Live Migration for OpenCL FPGA Accelerators

DOI:
10.1109/FPT.2018.00017

Document Version
Final published version

Live Migration for OpenCL FPGA Accelerators

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

***Abstract*—FPGAs are currently being deployed at a large scale across data-centres for various applications because of their performance and power benefits. In particular, cloud service operators are now offering FPGAs as a Service. However, to completely integrate FPGAs in a data-centre environment like standard software systems, support for fault tolerance and task migration is essential. In this paper, we propose a live migration technique for FPGA accelerators to provide support for fault tolerance, system maintenance, and resource management. Our technique allows migration of OpenCL accelerators not only within a single FPGA but also across FPGAs with zero downtime. It achieves this by overlapping the computation with data-movements transparently from the user for OpenCL kernels. Moreover, distributed check-pointing mechanisms can be employed to recover from unknown faults with minimal loss of completed work. Altogether it enables system updates such as changing the static FPGA configuration or upgrading the OS without an interruption of service.**

I. Introduction

Reconfigurable computing is one of the approaches for achieving high-performance computing and energy efficiency in the post-Moore era. In particular, at present, many data-centres deploy FPGAs either in the infrastructure or directly offer an FPGA as a Service model to their customers [1]. With this large-scale deployment of FPGAs in distributed systems, it has become important to provide solutions for fault tolerance and high availability of these acceleration services.

In particular, migration is essential in modern data-centres for three main use cases:

- Fault tolerance: In case of a fault, an application needs to be migrated to another node where it can resume execution from the last known consistent state (using check-pointing).
- Maintenance: Consider a scenario where a node needs a hardware/software upgrade but is currently executing an application. With the help of migration, the application can be temporarily moved to a remote node while the upgrade is performed transparently.
- Resource management: Migration permits dynamic load-balancing to redistribute work as the workload changes.

Standard software systems employ live virtual machine migration with distributed check-pointing mechanisms to meet these requirements [2]. Conceptually, for FPGAs this would translate into migrating a hardware task from one FPGA to another with a potentially different number of resources available, using partial reconfiguration and state transfer over the network (as shown in Figure 1). However, for FPGAs, these techniques known from the software systems cannot be applied directly as the application running on FPGAs represent

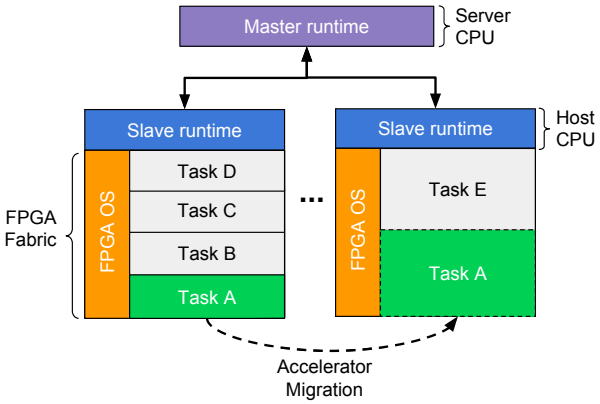


Fig. 1: Migration of FPGA accelerators.

hardware circuits rather than a sequence of instructions. A fully transparent preemptive context switch for FPGAs would require storing and restoring the entire state that is stored in registers, block RAMs and DSP registers [3]–[5]. A large body of research has been carried out for achieving this efficiently [6], however, it still remains one of the expensive operations to perform with many restrictions on how the accelerator hardware is designed (see also Section V-A).

In this paper, we propose taking a more coarse approach by saving and restoring states of accelerators only at particular points in execution time. These points can be referred to as 'consistency points' where little/no internal state propagation is required for the next execution step in the application, implying little/no internal state of the hardware accelerator has to be stored and restored during a context-switch. Often these consistency points occur naturally in data parallel applications for which FPGAs are usually employed. For example, when classifying images using Convolutional Neural Networks, such a point can be at the end of single image classification.

This paper targets OpenCL-specified accelerators as they represent the industry standard for FPGA acceleration using High-Level Synthesis (HLS) and heterogeneous computing in general. An OpenCL application consists of 'work-groups' i.e. a group of lightweight threads (known as work-items). Inside a work-group, work-items can perform synchronization via barriers and atomic operations. However, execution of different work-groups is independent by design in the OpenCL standard [7] to allow parallel execution for high performance. This serves as a natural consistency point for our purposes. We explore two methods which leverage this trait: 1) a blocking method which can migrate the accelerator at the end of a work-group execution, and 2) a non-blocking method which starts

978-1-7281-0214-6/18/$31.00 ©2018 IEEE
DOI 10.1109/FPT.2018.00017

execution of a new work-group on another node before the old node finishes its work-group execution. The latter allows the data movements to be completely overlapped with the computation resulting in zero downtime of the accelerator service.

One major attribute of our approach is that no modification or special care needs to be taken when designing the accelerator for state storage, which allows us to minimize the context-switching penalty to mostly the time required for partial reconfiguration (which could be masked in non-blocking operation).

