Heterogeneous Resource-Elastic Management for FPGAs: Concepts, Theory and Implementation

Preprint · September 2019

DOI: 10.13140/RG.2.2.15161.93280

3 authors, including:

Anuj Vaishnav
The University of Manchester
18 PUBLICATIONS **25** CITATIONS

SEE PROFILE

Khoa Dang Pham
The University of Manchester
35 PUBLICATIONS **107** CITATIONS

SEE PROFILE

HETEROGENEOUS RESOURCE-ELASTIC MANAGEMENT FOR FPGAS: CONCEPTS, THEORY AND IMPLEMENTATION

A PREPRINT

Anuj Vaishnav,
Department of Computer Science,
The University of Manchester,
anuj.vaishnav@manchester.ac.uk

Dirk Koch,
Department of Computer Science,
The University of Manchester,
dirk.koch@manchester.ac.uk

Khoa Dang Pham,
Department of Computer Science,
The University of Manchester,
khoa.pham@manchester.ac.uk

September 3, 2019

ABSTRACT

Despite deployment of FPGAs at the edge and cloud data centers due to their performance and energy advantage, FPGA runtime systems commonly tend to support only one-application-at-a-time and cannot adapt to dynamic workloads with reasonable response times. Therefore, this paper proposes the concepts and theory of resource elasticity for FPGA systems to allow a task running in hardware on an FPGA to change its resource allocation dynamically through module replication or migrating to implementation alternatives. To support resource elasticity, we identify its system requirements and provide an implementation for OpenCL applications which can not only accelerate multiple applications on FPGAs concurrently but also on CPUs through a common abstraction model. By allowing to transparently execute applications collaboratively on different compute substrates simultaneously, we provide a heterogeneous system with the ability to dynamically scale resources as needed for optimizing system performance and response times. Our results show that resource elastic execution on CPU+FPGA architectures can provide 2× the performance of SDSoC-like platforms for Spector benchmarks. Additionally, for various platform and workload types under varying workload sizes, we measured on average a 20% better makespan and 95% better wait time than standard run-to-completion schedulers without any changes to the applications.

***K*eywords** Resource-elasticity, Collaborative-execution, Heterogeneous, FPGA, OpenCL

1 Introduction

Field Programmable Gate Arrays (FPGAs) are now being deployed at large scale in data centers and the cloud as well as at the edge for IoT devices due to their performance and energy advantage for many applications. This change is primarily driven by the end of Moores law and maturation of High-Level Synthesis (HLS) which makes programming of FPGAs easier for software developers. However, if we look at the surrounding eco-system and execution model, we can find that FPGAs often cannot be shared across multiple applications or adapt to changing workload like software systems, which is one of the primary driving motives in particular for the cloud systems.

Moreover, the problem of underutilization and lack of flexibility is further compounded by the move towards a common platform architecture from industry. An example of this is the Xilinx RunTime system (also called XRT) [1] as shown in Fig. 1, where we can see an embedded system features a hardened ARM CPU on-chip while for data center systems, a soft-CPU is used to program and manage FPGA accelerators on behalf of the host machine. The soft-core on datacenter FPGAs primarily exists to avoids saturating PCIe bandwidth with small control transactions but can also be used to provide other means of abstracting I/O and management for accelerators [2–4]. Hence, regardless of being an embedded system or datacenter setup we can assume that there exists a CPU (hard or soft) on-chip along with the FPGA fabric, capable of accessing memory and of performing control oriented tasks efficiently.

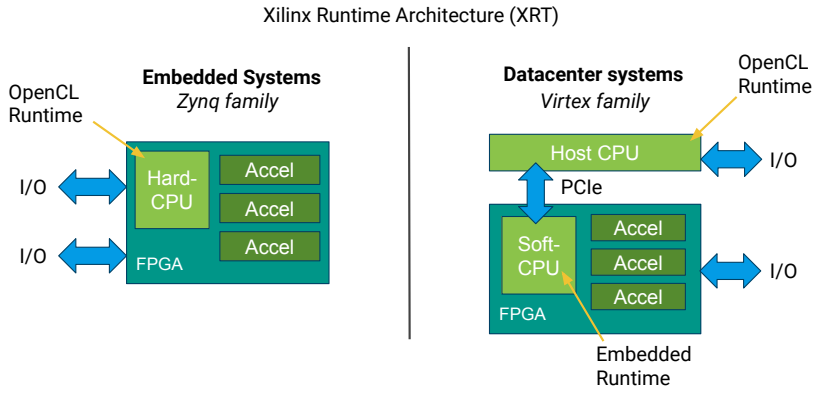

Figure 1: Xilinx RunTime system (XRT) architecture.

However, this CPU is often not used for acceleration and, in cases where it is, the decision of what work it should perform is left to the user. In particular, even advance heterogeneous systems with OpenCL runtimes for *both* CPU and FPGA, the execution is tied to an unchangeable set of resources and is only available for a single application at a time [5]. This highly restricts the potential performance achievable when using heterogeneous resources which can accelerate both control-flow and data-flow type of applications efficiently. To maximize the performance and utilization of such heterogeneous systems, a new paradigm of resource management is required which can support multiple applications while optimizing the four main scheduling problems:

1. *Device type selection*: Which device (or a combination of devices) an application should be executed on (i.e. deciding between CPU or FPGA)?
2. *Workload partitioning*: If scheduling on multiple devices, how much workload should be executed on each device? E.g., scheduling 25% of the threads on CPU and 75% of the threads on FPGA.
3. *Number of compute units*: How many instances of compute units should be allocated for a task? E.g., the number of CPU cores or FPGA accelerators.
4. *Accelerator type selection*: If a device has multiple *implementation alternatives* (e.g., FPGA accelerators with different micro-architectures or big.LITTLE CPU cores), which implementation should be selected?

Note, here and onwards in the paper, the keyword '*device*' refers to either CPU or FPGA on-chip as a whole and the keyword '*compute unit*' refers to an execution unit on a device i.e. CPU cores and accelerators running on an FPGA.

Current runtime systems tend to solve only a subset of these problems for a single application using profiling information (or manual programmer instrumentation) for deciding the appropriate device type, workload partitioning and number of compute units [5, 6]. These systems further refrain from context-switching on FPGA accelerators and treat an FPGA fabric basically again as an ASIC accelerator, ignoring the reconfigurability of FPGAs entirely. The main reason for omitting context switching is because this is a very expensive operation [7, 8] (see Section 2.2 for further details).

Hence, to tackle this, we first propose using cooperative scheduling for FPGAs, where context-switching is only performed at *'consistency points'* when the internal state to store and restore is minimal and all I/O transactions are completed, hence, resulting in very low overhead. Further, we combine this with an abstraction layer to treat both CPUs and FPGA equally with respect to the execution model. With the ability to context-switch on *both* CPU and FPGA, we can consider reallocation of resources at runtime. In particular, it allows us to change the device and accelerator type selection as well as the number of compute units at runtime. When applied with dynamic workload partitioning and heuristics for over-committing resources, we can tackle the four scheduling problems together and enable collaborative execution on all CPU and FPGA resources transparently from the user while maximizing system utilization. We call this ability to dynamically change the resource allocation for a task: *'resource elasticity'*. See Section 2.1 for details.

In this paper, we extend the preliminary work on an FPGA runtime that was presented in [9]. The key extensions in this journal include the 1) concepts and theory behind resource elasticity as well as 2) how corresponding systems can be built and 3) how to orchestrate heterogeneous resources seamlessly on CPU+FPGA platforms for better performance and response time. Moreover, we performed following original contributions in this journal:

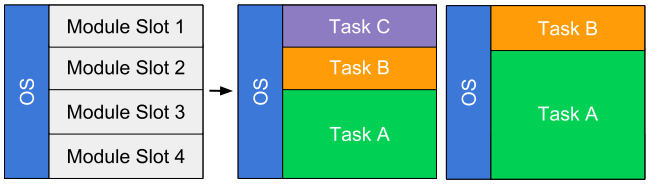

Figure 2: Physical FPGA layout featuring four partial reconfiguration regions (depicted as module slots). Hardware modules may take one or more adjacent slots and the communication is provided through a hardware operating system (OS) infrastructure.

- A detailed review and analysis of concepts and system requirements of resource elasticity on modern CPU+FPGA architectures (Section 2).
- A thorough theoretical problem description of resource elastic scheduling and its implications (Section 3).
- Implementation and integration of the ZUCL shell with heterogeneous resource elastic scheduler for CPU+FPGA architectures (Section 4 and 5).
- Extensive evaluation of a heterogeneous runtime system under varying platforms, workload types and workload sizes to asses scalability and behaviours of scheduling algorithms (Section 6).

2 Concepts

2.1 Resource Elasticity

Let us first consider a software system where multiple tasks run in parallel on a single CPU. In this case, a scheduler would allocate the compute resources in the *time domain* to maximize utilization, i.e. assigning each task a time slot in which it can use the CPU. In contrast to this, an FPGA is a *spatial computing device*, hence, a scheduler must allocate resource in the *spatial domain* to maximize utilization. This implies that we need to be able to allocate reconfigurable resources in a fractional amount to the tasks. We call this fractional block of reconfigurable resources, a slot. A visual representation of the slot is shown in Fig. 2. A scheduler can allocate one or more of these slots on the FPGA to tasks in order to share the FPGA. In a dynamic system, where tasks can arrive and leave at any time, the scheduler needs to be able to reallocate these slots to keep utilization high. To achieve this, the tasks on an FPGA must be, to some degree, '*resource elastic*' in terms of the number of slots it can operate on. We define resource elasticity as follows:

> ***Resource Elasticity:*** *the ability of a task to change its resource allocation transparently from the user. The change in resource allocation may be reflected in the throughput or latency of the accelerator used by the task.*

There are two main ways we can change the reconfigurable resources an accelerator uses: 1) **implementation alternatives** and 2) **replication**. With an implementation alternative, we can swap to another accelerator implementation which performs the same logical operation but with more or fewer resources for more or less throughout / latency of the operation. Note, an implementation alternative may provide *super-linear* performance benefit with respect to its resource requirements due to potentially a better algorithm, bigger local memory for data reuse, loop tiling, unrolling or pipelining. Whereas with replication, we can ideally change the number of instances for a linear change in performance with respect to resources by controlling parallelism.

Given a set of resource elastic accelerators, we can change the amount of reconfigurable resources at runtime to adapt to changing workload as shown in the example scenario in Fig. 3b. As shown in the figure, if we only have a single task executing on the FPGA, we can allocate all the resources to it and, consequently, maximize its performance. Whenever a 'scheduling event' occurs, we can reduce or increase its allocation as appropriate by trading area for performance. By 'scheduling event' here we refer to when 1) a new task arrives, 2) a task finishes, or 3) a context-switching point is reached. Details on context-switching points will be discussed further in Section 2.2. As shown in the example, changing resource allocation also causes some reconfiguration overhead. However, by dynamically growing and shrinking tasks, the scheduler can use free resources whenever available to accelerate the progress of executing tasks for achieving higher utilization and faster execution. Note, the resource reallocation can be performed transparently from the user by using context-switching mechanisms that involve partial reconfiguration of the FPGA.

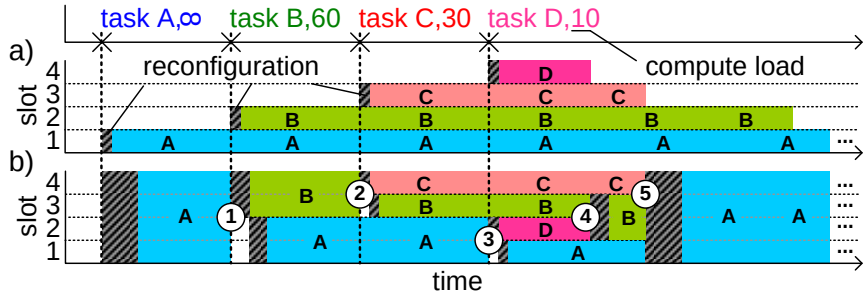

Figure 3: Resource allocation for tasks A, B, C and D in time when using a) Normal fixed module scheduling and b) Resource Elastic scheduling on a 4 slot FPGA. The circled events highlight cases where resources are needed to accommodate new arriving tasks (①, ②, ③) or cases where tasks complete (④, ⑤).

