PAPER Special Section on Reconfigurable Systems

Enabling FPGA-as-a-Service in the Cloud with hCODE Platform

Qian ZHAO[†a)], **Motoki AMAGASAKI**[†], **Masahiro IIDA**[†], **Morihiro KUGA**[†], *and* **Toshinori SUEYOSHI**[†], *Members*

SUMMARY Major cloud service providers, including Amazon and Microsoft, have started employing field-programmable gate arrays (FPGAs) to build high-performance and low-power-consumption cloud capability. However, utilizing an FPGA-enabled cloud is still challenging because of two main reasons. First, the introduction of software and hardware co-design leads to high development complexity. Second, FPGA virtualization and accelerator scheduling techniques are not fully researched for cluster deployment. In this paper, we propose an open-source FPGA-as-a-service (FaaS) platform, the hCODE, to simplify the design, management and deployment of FPGA accelerators at cluster scale. The proposed platform implements a Shell-and-IP design pattern and an open accelerator repository to reduce design and management costs of FPGA projects. Efficient FPGA virtualization and accelerator scheduling techniques are proposed to deploy accelerators on the FPGA-enabled cluster easily. With the proposed hCODE, hardware designers and accelerator users can be organized on one platform to efficiently build open-hardware ecosystem.

key words: *FPGA-as-a-service, hardware acceleration, open-source hardware*

1. Introduction

Field-programmable gate arrays (FPGAs) have demonstrated great speed performance and power-efficiency advantages over conventional processors in a variety of application domains, such as machine learning, big data, and image processing. Recently, major cloud service providers, including Amazon and Microsoft, have started employing FPGAs to build high-performance and low-power-consumption cloud capability. In line of this, the necessity of software (SW) and hardware (HW) co-design for cloud applications demands new development methods, tools and community construction efforts.

There are two main challenges in integrating FPGAs into the modern cloud system: the high HW/SW co-design complexity and HW virtualization. First, there is no clear division between the HW design flow and the SW design flow. Co-designing of HW and SW under current methods requires developers to have knowledge of both HW and SW development, which is difficult especially when architecting large projects involving tens to hundreds of people. In addition, there is no sophisticated open-source platform for hardware developers to publish their designs and from which software developers can directly obtain accelerators out of the box.

Second, modern cloud computing servers are highly virtualized, so that multiple applications from one or multiple tenants can be concurrently executed on the same physical servers without disturbing each other, which improves the server utilization rates and reduces cloud operating expenses. In order to employ FPGAs in such a highly virtualized cloud, it is necessary to provide virtualization techniques for sharing the resources of one FPGA, such as lookup tables (LUTs), block RAMs (BRAMs) and digital signal processors (DSPs), to allow multiple accelerators to be implemented. Based on the FPGA virtualization, a new scheduling policy for efficient accelerator arrangement at the cluster scale is also essential. Different with the conventional SW task schedulers that only consider processor cores and memory size, an accelerator scheduling policy for FPGAs has to take FPGA resource utilization as well as communication bandwidth in consideration.

In this paper, we propose an FPGA-as-a-service (FaaS) platform named the Heterogeneous Computing Oriented Development Environment (hCODE). The hCODE is the first open-source platform approach for simplifying the design, management, and deployment of FPGA accelerators at a cluster scale. As a solution of the described problems, the main contributions of hCODE are as follows.

1. *FPGA virtualization.* A Shell-and-IP design pattern is adopted in hCODE. The virtualized shell logics provide high reusability and portability for HW designs. Moreover, multiple accelerators can be implemented on one FPGA with a multi-channel shell to improve HW utilization rates.
2. *Accelerator repository.* An accelerator repository is developed to host shell and IP projects. We also provide tools for easily searching, downloading and auto-assembling accelerators.
3. *Cluster management.* Standalone master and slave server functions are embedded in hCODE to simplify cluster management. An accelerator scheduler is implemented to efficiently allocate accelerators onto the cluster.