Furthermore, our proposed methods are not limited only to FPGAs; the same mechanisms can also be extended to all other devices supporting OpenCL i.e. CPUs, GPUs, and ASICs. Workloads could potentially be migrated across these devices in a transparent manner e.g. from a local FPGA accelerator to a remote GPU/CPU.

To the best of our knowledge, this paper is the first to propose live migration mechanism for FPGAs that use OpenCL accelerator modules.

In this paper, our contributions are as follows:

- Migration of OpenCL hardware accelerators with low overhead (Section III).
- Live migration with resource elastic virtualization for zero downtime and load-balancing (Sections III-B, IV).
- Evaluation of the migration techniques with the Spector benchmark suite [8] under different scenarios (Section IV).

II. Background

A. 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 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*. 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 computing units for high performance [7]. At the end of the work-group execution, the result is written back to the global memory without any synchronization for FPGAs.

B. Resource Elastic Virtualization

As in the software case, where a task may take a different share of available CPU time, in the FPGA case the resources available on the FPGA (the available area) may differ due to task migration (see Task A in Figure 1). Therefore it is beneficial to have a system which is *resource elastic*.

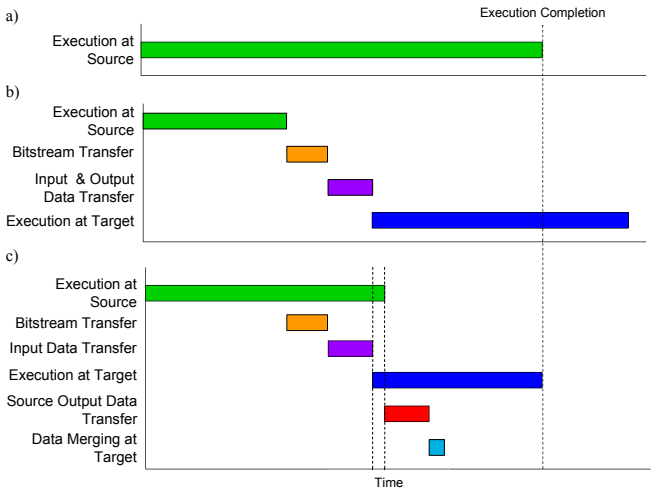


Fig. 2: Execution trace of accelerator where a) represents no migration case, b) represents the blocking migration case and c) represents the non-blocking migration case.

Hardware resource elasticity is the ability of an accelerator to change its resource footprint dynamically at run-time, transparently from its user task [9]. An accelerator may change its size by replication or switching to an implementation alternative. This allows scheduling of accelerators in the time domain as well as the spatial domain to maximize the utilization and system performance. The orchestration of growing and shrinking modules in a user transparent manner is called Resource Elastic Virtualization (REV) [9].

A REV system can dynamically adapt the local allocation of resources, allowing the existing software load-balancing schemes for data-centres to be applied for FPGAs as well.

III. Transparent Migration

A. Techniques

To be able to migrate an OpenCL accelerator, we first need to be able to perform a context switch such that execution of a kernel can be paused and resumed later. There are various techniques which have been employed for hardware preemption [3]–[5], however, these either require time-consuming configuration read-back (saving the content of all the registers and block RAMs in the design) or adding extra logic for retrieving the state of the hardware module (e.g. scan-chains [10]).

Thus, in this paper, we consider an alternative approach of only performing a context switch when the application reaches a *consistency point* whereby a hardware module holds no internal state required for execution correctness. In particular, for OpenCL, these points occur naturally as no synchronization between work-groups and memory write-back to global memory takes place at the end of a work-group execution. This allows an accelerator to be relinquished at the end of work-group if required without the heavy penalty of configuration read-back. Note, since the length of a work-group execution and the number of work-groups is application dependent and commonly bounded, it provides means for a co-operative scheduling scheme.

Given an OpenCL kernel, we can perform the migration by pausing the kernel on a source node and transferring the necessary state and data information to a target node and then resuming the execution again on the target node (as shown in Figure 2b). We call this process 'blocking migration', whereby the computation stops while the state information is being transferred between the nodes. Similar techniques are employed for GPUs [11], where a kernel is broken down into sub-kernels (a group of work-groups) and executed one after the other, allowing the application execution to be paused and resumed at the end of sub-kernel execution for migration purposes. However, the blocking nature of the process represents the major limitation, as the overhead becomes directly proportional to the amount of data transfer required for the application.