Table 1: Categorisation of context-switching techniques for hardware accelerators based on state storage and restore format.

Context-Switching Technique	Type	Latency	Logic overhead
Configuration Read-back	System-specific	High	High (logic constraints)
Memory DMA	Task-specific	Medium	Medium
Scan Chain	Task-specific	Low	High (lack of BRAMs)
Data Parallelism	Task-specific	Low	Low

2.2 Hardware Context-Switching

In order to reallocate FPGA resources at runtime, we need to be able to perform context-switching of the hardware accelerators. Unlike CPU systems where we can use software instructions to store and restore registers to and from memory, for FPGAs, we have to pause and resume a hardware circuit with potentially arbitrary used storage elements like flip-flops, block RAM, latches and DSP registers. There has been a large body of research dedicated to achieving this efficiently. There are two major research directions aiming at hardware context-switching (as summarized in Table 1): system-specific and task-specific techniques. Each represents a distinct philosophy of implementing context-switching systems. System-specific techniques aim for context-switching to be performed by the system in a pre-defined standard method (typically using configuration read-back technique) while the task-specific techniques aim for letting the task perform the context-switch with its application-specific logic and format (e.g., by adding a scan chain into the accelerator modules). Consequently, the latency of context-switching is fixed for the system-specific category and application dependent for the task-specific category. Note that partial reconfiguration needs to take place in all techniques to change the logic on the fabric, incurring overhead in the range of milliseconds in addition to the application latency for the actual context-switch (saving-and-restoring the internal module state). Hence, keeping the context-switching latency minimal is very important for a dynamic system.

2.2.1 System-specific Mechanisms

The commonly used system-specific technique is configuration read-back. For configuration read-back, the storage elements usable by the accelerators are accessed using the debug mechanisms provided on FPGA chips to read back the FPGA configuration including the state of accelerators. This state can then be extracted and used to resume a module at a later point in time if required [7, 10, 11]. This imposes penalties in the range of many milliseconds and restricts the type of storage elements that can be used for context-switching. For example, Xilinx FPGAs feature block RAM and multipliers (DSP) primitives which can provide pipeline registers. These registers are however not accessible using built-in debug mechanisms of the FPGA.

2.2.2 Task-specific Mechanisms

There are three main task-specific context-switching mechanisms: 1) memory DMA, 2) scan chain, and 3) data parallelism. The memory DMA technique relies on adding additional logic in the accelerator to store and restore its state to memory when requested for context-switching [12]. Thus the latency of context-switching depends directly on the memory transfer required by the accelerator. It also requires additional control logic in the accelerator to be able to perform the transfer, which may become a critical path or make the accelerator resource bound. The second approach is using scan chains where an accelerator is instrumented to include all necessary memory elements in an alternative

Figure 4: 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.

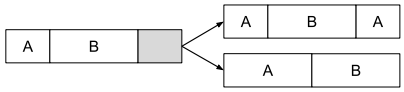

Figure 5: Example scenario where there are two possible alternatives. 1) To replicate A or 2) To perform defragmentation to use different sized module for A.

shift register mode for state access [8, 13]. This imposes the penalty in the range of milliseconds for context-switching (depending on the size of the context).

Overall, the system-specific techniques are cheap and eventually automatic but slow and with some restrictions while task-specific techniques are fast at the cost of extra logic. Suitability of these techniques depends on the particular application and non-functional properties.

The context-switching latency of the above methods can be reduced by minimizing the relevant state space, in particular employing context-switching at only selected execution points (cooperative context-switching), where the internal state space is minimal for an application, which can further improve performance. For example, in a Convolutional Neural Network (CNN), context-switching may only be performed at frame borders where internal line buffer don't carry a state needed in the future. We call these minimal state points *'consistency points'* as they also need to make sure that any pending I/O transactions are handled without any loss of data. Hence, there is a trade-off between scheduling flexibility and overhead for context-switching when selecting between preemptive context-switching and cooperative context-switching.

The third approach of task-specific context-switching is to rely on data parallelism i.e. perform context-switching only at the end of a parallel data chunk. This avoids the need for storing and restoring the state entirely. However, it cannot provide preemptive context-switching capabilities on its own, as the context-switching point is decided based on the execution latency for processing a parallel data chunk. Although, it can be combined with the other context-switching techniques to provide preemptive context-switch if required. Note that data-parallelism is easily found for most common FPGA applications i.e. streaming and batching type FPGA applications. Example of these applications include image processing, neural networks, and data analytics.

Overall for implementing resource elasticity, the techniques like scan chains and configuration read-back cannot be applied directly, as the state encoding is tied to the accelerator design. This does not easily allow a system to change to implementation alternatives without state transformation. Whereas memory DMA and data parallelism allow in most cases directly switching to implementation alternatives, as the accelerator is in control of how storage elements are used *inside* the accelerator. However, without data parallelism, it is not possible to change the number of accelerator instances dynamically. Hence, in this paper and for implementing resource elasticity in general, we use data parallelism as means of context-switching with cooperative scheduling to achieve resource elasticity in both forms (implementation alternative and replication) at minimal context-switching overhead. This can alternatively be combined with memory DMA for applications without data parallelism to improve the scheduling flexibility if needed for a more holistic solution.

2.3 Trade-offs

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

1. Multiple instances vs Different sized modules.
2. Run to completion vs Changing module layout.
3. Collocated change vs Distributed change.

Firstly, at runtime, if we can allocate more resources to a task for better performance, we need to decide whether to do so by replicating an accelerator or by switching to an implementation alternative (Trade-off 1). This has to consider that 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. Fig. 4 shows an example scenario with allocation alternatives which a resource elastic scheduler must choose from.

Secondly, we must decide if it is worth to change the resource allocation for a task given the amount of work left and the partial reconfiguration overhead it may cause (Trade-off 2). This holds regardless if we use reconfiguration for

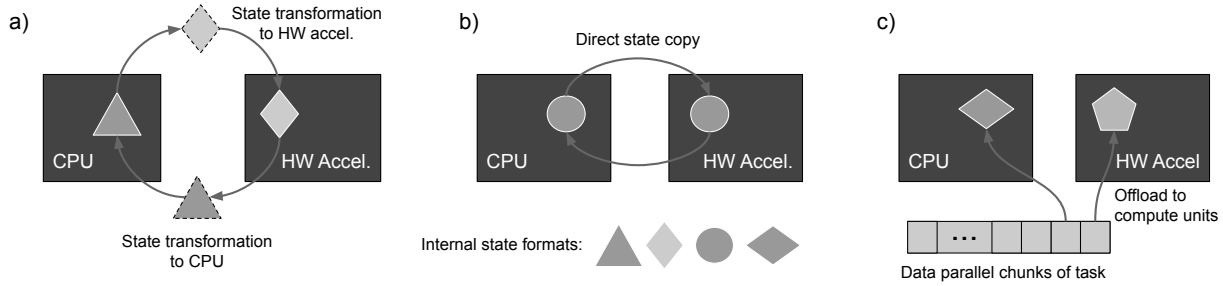

Figure 6: Three ways to perform application migration across device types, where a) depicts method transforming the state at runtime across devices, b) represents technique of common encoding format, and c) shows the data parallelism approach.

shrinking and expansion of accelerators or for defragmenting the FPGA slots. In both cases, a decision must be taken on whether changing the resource allocation to achieve better system-level performance at the cost of performance-sacrifice for certain modules is beneficial or not.

Finally, the decision of selecting a multi-instance option over a different module size may also depend on the location of available free slots and the implementation alternatives available for the tasks (Trade-off 3). For example, consider the scenario as shown in Fig. 5 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 Task A. The complexity of the possible situation increases as we consider multiple distinct tasks where it may be necessary to sacrifice performance for a particular task to accelerate another for achieving higher overall system performance. Further, the objective of the scheduler may not be performance but fairness, energy, or quality of service, in which case, the decision would need be made by scheduling policies accordingly.

At first glance, a slotted architecture as shown in Fig. 2 gives the impression that resource elastic scheduling is similar to the multi-core scheduling known from software systems. However, FPGA scheduling has to consider several very different aspects such that i) an accelerator may occupy multiple adjacent resources (slots) simultaneously (causing fragmentation issues), ii) that accelerators may have implementation alternatives that may work on different problem sizes, and that iii) context-switching is significantly more expensive. This increases the complexity of the scheduling problem when combined with the fact that each implementation alternative has its own performance behaviour.

Moreover, if the available slots are not sufficient to accommodate all tasks even when selecting the smallest implementation alternatives, a resource elastic scheduler can use time division multiplexing (TDM) as 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 than physically available at some reconfiguration penalty.

2.4 Heterogeneous Resource Elasticity

As mentioned in Section 1, there is often a CPU available on-chip along with the FPGA resources, hence, for realizing the full potential of such heterogeneous systems, we need to use resources of both device types (CPU and FPGA) together.

From a conceptual point of view, resource elastic scheduling can also be performed for CPUs by treating it as another form of compute device with its own performance and resource requirements. However, to dynamically move a task from CPU (software version) and to FPGA (hardware version) or vice versa, we need to standardise context-switching policies across different device types. There are three main ways this can be achieved (as shown in Fig. 6):

1. By transforming the device A's state into the device B's state encoding at runtime.
2. Having a common state storage encoding for all device types.
3. Exploiting data parallelism available for application such that no internal state needs to be preserved when switching between devices.

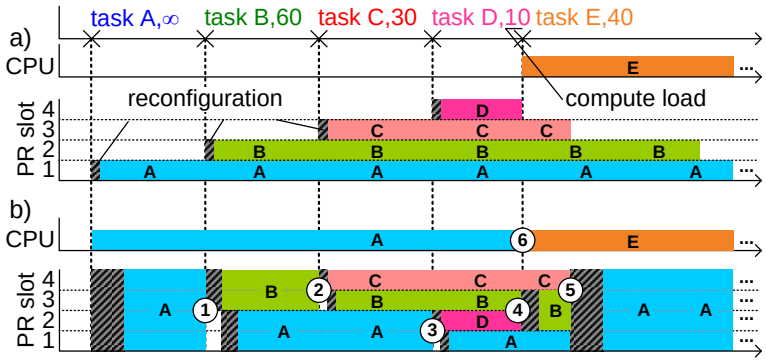

Figure 7: Resource allocation for tasks {A-E} in time when using a) round-robin scheduling and b) heterogeneous resource-elastic scheduling on a CPU+FPGA architecture. The circled events highlight cases where resources accommodate newly arriving tasks (①, ②, ③, ⑥) or cases where tasks complete (④, ⑤).

Transforming the state encoding at runtime can potentially be complex and can impose additional latency even if its performed by a specialised hardware unit [14]. Moreover, it can only be applied to a subset of application, as the runtime system would then need to be aware of differences in state encodings and their transformation. This prevents the migration approach that require state translation to be used for general purpose heterogeneous systems.

To transparently switch between devices it is also essential to abstract the application specification, such that a common state encoding for all device types is used. Hence, in this paper, we adopt this migration method with data-parallelism as our cooperative context-switching mechanism. Note, we can also employ data-parallel migration across devices for performance optimization if needed.

Fig. 7 shows an example execution trace of a system which can migrate tasks across devices in addition to resource elastic scheduling on FPGA. Whereby when a single task is executing on the system, it will use both CPU and FPGA resources for execution. Upon arrival or finish of other tasks, the resource allocation is adjusted without the user realising by using a common context-switching mechanism (e.g., using data parallelism as in our case).