The prototype of hCODE is open-sourced on Github (https://github.com/hCODE-FPGA/hCODE) for evaluation. The rest of this paper is organized as follows. Section 2 introduces related works. A quick overview of hCODE is

Manuscript received April 27, 2017.
Manuscript revised September 6, 2017.
Manuscript publicized November 17, 2017.

[†]The authors are with Faculty of Advanced Science and Technology, Kumamoto University, Kumamoto-shi, 860–8555 Japan.
a) E-mail: cho@arch.cs.kumamoto-u.ac.jp

DOI: 10.1587/transinf.2017RCP0004

given in Sect. 3. Section 4–6 describes the proposed FPGA virtualization, accelerator repository and cluster deployment of hCODE platform, respectively. Evaluation and case studies are shown in Sect. 7. Section 8 gives conclusions and discusses future work.

2. Related Work

2.1 FPGA Design Pattern

Packaging common modules into a reusable shell framework is a traditional practice to improve HW design productivity. Although FPGA vendors provide fundamental intellectual properties (IPs) in their development tools, it still requires efforts to utilize these IPs in specific projects. Third-party frameworks like RIFFA [1] and XILLYBUS [2] have done a lot of works on HW and drivers in order to provide an easy-to-use interface based on the official PCIe IP core. [3] introduced a complete open-source framework that provides PCIe, Ethernet, and DRAM interfaces. Microsoft also introduced a shell-and-role HW architecture in [4]. However, no framework can efficiently fit all application scenario. General purpose frameworks are complexly implemented for more functionality and flexibility, which may result in a larger area and a lower frequency for FPGA designs.

2.2 Accelerator Management and Open-Source

OpenCores is an open-source HW website that offers a variety of open-source IPs [5]. However, OpenCores does not provide any functions beyond online storage. In contrast, modern SW package managers provide an online repository, version control, dependency control, and even allow automatic integration of third-party packages into a user project [6]. However, managing a HW project is more challenging when considering portability and scalability on different devices.

2.3 Accelerator Deployment on the Cloud

There are several related works on FPGA utilization within cloud computing. Some works [7]–[9] integrate FPGAs into the OpenStack system, while Blaze [10] focuses on accelerating big data processing platforms such as Hadoop YARN and Apache Spark. For the FPGA virtualization, most works [7]–[9] partition an FPGA into several pre-defined partial reconfiguration (PR) regions to implement accelerators. However, because the sizes of pre-defined PR regions cannot always fit requested accelerator, an efficient full reconfiguration mechanism of the entire FPGA is necessary, which is not discussed in these works. For accelerator scheduling, an accelerator-centric scheduling method is implemented in Blaze [10], which suggests arranging tasks requiring the same accelerator on the same FPGA-enabled server to avoid reconfiguration overhead. However, FPGA virtualization for resource utilization improvement is not considered in this work. An accelerator scheduling policy based on virtualized FPGAs is mentioned in [7], which suggests scheduling accelerators of different characteristics together to improve efficiency. The authors also used a simple case to prove their ideas, however, no details of the implementation methodology is shown.

3. hCODE Platform Overview

The target of the proposed hCODE platform is to simplify the design, management, and deployment of FPGA accelerators. As Fig. 1 shows, there are three participant roles on the hCODE platform, the shell designer, the IP designer and the accelerator user, who are organized with an open accelerator repository. A shell designer provides application-independent logics. An IP designer develop application logics by reusing a shell to reduce design cost. At last, accelerator users can easily deploy accelerators on a single machine or an FPGA-enabled cluster by using hCODE commands.