As a faster alternative, we propose a non-blocking approach which relies on the data-parallel nature of the OpenCL execution model to overcome this limitation. Given the assumption that input data is not being overwritten during the execution process, input data could be transferred while the execution takes place on the source node along with necessary control information and bitstream of the accelerator to the target node. Further, as the execution order of different work-groups are undefined, the target node can start execution of the next work-group before the source node completes its own work-group execution. In detail the algorithm works as follows (see also the corresponding execution trace shown in Figure 2c):

1) Transfer the accelerator bitstream and the input data while continuing execution on the source FPGA.
2) After initialization of the accelerator on the target FPGA, stop issuing new work-groups to the source FPGA accelerator.
3) Start issuing new work-groups to the target FPGA accelerator.
4) After finishing all outstanding work-group executions on the source FPGA, transfer the output data to the target FPGA.
5) Continue execution of the remaining work-groups on the target FPGA and perform data merging on the output data in the background.

The above scheme leads to no-break execution from the user's perspective as, at any point in time, computation is taking place in the system. Moreover, the technique can be modified to support rollback in case of an unexpected fault or emergency migration, by re-executing the work-group on the target FPGA to reduce the total time needed to perform the migration, at the cost of an overhead for total kernel execution time. This allows masking of node failures in a large system. To mask node failures, input data and configuration data has to be stored redundantly in the system and node or task monitoring is needed (e.g. heart-beat monitoring messages).

B. Virtualization Architecture

Our virtualization architecture is based on a master and slave approach which is similar to Apache Hadoop YARN [12] (one of the industry standard cluster management systems), as shown in Figure 3. Here each node has a copy of both master and slave run-time such that the master can be selected based on a standard election algorithm in case of a fault occurrence. The slave run-time system (as shown in Figure 4) employs Resource Elastic Virtualization (REV) such that the number of instances or module implementations can be changed dynamically based on the workload [9]. This allows our system to test one of the prime use cases of migration i.e. migrating a kernel from an overloaded FPGA to a free FPGA for load-balancing. Further, the slave run-time provides heart-beat signals and checkpoint information to a master node on a regular basis to detect and mitigate unexpected crashes. The master run-time is responsible for host code execution and can be used to distribute workload across multiple FPGAs.

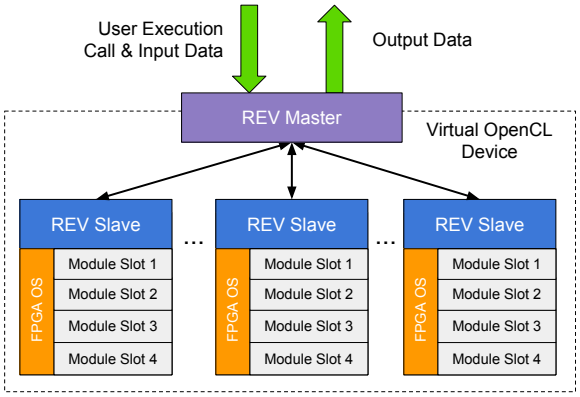


Fig. 3: Virtualization architecture to abstract number of nodes in the system.

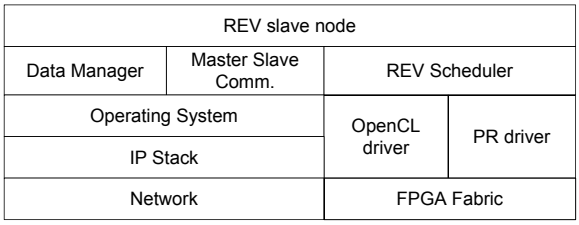


Fig. 4: Software architecture required for REV and live migration.

The data payload between slave nodes is transferred using Transmission Control Protocol (TCP) to ensure reliability and error checking of the data during transit. The transferred data is then picked up by the slave run-time running on the target node and is used to set up buffers, control state and the programming of OpenCL kernels via partial reconfiguration and a generic OpenCL driver.

Merging of the output data requires identification of the memory locations in buffers written by the source and target nodes respectively. In case of mapping the output data from a source buffer to a target buffer, only the memory locations written by the source must be updated into the target buffer. This can be performed by tracking the memory location with write flags and updating only the memory location in the target buffer where it is set in the corresponding source buffer. However, in most cases, this can be easily determined based on

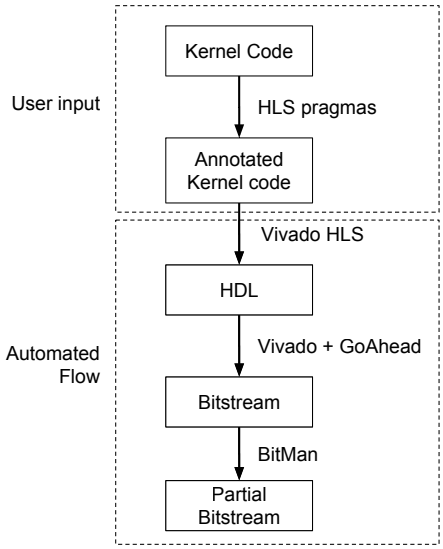


Fig. 5: Design flow for relocatable accelerator generation from OpenCL kernels.