2.5 System Requirements

To build a resource elastic system which can execute workloads on both CPU and FPGA, we need the following four requirements to be met:

1. A common task description model which can compile to both software and hardware, such that the user need not be aware of which implementation is used during execution except for the difference in performance.
2. The software must be multi-thread implementation to be able to scale it to more or less CPU cores.
3. An FPGA shell (static system) infrastructure which can provide three main features to support resource elastic hardware accelerators:
   (a) **Multiple partial regions**: to change resource allocation of tasks without interrupting other concurrent tasks. Also called reconfigurable slots in the previous section.
   (b) **Variable size partial regions**: to support implementation alternatives with different reconfigurable resource requirements.
   (c) **Relocatable modules**: to keep the number of bitstreams minimal. This is because the standard partial reconfiguration flow requires to generate a bitstream per region for each implementation alternative, which can potentially be expensive for multi-tenant systems.
4. A runtime system which can perform resource elastic scheduling, i.e. a system that can dynamically allocate both CPU and FPGA resources based on runtime workload requirements.

With these building blocks, a system is enabled to adapt to varying workload and to move the execution from hardware to software and vice versa, in order to maximize system performance (or other objectives).

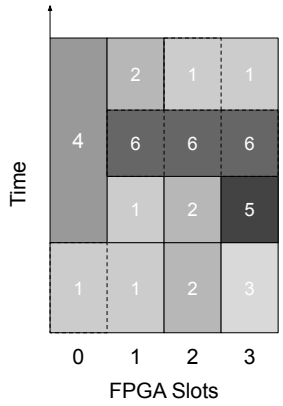

Figure 8: Example of strip packing solution where strips can be rotated or cut using Guillotine cuts.

3 Theory

To optimally solve a scheduling problem, we must first establish a theoretical description of the problem, which also provides a better understanding of its relationship with other scheduling problems and with related real-world problems. The most common approach for describing a spatial scheduling problem in computer science is the strip packing problem. Whereby the tasks are thought of as strips with the x dimension being the compute resources it requires and the y dimension being the latency of the task. The problem then is to pack the strips in a fixed size x dimension (compute resources available in the system), such that the y dimension (time to completion) is minimized. Fig. 8 shows visualization of one such packing example. The strips can be rearranged as well as cut to achieve a denser packing. This models the ability of the task to be preempted or spread out on more or fewer resources using replication. However, for resource elastic scheduling, this problem description does not model all the problem characteristics. The reason is that the tasks can employ implementation alternatives with super-linear performance w.r.t. the resources, implying that *the total area occupied by the strip can shrink in itself*, making the problem more complex. Hence, we need a more general problem description to understand the roots of the scheduling problem. The following Section 3.1 describes the closest related scheduling problem to resource elastic scheduling and Section 3.2 details the changes required to it along with a mathematical problem description that we aim to solve in this paper.

3.1 Relationship to Project Scheduling Problem

The Business and Operation discipline under the project planning field studies a related problem called resource-constrained project scheduling problem (RCPSP) [15, 16]. The problem aims to minimize the total makespan (also called time to completion) and schedules a set of project activities using a fixed amount of resources such that precedence and resource constraints of the project activities are satisfied. This is equivalent to scheduling computing tasks with a run-to-completion model on a computing platform. For practical scenarios, RCPSP alone is a limited model, hence, many variants and extensions of the problem have been proposed to model the practical applications of the problem more accurately. The multi-mode resource-constrained project scheduling problem (MRCPSP) [17], allows each activity to be performed in one out of many several modes, where each mode has its own resource requirements and duration. For example, choosing to perform an activity using automation or manual labour. In the computing world, this would be equivalent to the task being able to execute on multiple device types or number of computing units. Moreover, in real-world projects, many complex circumstances can arise which may require interruptions. To model this, "preemption" was introduced to resource-constrained project scheduling by considering the activities to be composed of parts (sub-activities). In particular, a preemptive MRCPSP problem (P-MRCPSP) allows an activity to be preempted temporarily at any time and resumed again potentially in a different mode with *no additional cost unlike computing platforms*. Thus, allowing the resource consumption of an activity to be changed dynamically in time as required by changing the mode after each part of the activity.

3.2 Resource Elastic Scheduling Problem

Resource elastic scheduling (RES) can be modelled as a project scheduling problem variant with multi-mode and preemption extensions. The modes in which an activity can be performed can be considered equivalent to the implementation alternatives a task has. Similar to different activity modes, each implementation alternative can have different

Table 2: Notation and parameter definitions.

Sets		
V	A set of task supporting cooperative scheduling	
E	A set of arcs representing execution dependencies	
M_i	A set of modes for tasks i (i.e. implementation alternatives)	
Indices		
i, j	Index of tasks	
m	Index of execution modes	
k	Index of resource type	
p	Index of task parts	
Parameters		
n	Number of tasks to schedule	
$0, n+1$	Dummy starting and ending tasks	
U_i	Maximum number of task parts (sub-tasks) for a task i	$i \in V$
$U_i + 1$	Dummy part of task i marking the end of execution	$i \in V$
α_i	Maximum interruption time for task i	$i \in V$
$\|M_i\|$	Number of execution modes for task i	$i \in V$
$r_{imk,p}$	Resource required of type k for part p of task i in mode m	$i \in V;\ m \in M_i;\ p = 1, 2, ..., U_i$
R_k	Maximum resource units available of type k	
$\theta_i(M^p, M^n)$	Switching latency of task i from mode M^p to mode M^n	$i \in V;\ M^p, M^n \in M_i$
d_{im}	Duration of task i in mode m	$i \in V;\ m \in M_i$
ϵ_{im}	Minimum duration for task i in mode m	$i \in V;\ m \in M_i$
Decision variables		
$S_{i,p}$	Positive decision variable: Start time for part p of task i	$i \in V;\ p = 1, 2, ..., U_i$
$F_{i,p}$	Positive decision variable: Finish time for part p of task i	$i \in V;\ p = 1, 2, ..., U_i$
$x_{im,p}$	Binary decision variable: is 1 if part p of task i is executed in mode m; and 0 otherwise.	$i \in V;\ m \in M_i;\ p = 1, 2, ..., U_i + 1$
$t_{im,p}$	Positive continuous decision variable: Duration of part p of task i in mode m	$i \in V;\ m \in M_i;\ p = 1, 2, ..., U_i$

time, area, energy and throughput characteristics. Preemption of an activity is similar to an accelerator context-switch for accelerators with the *additional constraint of given minimum execution time* before the preemption can be performed (i.e. to reach a consistency point with a small state space in an execution flow). After a context-switch, a task can potentially be reloaded onto an FPGA with a different implementation alternative in a similar manner as an activity can change its mode after preemption, except that changing to another implementation alternative imposes a *context-switch overhead* (time penalty) depending on the type of change (e.g. changing the allocation from one instance of an accelerator to two or more). These differences require modifications in standard P-MRCPSP constraint definitions (as revealed in the following Section 3.2.1).

3.2.1 Notation and problem formulation

A resource elastic scheduling problem analogue to the P-MRCPSP problem can be described as a graph $G(V, E)$, where V is the set of nodes representing tasks and E is set of arcs representing execution dependencies. Dummy tasks 0 and $n + 1$ act as the beginning and end of the schedule. Each task in $i \in V$ is made up of parts (sub-tasks) p which can be executed in mode $m \in M_i$ where a mode m has its own time (d_{im}) and resource requirements ($r_{imk,p}$). Dummy tasks $i = 0$ and $i = n + 1$ have zero duration and resource requirements. For resource k, the availability of R_k is constant and is constrained throughout the scheduling problem. Cooperative preemption of tasks incurs a time penalty (O_i) depending on the overhead associated with the change from mode M^p to M^n for the tasks. The maximum number of context-switches allowed is U_i, which is the maximum number of parts of a task. For each task i, a minimum execution time ϵ_{im} is defined during which a task i in mode m cannot be interrupted. Mode M^0 and M^{U_i+1} represent the uninitialized state of the resources (i.e. no resource allocation). Table 2 summaries the notation and parameter definitions used by this formulation.

The following mixed integer non-linear programming formulation models the resource elastic scheduling as a variant of the P-MRCPSP problem with context-switching overhead:

$$\text{Min } Time = S_{n+1,0} \tag{1}$$

The relation (1) describes the minimization of the makespan as an objective. Such that:

$$\sum_{m=1}^{|M_i|} x_{im,p} = 1; \quad \forall i \in V;\ p = 0, 1, ..., U_i \tag{2}$$

Constraint (2) enforces that part p of task i can only exist in one mode at a given time.

$$\epsilon_{im} \leq t_{im,p} \leq d_{im}; \quad \forall i \in V;\ p = 0, 1, ..., U_i + 1 \tag{3}$$

Constraint (3) models that there is a minimum execution time ϵ_{im} and a maximum execution time d_{im} during which the cooperative task i in mode m is in progress without interruption.

$$\sum_{p=1}^{U_i} \sum_{m=1}^{|M_i|} \left(\frac{t_{im,p}}{d_{im}} \right) x_{im,p} = 1; \quad \forall i \in V \tag{4}$$

Constraint (4) ensures that the sum of the relative progress of all parts of a given cooperative task in all execution modes is equal to one unit task, i.e. each task executes entirely.

$$F_{i,p} - S_{i,p} = \sum_{y=1}^{|M_i|} \sum_{m=1}^{|M_i|} \Big(x_{im,p} t_{im,p} + O_i(yx_{iy,p}, yx_{iy,p}, p) \Big)$$
$$\forall i \in V;\ p = 0, 1, ..., U_i + 1 \tag{5}$$

Constraint (5) checks that the duration of the part p of task i in mode m should be equal to the difference between the finish and start time for part p of task i plus the penalty of changing the mode if any. It models the execution latency of the task plus the overhead imposed by context-switching to another implementation of the task.

$$F_{i,U_i} \leq S_{j,0}; \quad \forall (i,j) \in E \tag{6}$$

Constraint (6) ensures that the earliest start time of task j is forbidden to be smaller than the finish time of its predecessor task i.

$$\sum_{i=p_t} \sum_{m=1}^{|M_i|} x_{im,p} r_{imk,p} \leq R_k \quad p = 1, 2, ..., U_i;\ k = 1, 2, ...k$$
$$p_t = \{ i \in V \mid S_{i,p} < \ t \ \leq S_{i,p+1} \}; \tag{7}$$

Constraint (7) enforces resource limit of R_k, for each time instant t, and for each resource type k.

$$O_i(M^p, M^n, p) = \begin{cases} 0, & \text{if } M^n = M^0 \text{ or } p = 0. \\ \theta_i(M^p, M^n), & \text{otherwise.} \end{cases} \tag{8}$$

Equation (8) defines the overhead (e.g., for FPGA reconfiguration) of changing from mode M^p to mode M^n for part p of task i.

$$F_0, S_0 = 0 \tag{9}$$

Constraint (9) establishes that there exists F_0 and S_0 as dummy parts to simplify the formulation. Note that these dummy parts are defined to be of zero latency in Table 2 and, hence, does not affect the actual solution space.

$$F_{i,p}, S_{i,p} \geq 0; \quad \forall i \in V;\ p = 0, 1, ..., U_i + 1 \tag{10}$$

$$x_{im,p} \in \{0, 1\} \quad \forall i, m, p \tag{11}$$

Constraints (10) and (11) guarantee the start and finish time are positive and decision variable is binary, respectively.

The mapping of heterogeneous resource elastic scheduling to the above problem formulation can be found in Appendix A.

3.3 Discussion

The resource-constrained project scheduling based formulation in Section 3.2.1 shows that the RES problem shares its root with scheduling problems from the business and operation discipline and that it can be considered theoretically equivalent to the project scheduling for software team or large scale government projects [15, 16].