The interface provided for users of hCODE is a command line tool. The main functions of hCODE are listed in Table 1. And three main commands are introduced as follow.

ip get: This command downloads necessary IPs and shell projects to make an accelerator project. The user specifies the name of the requested IPs as arguments, and then hCODE automatically downloads these IPs as well as a shell that compatible with current HW setup to make the accelerator project.

ip make: This command performs the compilation

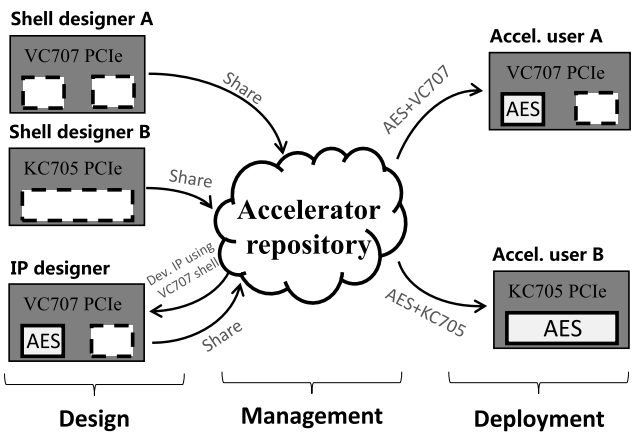


Fig. 1 An example application on hCODE platform.

Table 1 List of main hCODE commands.

Command	Functional description
setup	Set up the hCODE system files in ~/.hcode folder.
repo	Add/remove project repositories.
list/search	List projects/Search projects with a *QUERY* string.
ip get	Download specified IP(s) and compatible shells.
ip make	Compile the accelerator from IP(s) and shell project.
fpga program	Program specified bitstream to local/remote FPGA(s).
cluster master	Start a cluster master server.
cluster slave	Start a cluster slave server.
cluster status	Collect FPGA status and logs from slave servers.
cluster schedule	Deploy accelerator on cluster with hCODE scheduler.

of the accelerator project prepared by the *ip get* command. This process configures and compiles the IPs, integrates the product of the IPs into the shell project, and finally compiles the combined project to generate a bitstream file.

cluster schedule: This command reads user-specified IP names, target server names and other constraint information from command arguments, generates an allocation solution on the cluster, calls *ip get* and *ip make* to generate accelerator bitstream, and finally configure target FPGAs with *fpga program* command.

4. FPGA Virtualization of hCODE Platform

In this section, we explain the basic design methodology on the proposed hCODE platform. We first introduce the Shell-and-IP design pattern for FPGA virtualization, which provides several important HW implementation features such as design reusability, IP portability and FPGA on-chip resource sharing. Then we discuss details of FPGA on-chip resource virtualization that allowing multiple IPs to be implemented on one FPGA.

4.1 Shell-and-IP Design Pattern

The hCODE implements the Shell-and-IP design pattern and defines participant roles to reduce the design costs of accelerators and provide FPGA virtualization capabilities. It is a common methodology to partition a HW design into an application-independent shell part and an application-logic IP part, as shown in Fig. 1. The shell logic, which is also named the service logic in [7] or the static logic in [8], provides common modules and functions such as communication, memory control, etc. IPs and a shell are linked to produce an accelerator. The shell layer can also be considered as HW virtualization layer. With this method, a developed shell can be reused by different IP projects to reduce IP design cost. An IP can be integrated with different shells to achieve portability between FPGAs. IP portability is necessary for open-HW community construction and FPGA-enabled cloud management because different users or cloud providers typically use different FPGAs.

Based on this design pattern, we define three participant roles on the hCODE platform: the shell designer, the IP designer and the accelerator user. The shell designer provides a shell hardware design and a shell driver. An IP designer reuses a shell and develops the IP HW and IP driver for accelerator users. As a result, accelerator users are allowed to utilize accelerators on hCODE platform without caring much HW details. With a clear division of responsibility, the three roles on hCODE platform can collaborate effectively to build large SW and HW co-design projects.

4.2 FPGA Virtualization

In order to efficiently use an FPGA-enabled cluster, hCODE supports three FPGA utilization modes by designing corresponding shells: the non-virtualization mode, the PR-virtualization mode, and the static-virtualization mode. The pros and cons of each mode are summarized in Table 2. In the traditional non-virtualization mode, the only accelerator occupies the entire FPGA's resource in a straightforward way. Next, we mainly explain two virtualization modes in detail.