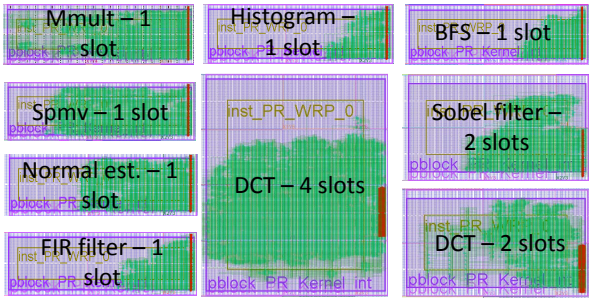
the work-item ID as it tends to correspond one-to-one or one-to-N with the index in the output buffer. It allows the tracking to be completely eliminated if the information of how many output values are generated per work-item is available. This can either be provided by the user as an additional argument in the execution API call (extension of OpenCL API) or by a compiler as meta-data. In this paper, we take the API extension approach to minimize the data merging time as the current OpenCL compilers for FPGAs do not provide such information. Note, this does not require extra effort from the developer to identify this information as it is a part of the algorithmic design (i.e. how much work each thread needs to perform).

C. Accelerator Generation

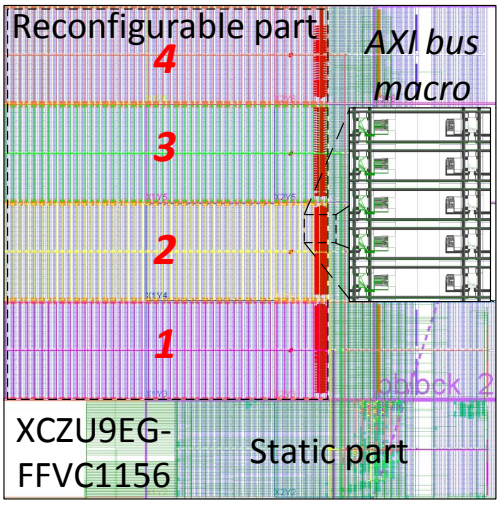
Studying the example from Figure 1 again, we can see that migration may require relocating a module to different positions in the system or even using different sized modules while always being able to integrate and communicate with these modules. To generate *relocatable accelerators* for migration within an FPGA fabric, we use the design flow shown in Figure 5. 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 properties 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 OpenCL accelerators.
2) All physical resources of an accelerator must stay in a pre-defined bounding box, including routing wires. No wiring violation to surrounding regions is allowed.

The above conditions are guaranteed by utilizing a custom Partial Reconfiguration (PR) methodology [13] instead of



(a)



(b)

Fig. 6: The physical implementation of the system on Zynq UltraScale+ (ZCU102) where a) shows the accelerators from Spector Benchmark [8] and b) shows the static system with neighbouring partial regions (slots) based on the ZUCL framework [13].

following the default Xilinx PR flow [14]. In this custom PR flow, required place-and-route constraints are transformed to TCL scripts with automatic assistance from the GoAhead tool [15]. These TCL scripts are then applied to Vivado for the accelerator's physical implementation.

In order to use the same accelerator bitstream at different positions on the chip, the bitstream manipulation tool and API, BitMan [16] is used to generate partial bitstreams at design time and to relocate accelerators at run-time. Figure 6 displays our static systems and 8 implemented relocatable accelerators from the Spector benchmark suite [8]: Sobel filter, Sparse Matrix-Vector Multiplication (SPMV), 3D-distance Normal Estimation, Finite Impulse Response (FIR) filter, Histogram, Matrix Multiplication, Breadth First Search (BFS) and Discrete Cosine Transform (DCT).

Note, our approach can also potentially be used with other platforms such as PCIeHLS [17].

IV. Experimental Evaluation

To evaluate the different mechanisms and scenarios for live migration, we use OpenCL kernels from the Spector Benchmark suite [8] representing applications from signal

TABLE I: Spector benchmark suite implementation and its resource usage.

Applications	Slots
3D Normal Estimation of Point Cloud	1
Sparse Matrix-Vector Multiplication	1
Sobel Filter	2
Time-domain FIR	1
Discrete Cosine Transform	2, 4
Histogram's main kernel	1
Matrix Multiplication	1
Breadth First Search[1]	1

TABLE II: Resources available per slot i.e. partial region.