However, since the RES problem is a variant of resource-constrained project scheduling problem and more complex than strip packing problem, it is an NP-hard problem [18, 19]. Implying there only exists algorithm with exponential complexity for its solution, unless $P = NP$. The common approach to solve a mixed integer non-linear programming formulation like the on presented in Section 3.2.1, is to use a Branch and Bound technique with a combination of standard nonlinear optimization methods. Whereby the technique begins by recursively breaking down the problem into multiple ILP sub-problems until a solution that is satisfying all integer constraints is found. Overall, finding a solution using this method can take anywhere between milliseconds to days depending on the complexity of the problem instance.

Consequently, for a dynamic system where tasks can arrive at any time without prior notice, this exploration needs to take place at runtime which makes it infeasible for our purposes. Hence, there is a need for domain-specific heuristics to simplify the problem and solving it as close to the optimal solution as possible. To this end, we propose solving the problem in snapshots i.e. allocating the resources given the runtime conditions at the scheduling event and not looking into the future for an entire scheduling trace. This is because computing the schedule ahead of time may become obsolete if a new task can arrive at any time. Overall, this simplifies the problem into a variant of the bin-packing problem by removing the time dimension and not considering dependent waiting tasks. We can solve the resultant bin-packing problem optimally using standard Branch and Bound technique. However, to find an approximate solution for makespan optimization of the RES problem we still need to link these individual snapshot solutions. To achieve this we can use domain-specific heuristics to estimate the resulting makespan latencies at each step. In Section 5.2 we provide an implementation of one such Branch and Bound technique with the snapshot heuristic and ranking functions (domain-specific heuristics) to solve the RES problem at runtime.

4 Implementation: Base System Infrastructure

As mentioned earlier in Section 2.2 and 2.4, in this paper, we utilize a data parallelism based cooperative approach for treating all the compute resources under a unified execution model. This simplifies the scheduling problem for compute engines with different latencies and resource requirements which provide the same operations.

Given that OpenCL is a widely accepted industry standard for programming data-parallel applications on heterogeneous platforms (i.e. CPUs, GPUs, FPGAs and ASICs), we target OpenCL modelled applications for dynamic runtime scheduling with the extension of sharing all devices transparently. This not only resolves the portability issues across devices but also raises the abstraction to the OpenCL execution model, hiding the complexity of integrating heterogeneous compute resources (i.e. CPU and FPGA in our case). The following Section 4.1 describes the OpenCL execution model in detail.

To support OpenCL execution on CPU cores, we use the POCL runtime [20] and, for FPGA systems, we use the ZUCL shell [21]. The compilation for OpenCL to software execution on CPU is as per the standard LLVM compilation flow (details of which can be found in [20]). However, for the FPGA system, as we have to satisfy special requirements mentioned in Section 2.5, we detail the flow and system specification in Section 4.2.

4.1 OpenCL Execution Model

An OpenCL application has two components: a *host program* and *kernels*. A host program manages memory objects and issues execution commands while executing on a host machine, whereas kernels are compute-heavy functions

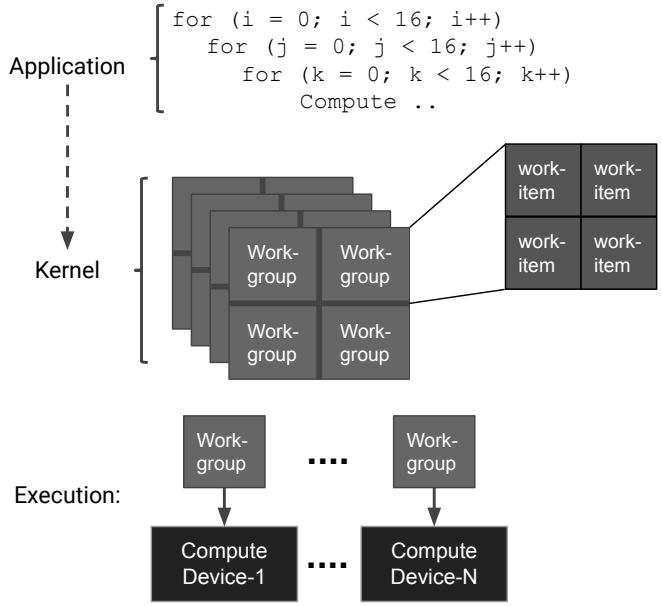

Figure 9: Structure and execution of OpenCL kernels on computer devices.

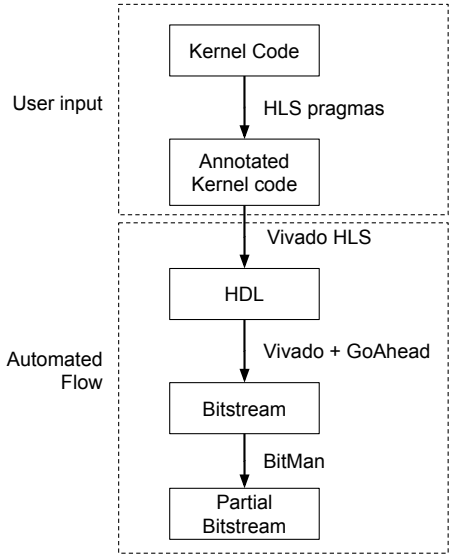

Figure 10: Design flow for relocatable accelerator generation from OpenCL kernels.

Table 3: Resources available for each partial region allocations.

Resource Type	1-Slot	2-Slot	3-Slot	4-Slot
CLB LUTs	32640	65280	97920	130560
BRAM Tiles	108	216	324	432
DSPs	336	672	1008	1344

which are executed on an accelerator's compute unit. When a kernel execution command is issued by the host program, an abstract index space known as *NDRange* is generated. A kernel is executed once for every point in this NDRange index, which is known as a *work-item*. Work-items are then grouped together for execution on a computing unit, this group of work-items is called a *work-group*. Fig. 9 shows the visual representation of the kernel. It provides a coarse decomposition of NDRange and allows work-items to share and synchronize data using barriers on local memory. However, there is no execution order defined between work-groups by the OpenCL standard which allows them to be executed concurrently on different compute units for high performance [22]. At the end of the work-group execution, the result is written back to the global memory without any synchronization.

Since there is no execution order defined and all the state information is stored in memory at the end of work-groups, we can perform a data parallelism context-switch for OpenCL applications cooperatively as mentioned in Section 2.2. With this, we can *at each end of the work-group* change to another implementation alternative, pause the execution and resume later, or run the kernel concurrently on multiple hardware accelerators as well as CPUs.

4.2 ZUCL: Base Shell Design

The ZUCL shell [21] supports the ZCU102 board featuring a Zynq Ultrascale+ FPGA. This FPGA integrates a quad-core ARM Cortex-A53 with a reconfigurable fabric. The shell provides 50% of the reconfigurable resources as four partial regions to host user accelerators where each partial region has its own clock and AXI master and slave interface of varying data-width (32, 64, and 128) (see Fig. 11b). Table 3 shows the resources available per region. These partial regions can also be combined to host larger accelerators as shown in Fig. 2 as well as partial bitstreams can be relocated to other partial regions. Thus, meeting our basic FPGA system requirements (see Section 2.5). The accelerators for the shell can be compiled from both RTL and HLS (C, C++, and OpenCL) allowing us to adopt the OpenCL standard and all types of context-switching mechanisms. The relocatable nature of the accelerators allows them to be developed independently from the static system, i.e. a change in the static system does not require the accelerators for it to be recompiled as long as the interface between them is unchanged. Details of the compilation flow are presented in Section 4.2.1.

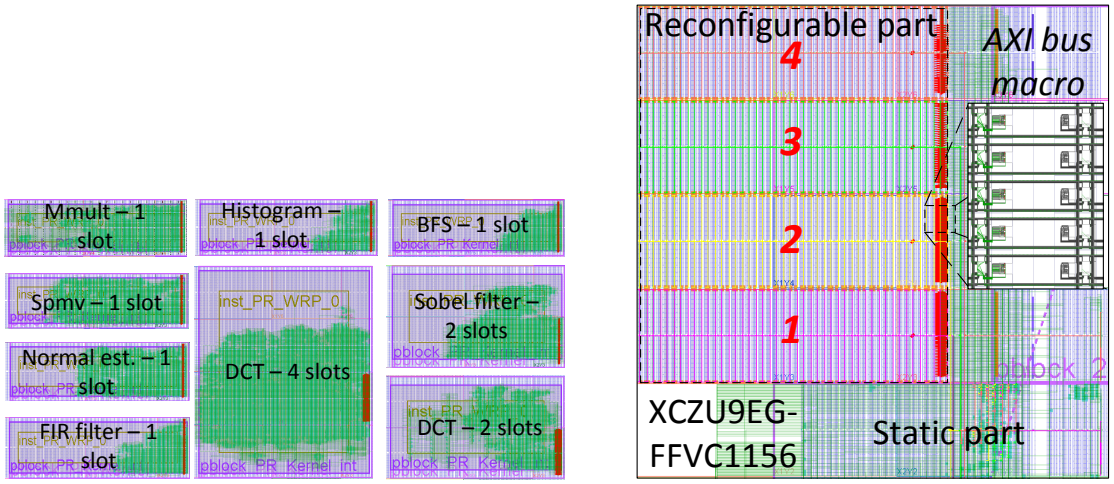

Figure 11: The physical implementation of the system on Zynq UltraScale+ (ZCU102) where a) shows the accelerators from Spector Benchmark [23] and b) shows the static system with neighbouring partial regions (slots) based on the ZUCL framework [21].

Additionally, to aid the development of the software system on top of it, it supports Xilinx PetaLinux environment on the ARM core with various root file systems (Ubuntu, Debian, and PetaLinux rootfs). This ability was used to build a custom runtime on top of Ubuntu (see Section 5 and 6.1). This runtime provides the same API as the SDSoC framework available by the FPGA vendor Xilinx but our system transparently adds the heterogeneous resource management feature to it.

4.2.1 HLS Compilation Flow

To generate *relocatable accelerators* for migration within an FPGA fabric, ZUCL uses the design flow shown in Fig. 10. It takes the user-level OpenCL code as an input and translates it into partial bitstreams without further user intervention. The resulting design satisfies the following key design rules to support module relocatability:

1. Identical top-level communication interfaces must be used in terms of bus protocols and the number and position of interfaces signal wires generated by Vivado HLS for the individual OpenCL accelerators.

2. All physical resources of an accelerator must stay in a pre-defined bounding box, including routing wires. No routing violation to surrounding regions is allowed.

The above conditions are guaranteed by utilizing a custom Partial Reconfiguration (PR) methodology [21] instead of following the default Xilinx PR flow [24]. In this custom PR flow, required place-and-route constraints are transformed to TCL scripts with automatic assistance from the GoAhead tool [25]. These TCL scripts are then applied to Vivado for the accelerator's physical implementation. To use the same accelerator bitstream at different positions on the chip, the bitstream manipulation tool and API, BitMan [26] is used to generate partial bitstreams at design time and to relocate accelerators at runtime.

Fig. 11 displays the resulting static system and 8 implemented relocatable accelerators from the Spector benchmark suite [23]: Sobel filter, Sparse Matrix-Vector Multiplication (SPMV), 3D-distance Normal Estimation, Finite Impulse Response (FIR) filter, Histogram, Matrix Multiplication (Mmult), Breadth First Search (BFS) and Discrete Cosine Transform (DCT).

Note, this compilation approach is portable to PCIe based platforms as well [27] and expresses that resource elasticity can potentially be used on datacenter FPGAs. Moreover, partially reconfigurable modules can be automatically build from any module specification supported by Vivado including RTL and C++. These modules can then be used with resource elastic scheduler if they satisfy the requirements listed in Section 2.5.