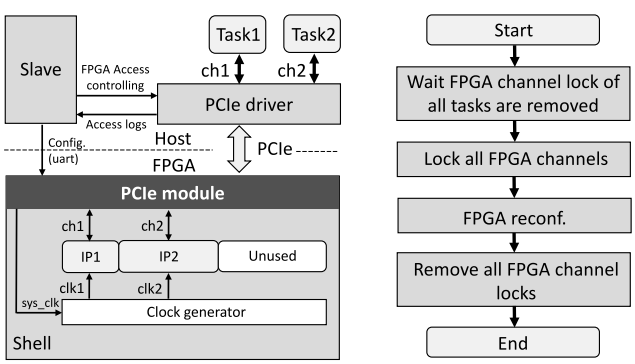
4.2.1 PR-Virtualization Mode

An obvious FPGA virtualization approach that employed by most related researches is to use the partial reconfiguration technique [2]–[4], which allows specified regions on an FPGA to be reconfigured separately and is supported by most modern FPGAs. However, the sizes of pre-defined PR regions cannot be changed dynamically, which means that when a new accelerator's size is too large for any of the PR regions, it cannot be implemented. In contrary, when the new accelerator's size is much smaller than any of the PR regions, FPGA resources will be wasted. This mode will be efficient if all IPs to be implemented in an application is determined, and we can analysis resources requested by them to find the best numbers, sizes and locations for pre-defined PR regions.

In hCODE, we built multi-channel shells to implement multiple accelerators on one FPGA. Figure 2(a) shows a slave node running multiple accelerators. First, we have provided multi-channel PCIe shells in the hCODE repository. In contrast to a single-channel shell, a multi-channel shell comprises a PCIe module with multiple independent communication channels enabled. For each IP, an independent clock domain FIFO is used for the connection with the PCIe module, and a separate clock is inputted from a clock gen-

Table 2 Comparison between FPGA utilization modes.

	Pros	Cons
Non	- No limitation for IP impl.	- Only one IP (low utilization rate)
PR	- Multiple IPs - Independent IP update - Shorter compile time	- PR-regions are fixed - Require PR support from cloud provider
Static	- Multiple IPs - No region limitation for IPs - Applicable on present shells of cloud providers	- Require full reconfiguration (FPGA stops & lose states) - Longer compile time



(a) Slave node (b) Full reconfiguration flow

Fig. 2 The slave node and the full reconfiguration flow.

A shell SPEC file

```
{
 "name": "shell-vc707-xillybus-ap_fifo128-pr-2ch",
 ...
 "hardware": {
   "board": "vc707",
   "device": "xc7vx485tffg1761-2"
 },

 "interface": {
   "host": {
     "bandwidth": 1800,
     "ports":{
       "ap_fifo_1": { … }, "ap_fifo_2": { … }, …}}
 },

 "resource": {
   "1" : {"LUT": 116000, "LUTRAM": 48600, "FF":
232000, "BRAM": 410, "DSP": 980},
    "2" : {"LUT":  …}
 },
 "compatible_shell": {
   "shell-vc707-xillybus-ap_fifo128": "2ports PR
version." }
}
```

An IP SPEC file

```
{
 "name": "ip-kvsorter",
 ...
 "shell" : {
  "shell-vc707-xillybus-ap_fifo128": {
   "property": {
    "max_number": 3,
    "acceleration": 2,
    "throughput": 580,
    "resource": {
      "LUT": 20778, "LUTRAM": 3968, "FF":
40414, "BRAM": 128 }
   },
   "ip_conf": {
    "clk": 150, "fifo_depth": 500, "size": 64
   },
   "shell_conf": {
    "clk": 150,
    "reference": "…"
   }
  },
  "shell-kc705-xillybus-ap_fifo128": {…}
  …
}}
```

Fig. 3 hCODE SPEC file examples.

erator.