Resource Type	Quantity
CLB LUTs	32640
BRAM Tiles	108
DSPs	336

processing, machine learning, computer vision and statistics. This diversity of applications is important to capture the various different compute to data ratios required, as migration overhead is assumed to be highly sensitive to it. Further, Spector is denoted with various optimization parameters for testing design space exploration tools which, in our case, are used to generate different implementation alternatives for employing resource elastic scheduling as part of our load-balancing mechanism. Note, Spector was originally written for Altera OpenCL platforms, hence we ported 8 of 9 application benchmarks (as shown in Table I) to Xilinx platforms, the only difference being that the SIMD optimization pragma is not employed as it is unavailable for Vivado HLS. The remaining application benchmark (merge sort) is not considered as it can potentially change input data, which violates our constant input data assumption and tends to generate incorrect results (at run-time) when synthesized with Vivado HLS 2016.2. It can be re-written such that our requirements are met and a correct result is produced, however, this would require changing the benchmark implementation considerably. All the experiments considered here perform migration halfway through the application execution for a fair comparison. The effective data transfer speed between the local and remote node is about 238 Mbps over the Gigabit Ethernet connection when performing TCP and therefore contributing to the main bottleneck during data transfer phase. The source and target FPGA platforms are Zynq UltraScale+ (ZCU102 development boards) consisting of 4 slots (partial regions) on an XCZU9EG-FFVC1156-1-I device (see Figure 6b). Each region provides the same resource footprint, allowing relocatable modules to be employed as shown in Figure 6b. The reconfigurable resources available per slot are listed in Table II. The relocatable modules for these regions are generated using the flow described in Section III-C.

A. Maintenance & Load-Balancing Scenario

Migration for maintenance can be coupled with load-balancing algorithms, to re-distribute workload across the other nodes while the system is upgraded/recovered. Hence in this section, we evaluate both scenarios (maintenance and load-balancing) together and highlight the key aspects of migration in terms of performance and overhead. In particular, we test the scenarios shown in Figure 7, where a kernel is migrated to a target node with equal or more resources. The migrated kernel may replicate or change module implementation on the target node, if possible, based on resource elastic scheduler [9].

[1]We merged two kernels of BFS by manual in-lining for better overhead representation of the entire application.

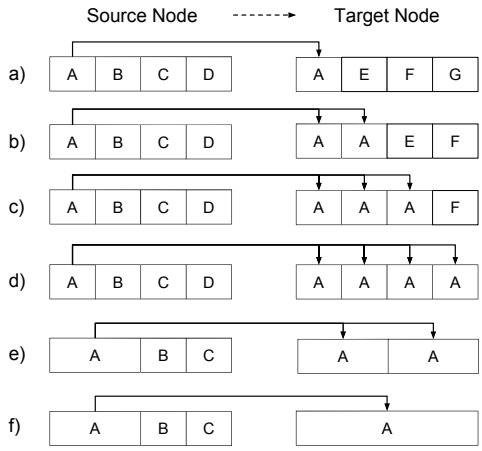


Fig. 7: Various migration scenarios of hardware tasks on a 4-slot FPGA are highlighted: a) shows migration to an equivalently busy node, b) to e) shows migration to the target node with free resources using replication and f) shows migration where module implementation is changed to maximize the benefit from free resources on the target node.

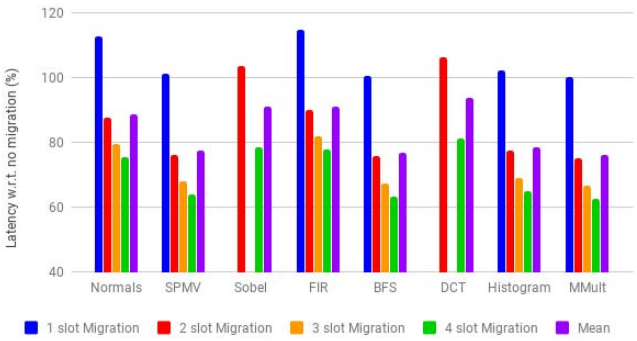


Fig. 8: The relative execution latency of kernels w.r.t. no-migration with the smallest module is shown for blocking migration mechanism. Note that the x-axis begins at 40%.

Figure 8 shows the execution overhead of the migration when performing blocking migration using *replication* on the target node. Overall, it results in overhead ranging from 0.064% for Breadth First Search to 14% for an FIR filter accelerator depending upon the compute-to-data ratio. In particular, when performing migration to another node with a lower workload (allowing more resources for acceleration), we note that the overhead is paid off by the acceleration achieved for all 8 applications, which implies migration coupled with load-balancing is preferable to migration to an equally busy node (even when performing maintenance). However, there still exist applications, like 3D normal estimation and FIR filter, which suffer from 13% overhead on average when

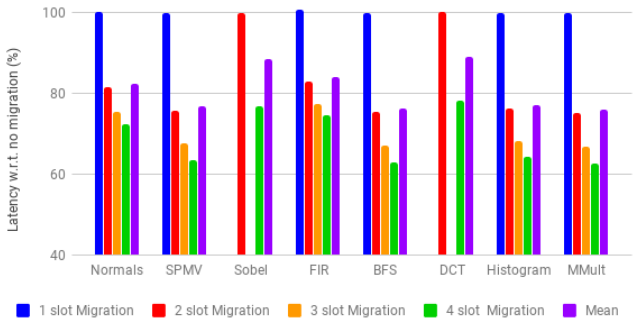


Fig. 9: The relative execution latency of kernels w.r.t. no-migration with the smallest module is shown for non-blocking migration mechanism. Note that the x-axis begins at 40%.