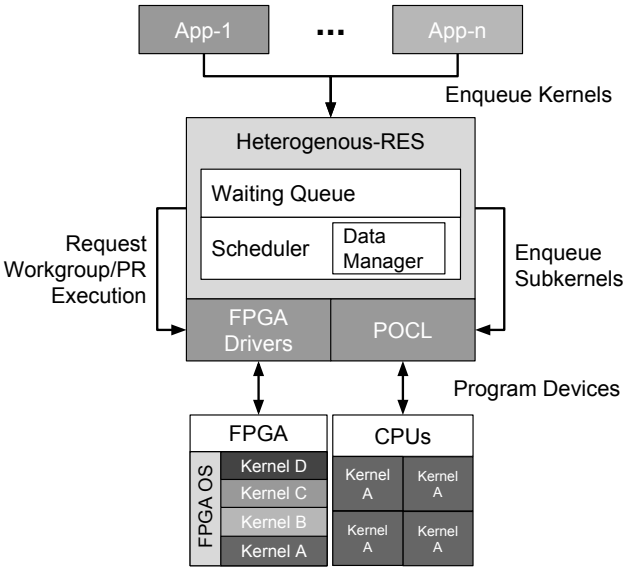

Figure 12: The complete execution stack of our runtime system as implemented on the case study platform (ZCU102).

5 Implementation: Runtime System

To perform resource elastic scheduling on our base infrastructure, we also need a runtime system which can take care of managing multiple user applications and reallocate resources without exposing low level device details to the users. The following Section 5.1 describes the system overview on how various components in the system integrate with each other as well as the view exposed to the user. Lastly, Section 5.2 describes the resource elastic scheduling mechanism based on Branch and Bound exploration with heuristics.

5.1 System Overview

Our system runs as a daemon to which multiple different applications can submit kernel execution requests, as shown in Fig. 12. It contains a waiting queue and scheduler which is responsible for executing kernels on FPGA accelerators and submitting sub-kernels (a set of kernel work-groups) to the POCL [20] runtime for CPU execution. However, unlike standard OpenCL runtime systems, we submit work to devices at fine granularity (work-groups) allowing the devices to be *reallocated* more often and being able of *interleaving* the execution of multiple kernels on the same device as necessary. This approach does not incur major overheads because the native kernel execution is often implemented at work-group granularity internally in the runtime systems anyway. For example, for FPGAs, the accelerator is synthesized for execution of a single work-group while for CPUs the kernel is compiled into a work-group function, which is then executed repeatedly for complete kernel execution [28].

At the end of each work-group execution for OpenCL FPGA accelerators, there is no internal state which needs to be stored, resulting in a natural consistency point. This allows the runtime to perform context-switching either by simply dispatching a work-group from another application or by replacing the accelerator for a new application using Partial Reconfiguration (PR). Given the context-switching ability for both device types subjected to cooperative scheduling, our runtime scheduler is designed to perform dynamic reallocation of resources for performance optimization and relieve the programmer from the responsibility of *workload partitioning* as well as supporting the *execution of multiple applications*.

5.1.1 User view

The complete design flow used for the development and execution of the OpenCL kernel with resource elasticity is shown in Fig. 13. The input required by the programmer is an OpenCL kernel with optionally different optimization levels and host code. There are two differences compared to the standard OpenCL flow [20, 29]. 1) The generation of different kernel versions (e.g., design alternatives with different resource-performance trade-offs) from the user viewpoint, which can be performed easily by selecting the relevant HLS pragmas options. 2) A generic OpenCL call

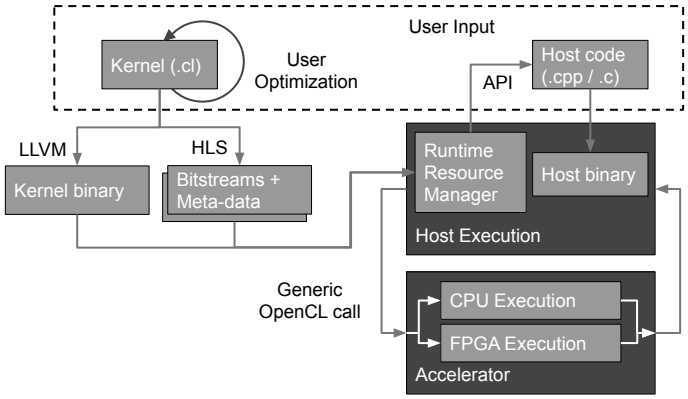

Figure 13: Design flow for the OpenCL kernel development (design-time) and execution (runtime) with resource elasticity.

is made by the host program rather than device specific call. This is because the runtime takes the decision on behalf of the user for selecting the appropriate device type (i.e. to run the kernel on CPU or FPGA or both).

Note that it is not a requirement that each module is implemented with different resource and performance variants as we also allow implementing resource elasticity by instantiating an accelerator multiple times. Our scheduler (see Section 5.2) can arbitrarily handle multiple accelerator instances of different size as well as multiple CPUs.

5.2 Scheduler

Given that the scheduling problem is a mixed integer nonlinear problem (Section 3.2.1), we use a Branch and Bound method with snapshot heuristic to explore the resource allocations where ranking functions (domain-specific heuristics) guide the resource management. The overall core of the algorithm can be categorized into three different steps (as shown in Fig. 14): kernel selection, resource allocation and resource binding. In summary, the algorithm first identifies a set of kernels to schedule together on the available resources. It then explores the potential resource allocations for the kernels while ranking each resource allocation based on a ranking function (e.g., predicted performance, fairness, energy, QoS or throughput requirements) with consideration of overhead caused by the new resource allocation. The overhead can include the cost of PR, data movement latency and software overhead. Thereafter once the best possible resource allocation is identified based on a ranking function, it performs the steps necessary to change current resource allocation by performing PR (if needed) and programming of the devices (e.g., FPGA accelerators and CPUs) for kernel execution, as detailed below.

5.2.1 Kernel Selection

To keep waiting times for tasks minimal and ensure kernels continue to maintain progress, we select the maximum number of kernels from the waiting queue. Allowing the system to degrade the performance gracefully while overcommitting resources. To implement this heuristic, we first relinquish control of as many FPGA accelerators and CPUs as possible (if any) and add them to the back of the waiting queue to free up resources for waiting kernels. At this stage, there may be kernels which have not reached their consistency points (i.e. currently executing). Hence, we select these kernels in-flight in addition to the maximum kernels that we can assign from the waiting queue using the newly recovered resources.

5.2.2 Resource Allocation

Given a set of independent kernels (K) to execute, we first identify the current resource pool allocation (c), see Fig. 15. A resource pool allocation is a collection of allocated resources from the heterogeneous system, i.e. the number of slots (PR regions) on FPGA and CPU cores (see Fig. 15). Then we perform exploration of the potential resource pool allocation transitions from i to a new resource allocation configurations using a Branch and Bound algorithm. The objective function (f) of the algorithm is calculated using Equation (13) which models the makespan of kernels based on a combined rate of execution for CPUs and FPGA accelerators. Where $T_c(r_k)$ denotes the projected time taken to execute remaining work-items ($W_l(k)$) on resource allocation for kernel r_k in resource pool allocation r given work-item latency $T_w^F(r_k)$ for FPGA allocation and $T_w^C(r_k)$ for CPU allocation. The transition cost from resource pool allocation c to r is captured by $O_i(c, r)$, which refers only to PR cost of the kernel in our system because CPU

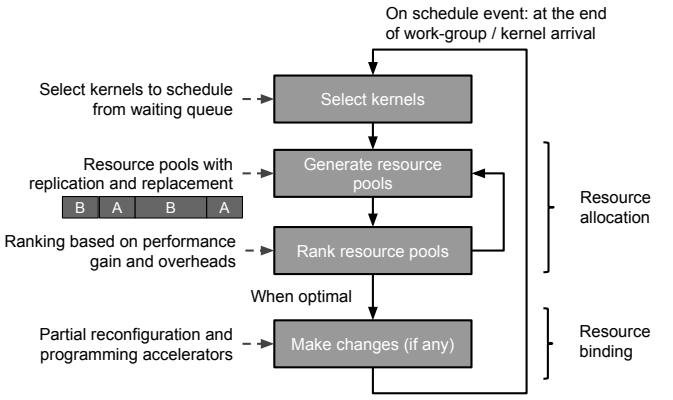

Figure 14: Structural overview of our heterogeneous resource elastic scheduler.

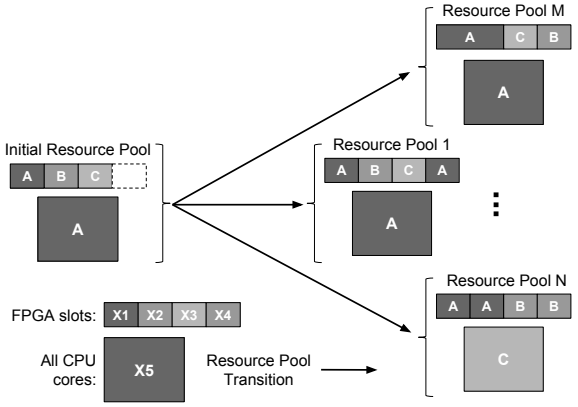

Figure 15: Resource pool allocation transition examples upon exit of the kernel hosted in the last FPGA slot (PR region).

cores and FPGA slots share the same memory in our target platform (ZCU102), hence removing the need for data movement between private device memory and system RAM.

$$T_c(r_k) = \frac{T_w^F(r_k) \times T_w^C(r_k)}{T_w^F(r_k) + T_w^C(r_k)} \times W_l(k) \tag{12}$$

$$\arg\max f(r) = \{T_c(r_k) + O_i(c, r) \mid i \in K\} \tag{13}$$

The upper bound (g) is calculated using Equation (15) where $f(r)$ denotes the execution latency of currently allocated resources, function $T^m(K_l)$ denotes the maximum latency of kernels yet to be assigned to resources (K_l) given their worst case latency ($T_m(k)$), number of free FPGA slots s and PR latency per slot P_l. Note, as we use the worst case latency of kernels in addition to worst-case overhead, we achieve an upper bound on makespan latency for resource pool allocation r. Execution latencies required for these calculations are based on the profiling information provided with the kernel implementation alternatives, hence, they only serve as a heuristics for problem-solving and are not meant to be absolute. In particular, these heuristics allow estimating the solution of sub-problems and help reducing the runtime of the algorithm while keeping deviation from the optimal solution minimal by using profiling information.

$$\arg\max T^m(K_l) = \{T_m(k) + s \times P_l \mid k \in K_l\} \tag{14}$$

$$g(r) = max(f(r), T^m(K_l)) \tag{15}$$

The solver generates decision branches starting with the FPGA whereby we create a branch for each implementation alternative of the requested accelerators available to the first fit location, i.e. we start by branching b number of times where b is the total number of bitstreams of a given kernel K. Intuitively, it can be imagined as filling up the FPGA from one end to the other end one step at a time at each level of exploration tree (see Fig. 16). Once, the FPGA cannot accommodate new modules our algorithm starts allocating CPUs and branches with a factor of $|K|$, i.e. CPU allocation for each kernel. However, the total number of branches are subjected to three primary constraints which help in reducing the search space along with the upper bound for recomputing a schedule (g):

i. *Availability* of only certain sized FPGA accelerators rather than all implementation alternatives (e.g., a two slot FPGA accelerator is infeasible when only one slot is free on an FPGA).

ii. *Runtime constraints*: executing kernels cannot relocate, which may fragment available free resources.

iii. *Over allocation*: we cannot allocate more resources than our heterogeneous system can provide.

5.2.3 Resource Binding

Given the new resource allocation, we identify necessary changes to the FPGA allocation and queue the request to perform PR through FPGA drivers. Thereafter, we program the CPUs by launching sub-kernels via the POCL runtime

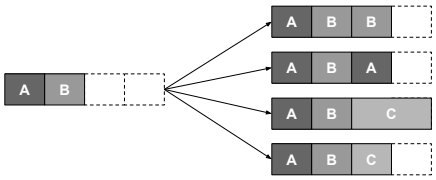

Figure 16: Branching cases for our FPGA Branch and Bound allocation algorithm.