The PR-virtualization mode is implemented by providing multi-channel shells with PR regions. First, a shell designer implements a multi-channel shell with PR-regions, and describes the quantity of available resources of each region in the *resource* section in the shell SPEC file, as shown in Fig. 3. On the other hand, an accelerator user can get an accelerator project of an IP on a specified region by executing *ip get* with parameters of IP name, IP version, shell name and PR region id. The hCODE also checks feasibility by comparing available resources of this PR region from shell project and requested resource from IP project. At last, *ip make* can be used to compile the accelerator project and generate bitstream, and *fpga program* can be used to configure the target FPGA in cluster.

4.2.2 Static-Virtualization Mode

In addition to non-virtualization mode and PR-virtualization mode, we proposed static-virtualization mode, which packs multiple accelerators in one project and generates bitstream without PR. This method can improve FPGA utilization rate in several scenarios that PR-virtualization cannot handle. First, present public cloud providers like AWS, implement custom logics in their shell logic's PR region so that PR customization is not applicable for users. Second, PR-virtualization is not always efficient because pre-defined PR-regions are fixed. For these scenarios, static-virtualization that does not use PR or limit resource region can be a choice.

However, static-virtualization requires a longer compile time than PR-virtualization when FPGA resources are sufficient, because the entire FPGA has to be redesigned. In addition, a full reconfiguration is required on the target FPGA, which means all operating accelerators have to be stopped and state will be lost after reconfiguration. For applications that all IPs and accelerator scheduling are determined, the IP packing is performed once during the precompilation stage, therefore the compile time will not be a cost at runtime. In order to perform a full reconfiguration without disturbing operating accelerators, a lock based reconfiguration flow is used, as shown in Fig. 2(b). For the reconfiguration time overhead, if the reconfiguration does not happen frequently (e.g., less frequently than several times per minute), this time overhead will not be significant for most long execution-time tasks that normally last for tens of minutes to hours. For accelerator having states, a state backup and restore mechanism should be implemented by accelerator designer.

The implementation of a static-virtualization shell is similar to an PR-virtualization shell, only without the PR process. Because there is no PR-region limitation for IPs, the *resource* section of shell SPEC file only describes total available resources like a non-virtualization shell. When performing the feasibility check, hCODE sums up all requested IPs' resources and then compares with total available resources of the shell. The IP and shell matching of the static-virtualization is implemented according to the hCODE compatible-shell mechanism. By adding new static-virtualization shells to the hCODE repository, and describing their compatibility relationship to the original single-channel shell, so that hCODE can automatically find the proper shell to use when multiple IPs are requested. Please note that different FPGA utilization modes implemented in hCODE do not require any changes of IPs. The virtualization of implementing multiple IPs on one FPGA can be realized by only utilizing hCODE commands, which significantly reduces the complexity of FPGA virtualization.

5. Accelerator Repository Implementation

In this section, we discuss implementation of the hCODE accelerator repository. The proposed accelerator repository connects shell designers, IP designers and accelerator users by organizing shells and IPs projects in one place. We also provide convenient tools for platform users to easily access repository and generate accelerator.

5.1 Accelerators Repository

The hCODE repository and command line tool is developed based on CocoaPods [6], which is a modern dependency manager for Objective-C and Swift projects. We extended the project specification (SPEC) file format in order to describe HW and appended functions to support Shell-and-IP design pattern.

A JSON format SPEC file with the name *hcode.spec* is required to describe a HW project. Figure 3 shows SPEC file example parts for a shell and an IP design. Basic information sections such as the design name, project type, author, summary, license, source for the two type of designs are the same for both shell and IP projects. In a shell SPEC file, the *hardware* section provides target board and FPGA device information. The *interface* defines ports and bandwidth between shell and IPs. The *resource* section describes available resources of custom logic regions for IPs. The *compatible_shell* section is used to implement IP portability and FPGA virtualization on hCODE platform. If an IP can be

implemented on a specific shell, it is considered to be implementable on its compatible shells of different devices or different number of channels too. On the other hand, in an IP SPEC file, implementation information of this IP on different shells are provided in the *shell* section. For FPGA virtualization and accelerator scheduling, the shell SPEC file describes properties of multiple channels in *interface*, which are used to implement multiple IPs on one FPGA. In order to achieve best performance of an IP on a target shell, the IP SPEC file has a *max_number* section to tell hCODE the ideal number of IPs for implementation.