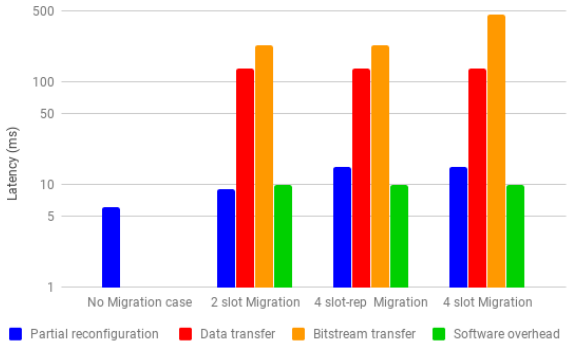


Fig. 10: The latency of important steps involved in migration for DCT benchmark.

moving to an equivalently busy node. These latency penalties are considerable for environments such as the cloud, where the service availability requirements could be strict as well as the probability of faults or need for load-balancing is high due to its large scale.

In contrast to blocking migration, non-blocking migration allows for almost zero overhead in terms of service downtime (i.e. when user computation is not taking place). Figure 9 shows the execution latency for the applications using replication on the target node, where the maximum overhead is 0.96% for FIR filter, consisting mainly of the software latency and partial reconfiguration. In the general load-balancing case, when migrating to lower workload node, applications show performance benefits ranging from 17% to 37.5% using replication despite the considerable latency of data and bitstream transfer latency (as shown in Figure 10).

To understand the gap between these two migration mechanisms, consider the case of DCT migration, as shown in Figures 11 and 12. The major contribution of additional latency belongs to the data and bitstream transfer over Ethernet. Thus, when utilizing the non-blocking mechanism where the computation continues on the source node while all the necessary data transfer for next work-group execution takes place, it essentially absorbs this major latency to provide very

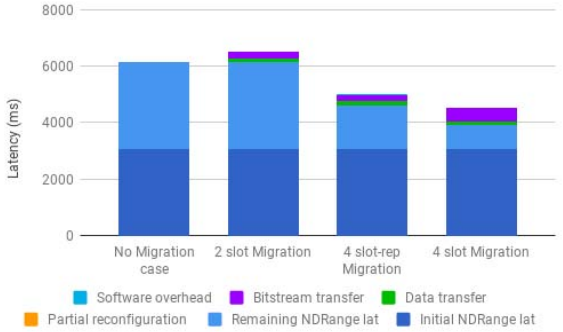


Fig. 11: The latency breakdown of execution trace for blocking migration scenario for DCT benchmark.

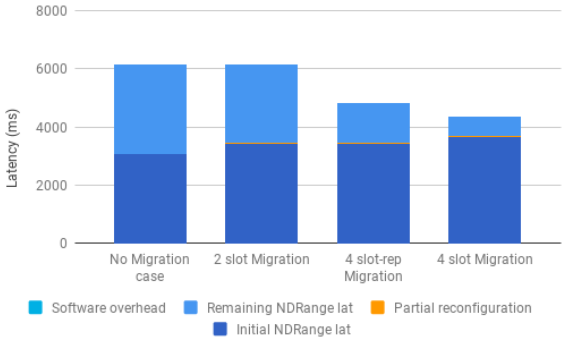


Fig. 12: The latency breakdown of execution trace for non-blocking migration scenario of DCT benchmark.

low overhead by increasing the time spent on the source node. The key aspect which needs to be noted for load-balancing is that, when a bitstream transfer represents a considerable amount of latency during migration, while moving from a small module on the source node to a large module on the target node, the large module must provide super-linear performance to overcome the overhead caused (i.e. a module taking n times the slots (resources) has to deliver more than n times the performance). This super-linear benefit can be gained by better sharing of data and control logic compared to replication for many applications. Additionally, the work left on the target node also needs to be enough for acceleration. An example of this is the DCT (as shown in Figure 12) where switching to a large module is preferable as the data transfer phase does not increase the time spent on the source node considerably.

Note that the bitstream transfer latency can be mitigated further by keeping local copies. It is also possible to reconfigure while receiving the bitstream with a more complex protocol to handle transfer problems.

B. *Fault Tolerance Scenario*

Migration can also help to provide support for fault tolerance at node level (i.e. when a node is unable to function correctly while other nodes can continue their progress). This

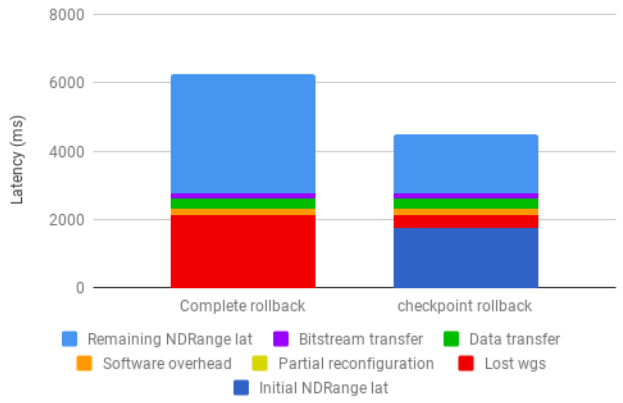


Fig. 13: Fault tolerance scenario where an unforeseen fault occurs halfway through the execution of 3D normal estimation benchmark.

fault scenario may occur due to a software bug, hardware fault (e.g., power failure) or problem in the local operating system. Hence, to evaluate this, we consider two scenarios: 1) Unexpected fault leading to a loss of work and 2) Full static system update to mitigate bugs or add new features.