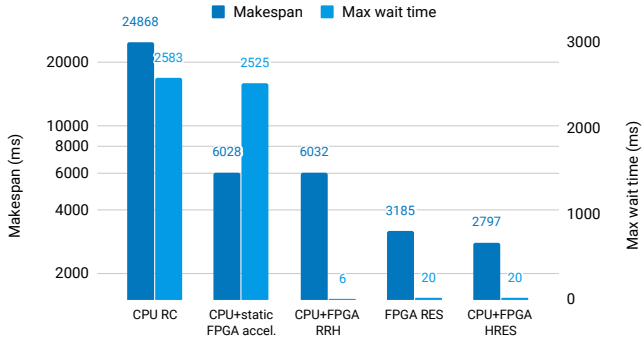

Table 4: Wait times and the total makespan (in milliseconds) for the Spector benchmarks used in the case study.

Benchmark	CPU	1-slot	2-slot	3-slot	4-slot
Normal est.	4x	**1x**	2x	3x	4x
FIR	**1x**	3.4x	6.8x	10x	13.6x
DCT	1.18x	-	**1x**	-	3.54x
MMult	**1x**	3.8x	7.6x	11.4x	15x

Table 5: Relative performance of accelerator benchmarks on various resources.

and the FPGA accelerators via our FPGA drivers. The data manager keeps track of the next work-group which needs to be scheduled and collects the data back from the accelerators and POCL. Note that our system performs dynamic scheduling of work-groups across different devices rather than static partitioning of kernel workload (work-groups). Further, this reduces the dependence on the accurate profiling information about execution latency as a faster execution unit will automatically receive a larger share of the workload based on the input data characteristics, thus improving performance without user intervention.

6 Implementation: Evaluation

We perform a two-stage evaluation, first, we undertake a case study with real-world workloads using the Spector benchmark to demonstrate the feasibility and practical performance benefits achievable with resource elasticity (see Section 6.1). In addition, we carry out event-based simulation experiments (accurate to milliseconds) with varying workload sizes, workload types and compute resources to assess the scalability and behaviours of the schedulers in worst case scenarios (see Section 6.2).

6.1 Case study

We evaluate heterogeneous resource-elasticity with real-world workloads on the ZUCL base infrastructure (described in Section 4.2) where each slot requires 3 ms for partial reconfiguration. The runtime (see Section 5) is combined with POCL version 1.2 [20] and Ubuntu 16.04.5 LTS based on the Xilinx PetaLinux 2018.2 kernel. We ran POCL in standard configuration with automatic vectorization enabled in LLVM, which maps the work-items operations to NEON vector instructions whenever possible. However, at this point, POCL does not support the separation of compute units as individual devices for the platform and hence runs the OpenCL kernel on all 4 cores concurrently. Consequently, we treat the system as a single logical CPU, together with a 4-slot FPGA.

For evaluation we select four OpenCL accelerators from Spector benchmark suite [23] with varying device type preferences: Time domain FIR (FIR), Discrete Cosine Transformation (DCT), 3D Normal estimation (Normal est.) and Matrix-Matrix multiplication (MMult). The results of which are shown in Fig. 11a and Table 5, where Normal est. is CPU favoured and has a 4× performance benefit over its FPGA implementation, FIR is FPGA favoured with 3.4× performance benefit over CPU, DCT on CPU is faster by 1.18× compared to 2-slot FPGA accelerator and 3× slower than a 4-slot FPGA accelerator and MMult is 3.8× faster on the FPGA compared to CPU only. The implementation difference between 2-slot or 4-slot DCT modules is based on larger buffer sizes and loop unrolling for better data reuse. In particular, DCT serves as an interesting edge case because, depending on the version which a scheduler con-

siders, it may be CPU favoured or FPGA favoured. Note that we compiled the benchmarks unmodified with default compilation settings and that our approach does not require any changes to the OpenCL code used. If target-specific optimization substantially changes the cost model, our runtime system would automatically incorporate that.

For simplicity, our case study schedule consists of the following arrival times: Normal est. at 0ms, DCT at 100ms, FIR at 200 ms and MMult at 300ms. We consider 5 main execution scenarios: 1) run kernels only on CPUs using POCL using a standard run-to-completion model (RC); 2) run kernels in run-to-completion mode on CPU using POCL and on static FPGA accelerator (similar to SDSoC [30] using FPGA drivers) where the FPGA accelerator is MMult (biggest kernel); 3) run kernels on CPU and FPGA using Round-Robin with Heuristic (RR-H); 4) run kernels only on FPGA using RES and 5) run kernels on CPU and FPGA using heterogeneous RES (HRES). The resulting wait times and makespans are shown in Fig. 4. We can see that the introduction of FPGA acceleration improves the makespan considerably as MMult gets to execute on its preferred execution unit. Further, the CPU+static FPGA accelerator system performs similar to RR-H but results in very long wait times as it obeys its run-to-completion model. Overall, speedups of 1.89× and 2.1× with 126× improvement in max waiting time over SDSoC-like systems are observed for FPGA with RES and HRES respectively. Moreover, compared to RES on FPGA only, its heterogeneous implementation achieves a 13% better makespan using the same platform and accelerators.

6.2 Simulation Experiments

6.2.1 Workload Characterization

We tested the system with varying workload, whereby compute-bound kernels arrive at the rate of (1, 5, 10, 15, and 20) kernels per seconds for 100 seconds based on a Poisson distribution, i.e. on average each schedule is made up of (100, 500, 1000, 1500, and 2000) kernels respectively. Additionally, each scheduling scenario is sampled 1000 times for error range and significance.

To model a heterogeneous workload where certain applications run faster on CPU while others run faster on FPGA, kernels are split into two categories: CPU favoured and FPGA favoured. The ratio of CPU favoured to FPGA favoured is assessed by three variants: 25% CPU - 75% FPGA, 50% CPU - 50% FPGA, 75% CPU - 25% FPGA. We run these kernels on platforms with CPUs ranging from 0 to 4 and FPGA slots ranging from 0 to 8.

Characteristics of each kernel are generated based on a uniform random distribution, i.e. base latencies of work-groups are in the range of [20, 100] milliseconds, arrival times are set randomly in the Poisson time interval, the slot size of each kernel bitstream and the number of bitstreams is defined independently in the range of [1, min(4, n-1)] where n is the number of available FPGA slots. It further receives a speed-up on base latency in the range of [2, 16] if the bitstream size is greater than one or if the kernel is CPU favoured. The total number of work-groups ranges in [10, 1000] to model a wide variety of short and long-running kernels. We assume that the I/O requirements of the kernels are not on the critical path of the application. The PR latency is modelled linearly with respect to slot size, where each slot requires on average 3 milliseconds (as found in Section 6.1). Note, this implies that there could be kernels where the PR latency can be as much as 9.6× the work-group execution latency as well as kernels where PR overhead is negligible. Restricting the maximum number of slots per bitstream allows us to investigate how the scheduling behaviour changes when a larger FPGA is available for the same workload (i.e. FPGAs with 6 and 8 slots).

6.2.2 Schedulers

We execute these randomly generated scheduling requirements on five different scheduling algorithms on the *same platforms* with both CPU and FPGA. Four of which serve as a baseline for our experiments:

1. Run-to-completion (RC), which is the standard for OpenCL runtime systems but with multiple partial regions (PR) for FPGA (e.g. SDSoC with multiple PR regions).
2. Run-to-completion with heuristic (RC-H) which selects the device based on the profiling information whenever both types of resources are available for execution (CPU and FPGA).
3. Round robin scheduling policy (RR) using the context-switch mechanism discussed in Section 5. Allowing the PR region to be reallocated at the end of each work-group to kernels from the waiting queue.
4. Round robin with heuristic (RR-H), couples standard round-robin policy with a heuristic to select the favoured execution unit when there is a choice between device type (CPU and FPGA).

Whereas, (5) heterogeneous resource-elastic scheduler (HRES) follows our implementation as described in Section 5.2. Note that the results of performing resource elastic scheduling for only a single device type (either CPU or FPGA) are captured in platforms with no CPUs or no FPGA slots.

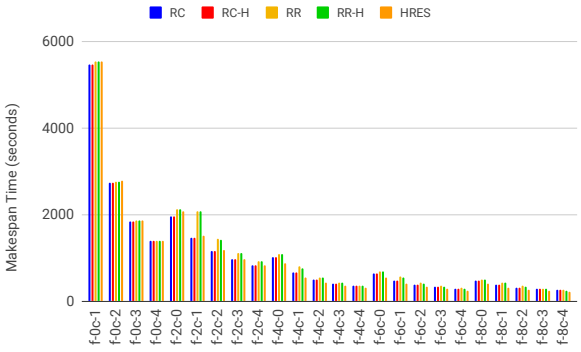

Figure 17: Avg. makespan time for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 25:75 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

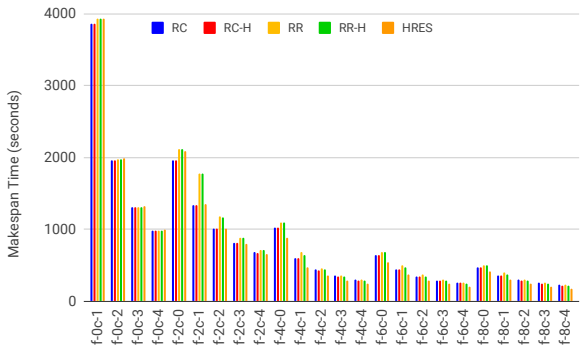

Figure 18: Avg. makespan time for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

Overall, our simulation experiments only change execution models (as above) and keep the platform capabilities and workload the same for evaluating the impact of schedulers on system performance and overheads.

6.2.3 Results

Fig. 17, 18 and 19 show the makespan (time to completion) latency of schedulers with kernels arriving at the rate of 5 per second with 25%, 50% and 75% of being CPU favoured. We can see that adding more resources to the execution platform allows the kernels to be executed in parallel and reducing the makespan as we would expect. When the workload is skewed and is made to execute completely on non-favoured device type (e.g. FPGA favoured kernels on CPU) the execution latency increases substantially. However, adding the smallest instance of the preferred device types helps reduce the latency considerably.

If we look at the relative performance of HRES compared to the run-to-completion scheduler on varying CPU-to-FPGA favour ratios in Fig. 20, we can see that overall when averaged across all workload sizes, the general trend is similar. When the number of resources available is small, HRES does not show any particular benefit and in fact, an overhead of 6% (data point f-2c-0) in the worst case. This quickly reduces into 20% performance improvement on average as we add more resources to the system, which allows HRES to dynamically allocate free resources for task acceleration. The benefit of around 15 to 20% improvement is maintained with more resources being added even when the workload is increased by 20×, this shows that the benefit of HRES over run-to-completion model would scale linearly with workload amount and workload types. Fig. 21, 22 and 23 show a detailed breakdown based on kernel arrival rate for varying CPU-to-FPGA favoured ratio. The exact latency behaviour changes minorly based on the ratio of CPU-to-FPGA preference of the workload for kernel arrival rates above 1. The low arrival rate of 1 shows a deviation on how quickly the performance changes with an increase in resources. This is because lower workload allows more opportunities for collaborative execution (i.e. executing a kernel on both CPU and FPGA), hence, improving performance rapidly.

We measure wait time as the difference between the arrival time of a task and the time this task begins execution (including PR latency). We can see from Fig. 24 that on average the wait time is reduced by 95% when using HRES compared to the run-to-completion model. The exceptions to this are pure FPGA platforms and highly CPU favoured workload. Pure FPGA platforms force the scheduler to wait for partial reconfiguration before starting the execution, hence, increasing the wait time. Whereas for CPU+FPGA platforms, the wait time is reduced by launching the kernel on CPU while the FPGA slot is being reconfigured. The wait time for workload with 75% of kernels being CPU favoured is largely skewed by the outlier behaviour of kernel arrival rate 1, as shown by the breakdown in Fig. 27. This is because when the workload is small, HRES takes a greedy decision of performing collaborative execution by allocating all resources to a single kernel, this induces wait times for the incoming kernel. Whereas a run-to-completion scheduler would refrain from allocating more resources than minimally required by the kernel, leaving the resources free for incoming kernel and, hence, manages to keep the wait time near zero. The effect of this decision can also be seen on relative makespan in Fig. 23 where makespan latency gets improved. However, this advantage of run-to-completion is soon lost as the workload size increases, i.e. when the number of kernels increases greater than the number of compute units. Overall, the advantage in waiting time also scales with respect to the workload and size and types as shown in the detailed breakdown of waiting times in Fig. 25, 26 and 27.