There is a central hCODE SPEC repository on Github that holds public information for all of the projects in the hCODE platform, as shown in Fig. 4. The folders in this repository are organized in a structure consisting of project name, project version, and project specification file. This repository is synchronized to local when a user executing *hcode setup* command. By indexing this repository, hCODE can perform project searching, downloading, and Shell-and-IP matching for platform users. Moreover, developers can also add private repository with hCODE *hcode repo* command.

Shell and IP designers are allowed publishing their projects on the hCODE platform by submitting SPEC files to hCODE repository. This flow is shown in Fig. 4. First, a developer has to fork the SPEC repository from the hCODE master branch to a private repository. Second, the developer creates a sample *hcode.spec* with *hcode spec create* command, makes necessary modifications on it, and pushes entire design into the private repository with *hcode repo push* command. At last, a pull request to the hCODE master repository has to be sent on Github. In order to ensure the platform quality, we will evaluate developers' requests. After confirmation, the project is merged and becomes public to all hCODE users.

5.2 Accelerator Generation

Accelerator users can easily derive accelerators by using hCODE *ip get* and *ip make* commands. The *ip get* command reads a list of names of requested IPs from arguments or a configuration file. For example, when the command *hcode ip get ip-kvsorter ip-kvsorter ip-kvsorter ip-aes128* is executed, hCODE will first check whether there is a matched bitstream in the local cache. If there is no matched bitstream in the cache, then hCODE will download *ip-kvsorter* and *ip-aes128* as well as a 4-channel shell that compatible with current FPGA HW to the current directory. Meanwhile, a configuration file named *hcode*, which describes the relationship between channels and IPs, will be generated. With this file, the user can execute the *hcode ip make* command to build the IPs, integrate the IPs into the shell project, and finally generate the bitstream by calling an FPGA design tool, such as Xilinx Vivado or Altera Quartus. Because hCODE only defines the project directory structure and the script command interface, this flow is compatible with various design tools. At last, *fpga program* command can be used to download generated bitstream to the local or a remote FPGA. Therefore, the IP portability and FPGA resource virtualization can be realized by utilizing only hCODE commands, which significantly reduces the complexity of FPGA virtualization.

6. FPGA Deployment at Cluster Scale

In this section, we introduce management of FPGA-enabled cluster with hCODE, which is the core function that implementing the FaaS for accelerator users. The main structure of an hCODE-based FPGA cluster is shown in Fig. 5. The master node reads a user's request for accelerators, performs scheduling, generates bitstreams, programs FPGAs on slave nodes and returns accelerator accessing method to user. The slave nodes report FPGA status to the master node, perform FPGA configuration and control FPGA access requests from SW tasks.

6.1 Accelerator Scheduling

The hCODE implements a resource tracking mechanism, a scheduler and a task execution controller for accelerator scheduling.

In order to quantify the FPGA resource utilization and bandwidth for scheduling, all shell projects have the total available resource and interface maximum bandwidth information in their SPEC files. All IP projects provide the implementation resource and ideal throughput information in their SPEC files, as introduced in Sect. 5.1. With this information, the scheduler can do calculations to combine IPs

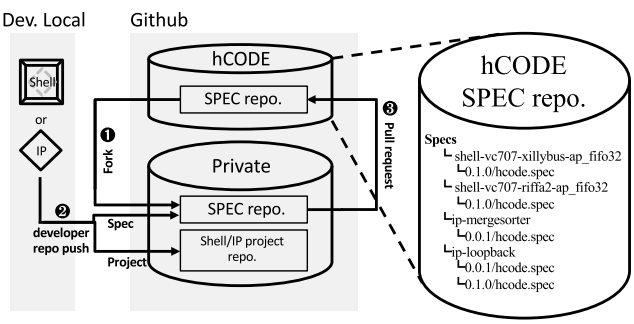


Fig. 4 Repository and design publish flow.