1) Unexpected Fault: To test the first scenario, we implemented a regular checkpoint mechanism using a server, as it is performed in traditional distributed systems with the help of logs [2]. In our test environment, the snapshot of the application is taken every 25% of execution length based on the number of work-groups and the unexpected fault is mimicked by a forced operating system shutdown. After fault occurrence, kernel execution is resumed again on another node. The standard implementations of OpenCL run-times for FPGAs [18], [19] do not perform context switching and, hence, they fail to leverage checkpoint mechanisms in case of fault occurrence, leading to a complete rollback (i.e. restart) of the kernel execution. In contrast, our technique allows the standard distributed computing mechanism to be integrated with OpenCL such that partial rollback can be performed for unexpected faults (e.g. power failure). This can save up to 75% of completed work in our scenario over standard OpenCL mechanisms (as shown in Figure 13) and the advantage is proportional to the frequency of check-pointing until check-pointing overhead starts to dominate. Equation 1 states this relation formally, where T is the total time taken for kernel execution with mean time to fault μ, n is the number of work-groups with latency of L_w, D is the downtime, R is the time taken for recovery, c is the number of work-groups between checkpoints (i.e. checkpoint interval) and L_c is the checkpoint latency.

$$T \approx nL_w\left(1+\frac{1}{\mu}\left(D+R+\frac{cL_w}{2}\right)\right)+\frac{nL_c}{c} \qquad (1)$$

2) System Update: Here, we describe the steps as needed for the ZCU102 prototype system running Linux on the embedded ARM core. Firstly, a new Linux image with updated static logic needs to be written to SD-Card while the migration to another node is performed. This allows the ARM core on chip to boot correctly with new device tree information. Then a reboot sequence needs to be initiated on the source node using the SD-Card, followed by initialization of the run-time system and migration back to the source node. The latency of each of these steps for the Matrix-Multiplication benchmark using 4 slots through replication is shown in Table III. The total latency for the static update is 33.26 seconds, the majority of which corresponds to the Linux boot up period. In terms of the total execution time of the Matrix-Multiplication benchmark, the overhead represents 13.25% of the total execution time. During this period live migration can allow continuous provisioning of acceleration services by moving to another node without pausing the kernel execution.

V. Related Work

This section highlights the related work for the crucial aspect of the FPGA accelerator migration: context-switching. Followed by migration techniques employed for GPU acceleration.

A. Hardware Check-pointing

Preemptive hardware context-switching shares its root with software systems, where the idea is to relinquish control of the reconfigurable resources on demand for sharing the FPGA in the time domain. To perform this transparently, earlier approaches employed configuration readback techniques to store and restore the accelerator state at run-time [3]–[5], [20]. However, this imposes heavy penalty on performance in addition to partial reconfiguration as every memory element in the FPGA fabric (i.e. registers, BRAMs and distributed RAM) must be read and written to for a complete context switch. To lower this overhead, strict storage element layout has been proposed such that the resources to be readback is reduced. This was further improved by employing scan chains for the important state registers in HDL [10] as well as automatic identification of such variables in HLS [21].

Moreover, accelerators with functionality for state access have also been proposed to lift the burden of constrained resource allocation for storage elements [6]. In particular, the co-operative approach of only performing the context-switch at minimal state space in execution has been proposed by Koch et al. [10] and Xia et al. [22] for custom hardware accelerators in user RTL descriptions.

However, all these approaches impose either heavy penalties or restrictions on accelerator design. Which, in turn, lower the potential performance in most practical cases, as the optimization techniques (e.g., latches, retiming, multi-cycle paths,

TABLE III: Full static system update latency breakdown.

Phase	Latency (ms)
Write-to-SD-Card + Migration to Temp. Node	852
Shutdown Period	600
Boot-up Period	29420
Static Logic Reconfiguration	104
Run-time Initialization	2010
Migration to Source	278
Total Time Taken	33264

multicycle I/O transactions, pipeline registers in primitives such as in multipliers and memory blocks) cannot be applied freely and lead to additional resource consumption or may not be supported at all.

B. GPU migration

GPUs are typically integrated into Virtual Machines (VM) to allow sharing between multiple different tenants [23], [24]. There are two ways in which the access to GPU is provided. 1) A pass-through mechanism where a GPU is directly accessed (mutually exclusive) by the application which does not allow transparent migration. 2) The other mechanism involves trapping the command calls [25]–[27] or memory allocation from the applications and mapping them into separate buffer spaces to provide memory isolation [28]. Since the commands tend to run-to-completion for GPUs, the usual approach is to block the issuance of new commands to GPU and wait for the current command to finish (or in some cases wait for the whole command queue to drain), to transfer state between nodes safely [28]. However, this applies block migration on coarse granularity. A more fine-grained approach specifically for OpenCL is to break the kernel execution call into sub-kernels i.e. execute n work-groups at a time out of the whole NDRange [11]. This allows reducing the overall waiting time for migration, using the blocking migration.