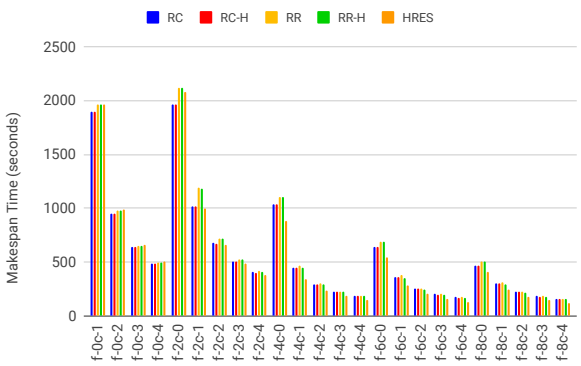

Figure 19: Avg. makespan time for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 75:25 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

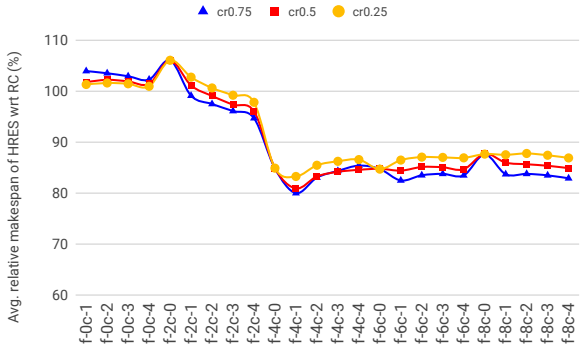

Figure 20: Avg. makespan w.r.t. run-to-completion model for varying CPU-to-FPGA ratio on various execution platforms, where cr denotes CPU favoured kernel proportion and f-nc-m denotes n FPGA slots and m CPUs.

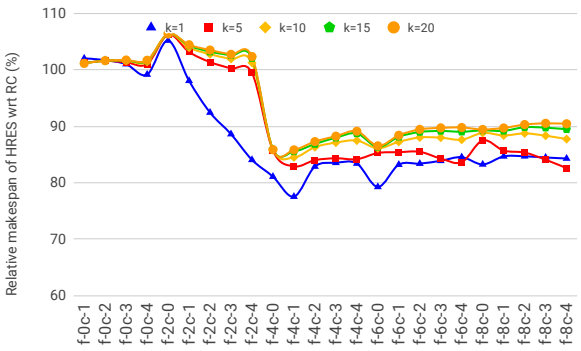

Figure 21: Avg. makespan w.r.t. run-to-completion model for schedules with CPU-to-FPGA ratio of 25:75 on various execution platforms and arrival rates, where k denotes arrival rate and f-nc-m denotes n FPGA slots and m CPUs.

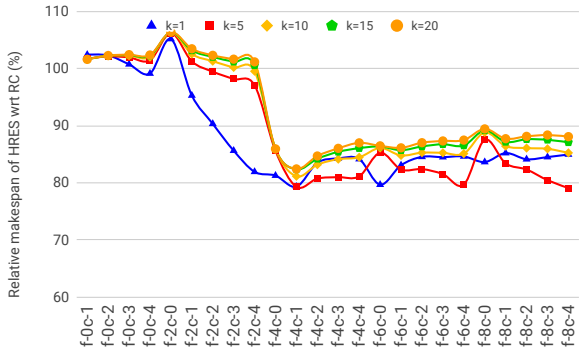

Figure 22: Avg. makespan w.r.t. run-to-completion model for schedules with CPU-to-FPGA ratio of 50:50 on various execution platforms and arrival rates, where k denotes arrival rate and f-nc-m denotes n FPGA slots and m CPUs.

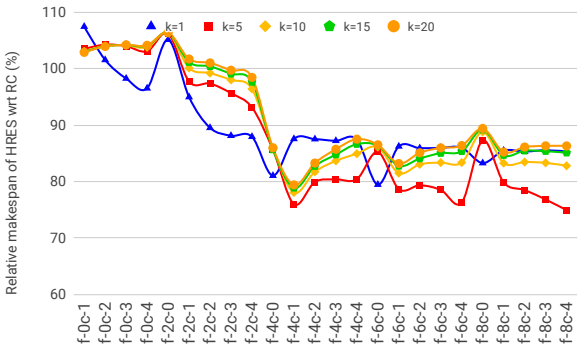

Figure 23: Avg. makespan w.r.t. run-to-completion model for schedules with CPU-to-FPGA ratio of 75:25 on various execution platforms and arrival rates, where k denotes arrival rate and f-nc-m denotes n FPGA slots and m CPUs.

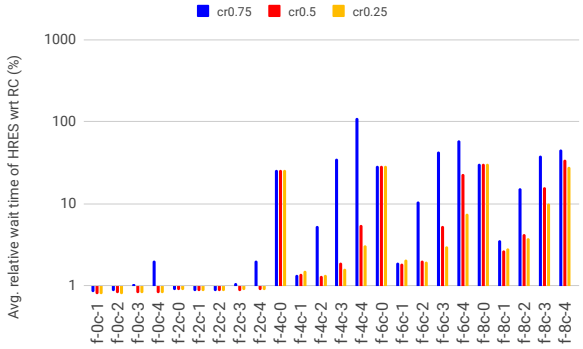

Figure 24: Avg. wait time w.r.t. run-to-completion model for varying CPU-to-FPGA ratio on various execution platforms, where cr denotes CPU favoured kernel proportion and f-nc-m denotes n FPGA slots and m CPUs.

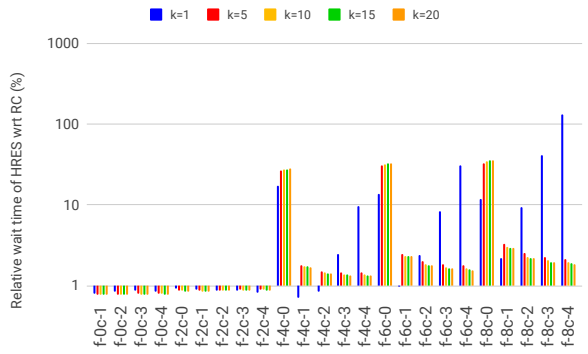

Figure 25: Avg. wait time w.r.t. run-to-completion model for schedules with CPU-to-FPGA ratio of 25:75 on various execution platforms and arrival rates, where k denotes arrival rate and f-nc-m denotes n FPGA slots and m CPUs.

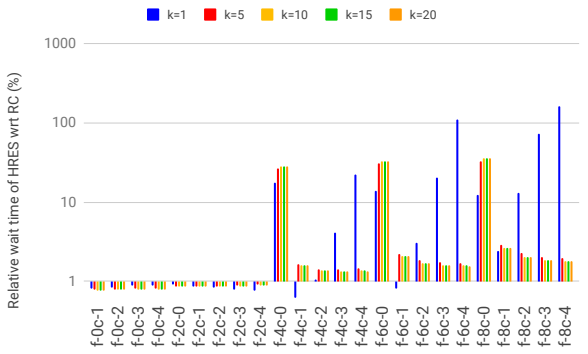

Figure 26: Avg. makespan w.r.t. run-to-completion model for schedules with CPU-to-FPGA ratio of 50:50 on various execution platforms and arrival rates, where k denotes arrival rate and f-nc-m denotes n FPGA slots and m CPUs.

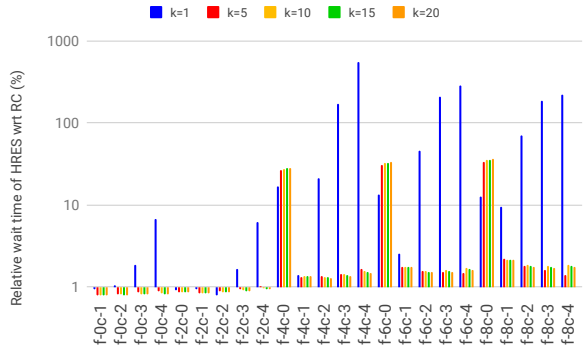

Figure 27: Avg. wait time w.r.t. run-to-completion model for schedules with CPU-to-FPGA ratio of 75:25 on various execution platforms and arrival rates, where k denotes arrival rate and f-nc-m denotes n FPGA slots and m CPUs.

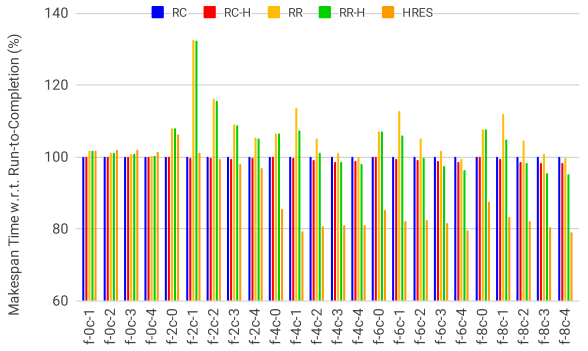

Figure 28: Avg. makespan w.r.t. run-to-completion model for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

If we look in detail at the behaviour of schedulers with an arrival rate of kernel 5 and CPU-to-FPGA ratio of 50:50, Fig. 28 shows relative performance differences compared to standard run-to-completion model. We can see that performing context-switching without resource elasticity via methods like round-robin can harm performance rather than improve it because the partial reconfiguration latency acts as a major overhead. Whereas, when coupled with resource elasticity it maintains an average 20% performance advantage over run-to-completion for FPGAs ≥ 4 slots by maximizing the system utilization. However, the waiting time is drastically improved by 95% for cooperative schedulers with the exception of HRES on pure FPGA configurations. This is due to two main reasons in this workload scenario: 1) the FPGA being overloaded leads to higher PR penalties for maximizing FPGA utilization; 2) fragmentation of resources may not allow a new kernel from the waiting queue to be loaded onto the FPGA. In this case, our HRES algorithm takes a greedy decision and continues accelerating the current kernels by reallocating them to free resources. The need for accelerating CPU favoured kernels on the FPGA only platform further worsens waiting times. Overall, the wait time decreases if we increase the number of FPGA slots available in the system, i.e. moving from 4-slot to 6-slot and 8-slot results in a 36% and 47% lower wait time respectively. This is suggesting that the ability to predict the arrival of a new kernel may be beneficial for the scheduler to avoid taking a greedy decision. Overall, for heterogeneous systems, HRES adapts better by allocating the incoming tasks to CPUs if FPGA resources are not available as a fallback mechanism, hence, allowing the wait time to be reduced considerably. This results in a heterogeneous system with a better makespan than run-to-completion but also with similar wait times as round-robin policies. Note, this advantage is achieved using only better orchestration of resource allocation and not by improving any accelerator implementations. Overall, using CPU and FPGA resources allows the system to deliver both fast response time (small wait time) as well as the high throughput (small makespan) of hardware acceleration.