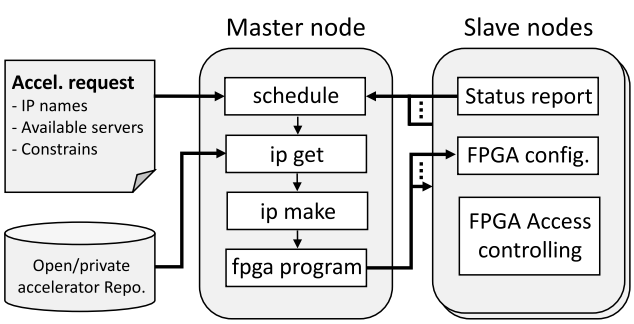


Fig. 5 Cluster-scale hCODE-based FPGA management.

with shells efficiently. In addition, the scheduler has to track all available FPGA resources on cluster. Slave nodes send FPGA boards as well as IP implementation status back to the master node periodically with a status report module, as shown in Fig. 5.

The implemented scheduler employs a policy that takes FPGA resources and bandwidth in consideration. The scheduler reads a list of names of IPs, the available servers and scheduling constrains (e.g., area-first or speed-first) from user's input. In order to generate an allocation solution for the requested IPs on the FPGA-enabled cluster, the proposed scheduler assigns priorities to nodes in the following (decreasing) order: 1) idle servers that have no accelerators on their FPGAs; 2) servers with enough bandwidth and unused FPGA resources; and 3) servers with enough unused FPGA resources. After the allocation solution is generated, the scheduler calls the *ip get*, *ip make*, and *fpga program* commands to generate bitstreams and configure the FPGAs on the slaves. Finally, the user will be notified of the names of allocated servers and the FPGA channels to use in deploying their SW application tasks.

6.2 hCODE with SW Framework

The hCODE reduces SW and HW co-design costs on the following points. First, the hCODE employs the Shell-and-IP design pattern and an accelerator repository to decouple SW and HW designs, as shown in Fig. 1. The interface between SW and HW developments is the driver API provided in each IP project. With this approach, HW designers develop reusable accelerators and SW developers can adopt them in various applications without knowing HW details. Second, accelerators in hCODE are managed in a similar style like modern SW managers, which allows heterogeneous application to be configured and compiled easily. For example, a SW and HW co-designed project only needs to embed a *hcode* configuration file with IPs and shell description, the rest steps for bitstream generation and FPGA configuration can be performed using hCODE commands.

The hCODE only processes the HW scheduling for FPGAs on the cluster. SW tasks execution on the cluster can be deployed manually or by utilizing distribution SW framework such as Hadoop. An example structure of hCODE works with Hadoop YARN is shown in Fig. 6. The hCODE manages HWs on FPGAs and YARN manages SWs on CPUs. An application that request for HW acceleration should send an IP list to resource manager at application master initiation function, then the YARN scheduler passes the same request as well as available servers to hCODE master. The hCODE performs scheduling and configures granted FPGAs, then returns granted server and channel list to the resource manager. At last, the YARN scheduler tags accelerator information on server's labels and performs the final label based scheduling that proposed in [10]. The hCODE has low dependency on the SW scheduling system, therefore it can be easily integrated to current SW platforms.

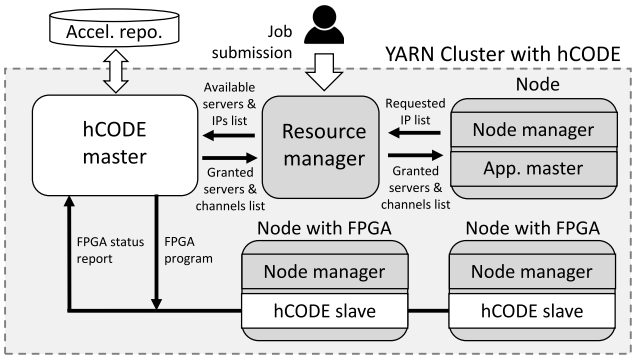


Fig. 6 An example of hCODE works with Hadoop YARN.