VI. Conclusion

In this paper, we presented a novel approach for live migration of OpenCL FPGA accelerators using asynchronous mechanisms and co-operative scheduling for hardware check-pointing. It allows migrating accelerators not only within FPGA but also across FPGAs for fault tolerance, maintenance, and load-balancing purposes. Our analysis with the Spector benchmark suite shows that the asynchronous approach for data parallel applications on FPGAs can allow almost zero overhead for migration and can be performed transparently from the user. Moreover, the check-pointing and migration mechanism can provide fault tolerance for FPGAs when coupled with a standard distributed system architecture, by resuming the accelerator execution from the last known consistent state. Consequently, this provides fault tolerance with fewer spare nodes (i.e. lower cost for redundancy), as the resource allocation across the cluster can be changed dynamically with low overhead. Further, it can serve as a base of dynamic load-balancing systems to provide a service with less compute nodes for saving cost and power. Lastly, it enables to perform system updates such as changing static FPGA configuration or upgrading the OS without service downtime.

VII. Acknowledgements

This work is supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632).

We would like to thank Dr James Garside from the University of Manchester for his valuable suggestions during manuscript editing.

Finally, we thank the Xilinx University Program for supporting this research with hardwares and design tools.

References

[1] A. Vaishnav, K. D. Pham and D. Koch, "A Survey on FPGA Virtualization," in *FPL*, 2018.
[2] M. Treaster, "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems," *CoRR*, vol. abs/cs/0501002, 2005. [Online]. Available: http://arxiv.org/abs/cs/0501002
[3] H. Simmler et al., "Multitasking on FPGA Coprocessors," in *FPL*, 2000.
[4] A. Morales-Villanueva and A. Gordon-Ross, "Partial Region and Bitstream Cost Models for Hardware Multitasking on Partially Reconfigurable FPGAs," in *IPDPSW*, 2015.
[5] M. Happe, A. Traber, and A. Keller, "Preemptive Hardware Multitasking in ReconOS," in *ARC*, 2015.
[6] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification," in *FPL*, 2001.
[7] A. Munshi, "The OpenCL Specification," in *Hot Chips*, 2009.
[8] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA Benchmark Suite," in *FPT*, 2016.
[9] A. Vaishnav, K. D. Pham, D. Koch and J. Garside, "Resource Elastic Virtualization for FPGAs using OpenCL," in *FPL*, 2018.
[10] D. Koch, C. Haubelt, and J. Teich, "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation," in *FPGA*, 2007.
[11] S. Xiao et al., "Transparent Accelerator Migration in a Virtualized GPU Environment," in *CCGRID*, 2012.
[12] V. K. Vavilapalli et al., "Apache Hadoop YARN: Yet Another Resource Negotiator," in *SOCC*, 2013.
[13] K. D. Pham, A. Vaishnav, and D. Koch, "ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications," in *FSP*, 2018.
[14] Xilinx, "UG909 - Vivado Design Suite User Guide: Partial Reconfiguration," Dec. 2017.
[15] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A Partial Reconfiguration Framework," in *FCCM*, 2012.
[16] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*, 2017.
[17] M. Vesper, D. Koch, and K. D. Pham, "PCIeHLS: an OpenCL HLS Framework," in *FSP*, 2017.
[18] L. Wirbel, "Xilinx SDAccel: A Unified Development Environment for Tomorrows Data Center," *The Linley Group Inc*, 2014.
[19] T. S. Czajkowski et al., "From Opencl to High-Performance Hardware on FPGAS," in *FPL*, 2012.
[20] O. Knodel et al., "Migration of Long-running Tasks Between Reconfigurable Resources Using Virtualization," in *HEART*, 2016.
[21] A. Bourge, O. Muller, and F. Rousseau, "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 1, pp. 9:1–9:23, Dec. 2016.
[22] T. Xia, J. Prvotet, and F. Nouvel, "Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators," in *FPT*, 2016.
[23] M. Dowty and J. Sugerman, "GPU Virtualization on VMware's Hosted I/O Architecture," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 73–82, Jul. 2009.
[24] Gupta, Vishakha et al., "GViM: GPU-accelerated Virtual Machines," in *HPCVirt*, 2009.
[25] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, June 2012.
[26] Giunta, Giulio et al., "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," in *EuroPar*, 2010.
[27] J. Duato et al., "rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters," in *HPCS*, 2010.
[28] M. Gottschlag et al., "LoGV: Low-Overhead GPGPU Virtualization," in *HPCC + EUC*, 2013.