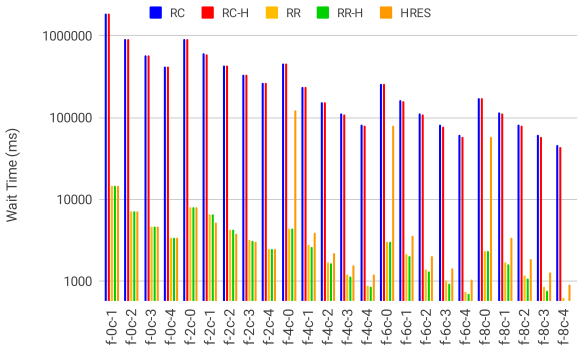

Figure 29: Avg. wait time for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

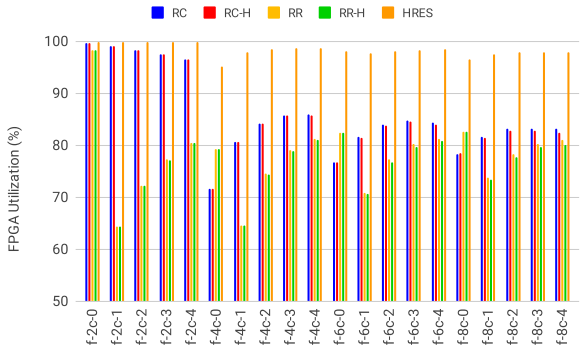

Figure 30: Avg. FPGA utilization for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

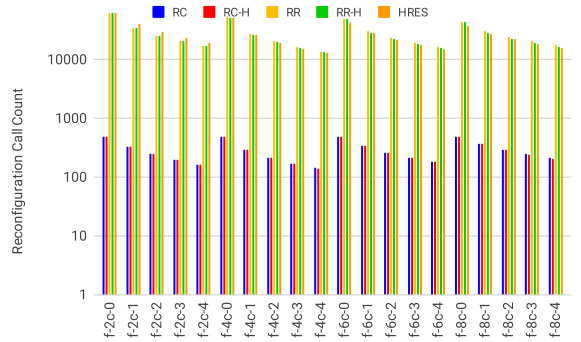

Figure 31: Avg. reconfiguration calls for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

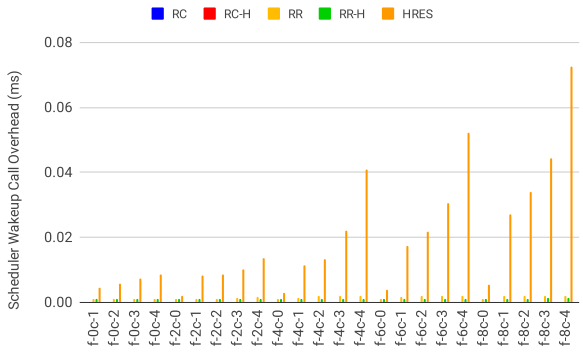

Figure 32: Avg. scheduler wake up call latency (time to allocation decision) for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50 on various execution platforms, where f-nc-m denotes n FPGA slots and m CPUs.

Further, we found that all schedulers manage to keep CPU utilization full as there are no constraints on its allocation. However, for the FPGA resources, runtime constraints can lead to fragmentation which is not dealt well by standard round-robin policies, as shown in Fig. 30. HRES, in contrast, is able to use the free resources and keep the utilization close to the maximum possible because of its ability to replicate accelerators dynamically. To perform this, it employs aggressive reconfiguration, increasing the number of reconfiguration calls considerably, as shown in Fig. 31. However, the reconfiguration call count is similar to other context-switching algorithms such as RR and RR-H, which suggests it is more of a side-effect of context-switching itself rather than HRES in particular. Moreover, HRES performs reconfiguration only if it helps in reducing the overall makespan or accommodating a new arriving kernel. It is important to point out that the higher reconfiguration penalties are greatly amortized by the acceleration of kernels using free resources that would otherwise be left unused, yielding in a better makespan and lower wait times, as shown in Fig. 28 and Fig. 29.

To further quantify and contrast the overhead caused by the schedulers, we measured the average execution time of scheduler wakeup calls for each scheduler on an x86 Intel Core i7-6850K running Ubuntu 16.04 LTS and results of which are shown in Fig. 32. As we can see, the overhead of HRES is 10x to 100x higher than that of other schedulers, particularly after the introduction of CPUs in the system. This is due to the Branch and Bound exploration not being able to terminate early as in the case of FPGAs due to a lack of constraints and goes onto exploring more possible permutations for CPU allocation. However, even in a CPU+FPGA system, this is comparatively negligible as it still requires only microseconds to compute which is marginal when compared to the PR latency which ranges in milliseconds. It can potentially be further reduced if required by adding constraints to CPU allocation for Branch and Bound mechanism based on the CPU cache preferences as a heuristic, such that redundant exploration is minimized with a minor change in performance.

To summarise, heterogeneous resource elastic scheduling can help provide 95% reduction in waiting time for kernels as well as achieve up to 15-20% better system performance (makespan) on varying platforms, workload type and sizes.

7 Conclusions

With the trend of FPGA system towards large scale deployment, all the way from the cloud to the edge, there is strong need for adopting a new resource management paradigm for FPGAs which can better adapt to dynamic workload and maximize system utilization while being transparent to the user. Hence, in this paper, we introduced the concepts and theory of resource elasticity for FPGA systems to build novel runtime systems which can dynamically allocate CPUs and reconfigurable resources, transparently from the user in a multi-tenant environment. We detailed what are the system requirements for such a system and how it can be implemented for OpenCL applications using open-source component such as POCL and our in-house ZUCL FPGA shell, without changing the user interface and effort. The here presented framework of the shell, accelerator compilation, and heterogeneous resource elastic scheduler substantially reduce the design complexity and help in making FPGA acceleration more accessible to a wider base of users. In particular, this paper provides for the first time a holistic solution to the resource management problem arising in heterogeneous FPGA-CPU systems for existing application base by managing FPGA resources in the spatial domain and CPU resources in the time domain, transparently from the user. Our evaluation against standard resource allocation mechanisms on a physical implementation demonstrates that a resource elastic scheduler for CPU+FPGA architectures can achieve $2\times$ performance benefit compared to SDSoC-like platforms for a set of Spector benchmarks. Moreover, our simulation experiments further extrapolate that it can provide scalable benefits of 20% performance improvement and 95% wait time improvement over the standard run-to-completion model, on various platforms across varying workload size and types.

Overall, our findings suggest five key takeaways for scheduling on future FPGA systems:

- Collaborative execution is ideal for heterogeneous resources (CPU and FPGA) and should be performed whenever possible for high performance, as it delivers fast response time and high throughput.
- Co-scheduling application on CPU+FPGA platforms can improve system utilization and, consequently, improve performance.
- Adding cooperative context-switching mechanism can reduce waiting time by 95% but when coupled with fixed resource allocation policies (e.g., round-robin) it can impose major overheads.
- Resource elasticity in almost all cases overcome the overheads of context-switching in dynamic systems while retaining its waiting time benefits and does not require changes in the accelerator implementation or standard user view.
- For dynamic high workload environments which are common in cloud and edge systems, resource elastic systems will often be a better fit than standard run-to-completion runtime systems due to its scalability.

References

[1] Xilinx. Platform overview — xilinx runtime 2019.1 documentation, 2019.

[2] Hayden Kwok-Hay So and Robert W. Brodersen. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, Jul 2007.

[3] Theepan Moorthy and Sathish Gopalakrishnan. Io and data management for infrastructure as a service fpga accelerators. *Journal of Cloud Computing*, 6(1):20, Aug 2017.

[4] F. Chen et. al. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conf. on Computing Frontiers*. ACM, 2014.

[5] M. A. D. Guzmán et al. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *The Journal of Supercomputing*, 75, 2019.

[6] A. Hugo et al. Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach. In *IPDPS*, 2013.

[7] M. Happe et al. Preemptive Hardware Multitasking in ReconOS. In *ARC*, 2015.

[8] D. Koch et al. Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In *FPGA*, 2007.

[9] A. Vaishnav et al. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *HEART*, 2019.

[10] H. Simmler et al. Multitasking on FPGA Coprocessors. In *FPL*, 2000.

[11] A. Morales-Villanueva and A. Gordon-Ross. Partial Region and Bitstream Cost Models for Hardware Multitasking on Partially Reconfigurable FPGAs. In *IPDPSW*, 2015.

[12] T. Xia et al. Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators. In *FPT*, Dec 2016.

[13] A. Bourge et al. Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow. *ACM TRETS*, December 2016.

[14] D. Koch, C. Haubelt, T. Streichert, and J. Teich. Modeling and synthesis of hardware-software morphing. In *2007 IEEE International Symposium on Circuits and Systems*, pages 2746–2749, May 2007.

[15] Snke Hartmann and Dirk Briskorn. A Survey of Variants and Extensions of the Resource-constrained Project Scheduling Problem. *European Journal of Operational Research*, 207(1):1 – 14, 2010.

[16] Farhad Habibi, Farnaz Barzinpour, and S Sadjadi. Resource-constrained Project Scheduling Problem: Review of Past and Recent Developments. *Journal of Project Management*, 3(2):55–88, 2018.

[17] Arno Sprecher and Andreas Drexl. Multi-mode Resource-constrained Project Scheduling by a Simple, General and Powerful Sequencing Algorithm. *European Journal of Operational Research*, 1998.

[18] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. Scheduling Subject to Resource Constraints: Classification and Complexity. *Discrete Applied Mathematics*, 5(1):11 – 24, 1983.

[19] S. Hartmann. Packing Problems and Project Scheduling Models: an Integrating Perspective. *Journal of the Operational Research Society*, 51(9):1083–1092, Sep 2000.

[20] P. Jääskeläinen et al. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 43(5), 2015.

[21] K. D. Pham et al. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *FSP*, 2018.

[22] A. Munshi. The OpenCL Specification. In *Hot Chips*, 2009.

[23] Q. Gautier et al. Spector: An OpenCL FPGA Benchmark Suite. In *FPT*, 2016.

[24] Xilinx. UG909 - Vivado Design Suite User Guide: Partial Reconfiguration. December 2017.

[25] Christian Beckhoff, Dirk Koch, and Jim Torresen. GoAhead: A Partial Reconfiguration Framework. In *FCCM*, 2012.

[26] Khoa Dang Pham, Edson Horta, and Dirk Koch. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *DATE*, 2017.

[27] Malte Vesper, Dirk Koch, and Khoa Dang Pham. PCIeHLS: an OpenCL HLS Framework. In *FSP*, 2017.

[28] G. Joet al. OpenCL Framework for ARM Processors with NEON Support. In *WPMVP '14*, pages 33–40, 2014.

[29] L. Wirbel. Xilinx SDAccel: A Unified Development Environment for Tomorrows Data Center. *The Linley Group Inc*, 2014.

[30] V. Kathail et al. SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC. In *FPGA*, 2016.

A Mapping Heterogeneous Systems to RES problem

In a heterogeneous environment where the resource type can significantly defer (i.e. CPU, GPU, FPGA, ASICs), the scheduling can become challenging while optimizing for performance as each task can have preferred execution mode (e.g., Task A might perform better on CPUs than FPGAs while another task can have the opposite behaviour). In particular, moving from one device type to another can have varying latency depending on the data transfer, setup time and minimum execution length required for the target device.

We can model these scheduling requirements using the variant of project scheduling problem defined in Section 3.2.1. In which heterogeneous tasks ($i \in V$) can run on different devices (mode M_i) each with different latency ($t_{im,p}$) and particular type of resource requirement ($r_{imk,p}$), where the availability of the resource type (device type) at any given time instance is bound by the heterogeneous system configuration (R_k). The cooperative nature of the problem allows changing the mode during the execution at the end of each part given that the minimum execution length (ϵ_{im}) constrained is satisfied. Further, the penalty of changing from one mode to another mode ($O_i()$) models the changing of execution device or mode of execution on the same device. Note, it is also possible to share the device between tasks as the resource capacity (R_k) can be more than 1, allowing task modes such as multi-threading for CPUs. Further, tasks can have a specific type of resource requirements ($r_{imk,p}$) and may not necessarily have mode m for each resource

type (i.e. a task may run on a CPU and GPU but not on FPGA). Finally, the objective is to minimize the total execution time as stated by relation (1).

Note, since resource elastic scheduling for both FPGA alone and heterogeneous systems map to the same problem, they are theoretically equivalent problems. Implying that complexity of scheduling for FPGA resources alone is equivalent to heterogeneous scheduling for platform with FPGA resources. This complies with common expectations given that FPGA can be used to model any other device (e.g. CPU or GPU) as a digital circuit.