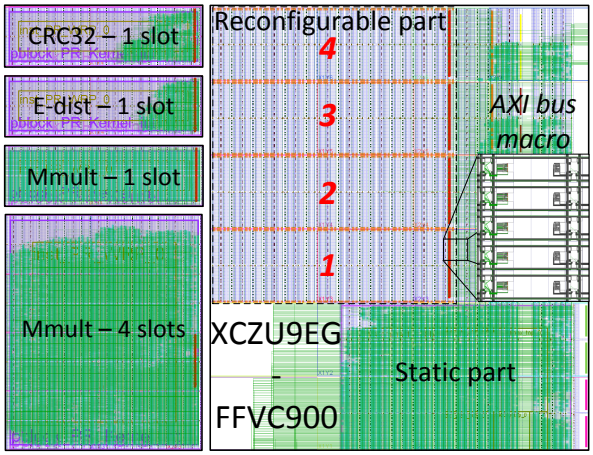

Fig. 9: Physical implementation and floor layout of the chip with dedicated area for partial reconfiguration (slots) and static system, such that OpenCL kernels can use the neighbouring slots if required.

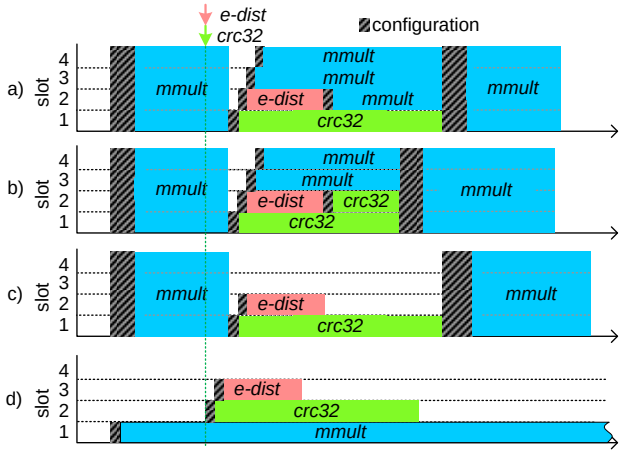

Fig. 10: Execution trace of the scheduler where a) is PRES, b) is SRES, c) is ACS, and d) is NS and CCS

to run 20480 work-items with an arrival time of zero. While CRC32 and Euclidean distance are modelled as short running kernels with 16384 work-items of low latency each and an arbitrarily chosen arrival time of 60 ms, such that it would interrupt the long-running kernel. The overall decisions taken by different schedulers are shown in Figure 10 and their respective performance characteristics are captured in Table I. We can see that ACS provides highest performance and utilization among the baseline schedulers. However, with the ability to grow and shrink modules, SRES and PRES effectively maximize the utilization and provides 36% and 26.5% better performance than ACS with similar waiting time, respectively. In this particular scenario, SRES outperforms PRES due to the lack of looking ahead in PRES, which leads to a greedy decision of accelerating matrix multiplication over CRC32 and hence, a higher total completion time.

	SRES	PRES	ACS	CCS/NS
mmult wait time	12 ms	12 ms	12 ms	3 ms
CRC32 wait time	4 ms	4 ms	4 ms	3 ms
e-dist wait time	7 ms	7 ms	7 ms	6 ms
Total completion time	320 ms	368 ms	501 ms	1221 ms

TABLE I: Wait time of kernels and completion time of schedule for the case study. Where, A/B denotes that scheduling policies A and B provides the same results.

VII. Conclusion

In this paper, we introduced the concept of resource elasticity as a novel run-time resource management solution to allow dynamic allocation of reconfigurable resources for FPGAs. We demonstrated how a resource manager can be designed for OpenCL to virtualize reconfigurable resources for a variable number of running kernels in order to improve utilization, and consequently, performance. Our evaluation against classical resource allocation strategies on a simulator and a physical implementation found that resource elasticity can provide about 2.3x higher utilization and 49% better performance on average while also delivering lower waiting times for the kernels.

Our results are a strong indicator that future FPGA operating systems and virtual machines need to consider mapping accelerators firstly in the spatial domain. We demonstrated that co-operative scheduling is a potentially better fit for FPGAs due to its trade-off between overhead and ability of resource reallocation, compared to state-of-the-art preemptive and run-to-completion approaches.

VIII. Acknowledgements

This work is supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632). We also like to thank Xilinx for supporting this research through their university programme.

References

[1] A. Bourge et. al., "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow," *ACM TRETS*, Dec. 2016.
[2] A. Vaishnav, K. D. Pham and D. Koch, "A Survey on FPGA Virtualization," in *28th FPL*, Aug 2018.
[3] J. Coole and G. Stitt, "Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing," in *CODES+ISSS*, Oct 2010.
[4] A. K. Jain et. al., "Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?" in *14th DASC/14th PiCom/2nd DataCom/2nd CyberSciTech)*. IEEE, 2016.
[5] C. Liu et al., "QuickDough: A Rapid FPGA Loop Accelerator Design Framework Using Soft CGRA Overlay," in *FPT*, Dec 2015.
[6] A. Brant et. al., "ZUMA: An Open FPGA Overlay Architecture," in *20th FCCM*. IEEE Computer Society, 2012.
[7] D. Koch et. al., "An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions," in *23rd FPL*, Sept 2013.
[8] H. Simmler et. al., "Multitasking on FPGA coprocessors," in *10th FPL*. Springer, 2000.
[9] A. Morales-Villanueva et. al., "Partial Region and Bitstream Cost Models for Hardware Multitasking on Partially Reconfigurable FPGAs," in *IPDPS*, May 2015.
[10] M. Happe et. al., "Preemptive Hardware Multitasking in ReconOS," in *Applied Reconfigurable Computing*, 2015.
[11] D. Koch et. al., "Efficient Hardware Checkpointing – Concepts, Overhead Analysis, and Implementation," in *15th FPGA*. ACM, 2007.
[12] M. Asiatici et. al., "Virtualized Execution Runtime for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, 2017.
[13] A. Munshi, "The OpenCL Specification," in *Hot Chips 21 (HCS)*, 2009.
[14] L. Wirbel, "Xilinx SDAccel: a Unified Development Environment for Tomorrows Data Center," *The Linley Group Inc*, 2014.
[15] R. Jain et. el., "Throughput Fairness Index: An Explanation," Dept. of CIS, Ohio State University, Tech. Rep., 1999.
[16] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proc. of the 8th ACM Euro. Conf. on Computer Syss.*, 2013.
[17] S. Byma et. al., "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," in *22nd FCCM*, May 2014.
[18] F. Chen et. al., "Enabling FPGAs in the Cloud," in *Proceedings of the 11th ACM Conf. on Computing Frontiers*. ACM, 2014.
[19] E. Lübbers and M. Platzner, "ReconOS: An RTOS Supporting Hard-and Software Threads," in *17th FPL*, Aug 2007.
[20] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*. IEEE, 2017.