7. Case Study

In this section, we give a few case studies to evaluate performance of FPGA virtualization, scheduling results, and discuss the development time improvement of the proposed hCODE.

The experimental environment is composed of a master server and two slave servers. The master server is used for cluster control and the slave servers are used for task processing. The master server has an i7-2600 CPU and 12 GB of memory. The slave servers have Intel Xeon(R) X5690 processors and 96 GB of memory each. One slave server has a VC707 FPGA board attached through a PCIe Gen2 8-lane slot; the other has a KC705 instead. The details of the used IPs and shells are given in Table 3.

7.1 FPGA Resource Utilization

For FPGA virtualization, although the PR technique has advantages, such as updating one region without disturbing accelerators operating in other regions, it is difficult to achieve high resource utilization. Figure 7 shows the FPGA resource utilization comparison between implementations with and without PR. When implementing three 64-way merge-sorters, PR with 4 PR regions left only one available region, which contains around 25% of resources (Fig. 7 (a)). In contrast, the implementation without PR has more available resources that can be allocated to more accelerators (Fig. 7 (b)). Therefore, the PR-based virtualization, which is commonly employed, is not always efficient. The proposed static-virtualization mechanism in hCODE provides better resource utilization for certain scenarios.

7.2 Bandwidth Utilization

In addition to improving on-chip resource utilization, the virtualization method can also improve PCIe communication bandwidth utilization to achieve accelerator performance gains. The top chart in Fig. 8 shows bandwidth utilization for sorting a 64^4 record dataset. Because the sorter IP is 64-way, it takes four loops of SW–HW communication to finish the whole dataset sorting. The average band-

width utilization is only 58.2%. The middle chart in Fig. 8 shows the result for the same calculation when three tasks (three 64^4 record datasets) are run on three IPs concurrently. We find that the average bandwidth utilization increased to 83.0%. This improvement comes from two aspects. First, having three concurrent tasks helps to keep the total throughput stable at the maximum bandwidth. Second, the HW vs. SW processing timing ratio is increased by concurrent execution.

In addition, further performance gains can be achieved with SW and HW co-scheduling. We noticed that the bandwidth is idle when all tasks shift to the SW processing stage together. So, we used the FPGA access controlling mechanism of hCODE slaves to force a 0.2 s (resp., 0.4 s) delay before the start of the second (resp., third) task. The result is shown in the bottom chart in Fig. 8. We can see that the idle time is reduced and the average utilization increased to 89.1%. At present, hCODE only provides basic function on the slave node to support such FPGA access controlling, co-scheduling decision is made by SW developer. We also evaluated a four-IP implementation, but not much further improvement was observed. Therefore, three IPs with the start time of tasks shifted achieves the best performance. Please note, some data points with bandwidth utilization over 100% in the results are because of SW buffers.

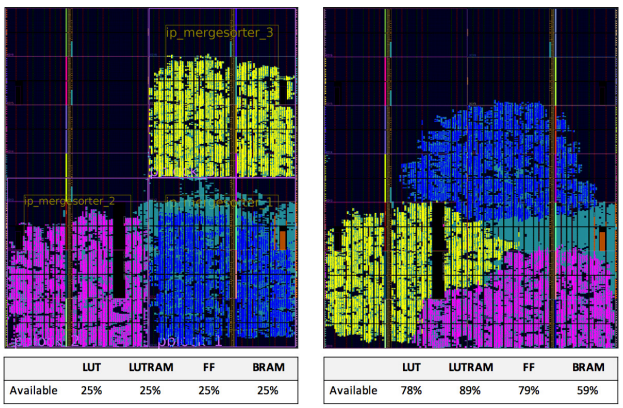


Fig. 7 Resource utilization comparison (three 64-way merge-sorter IPs are implemented).

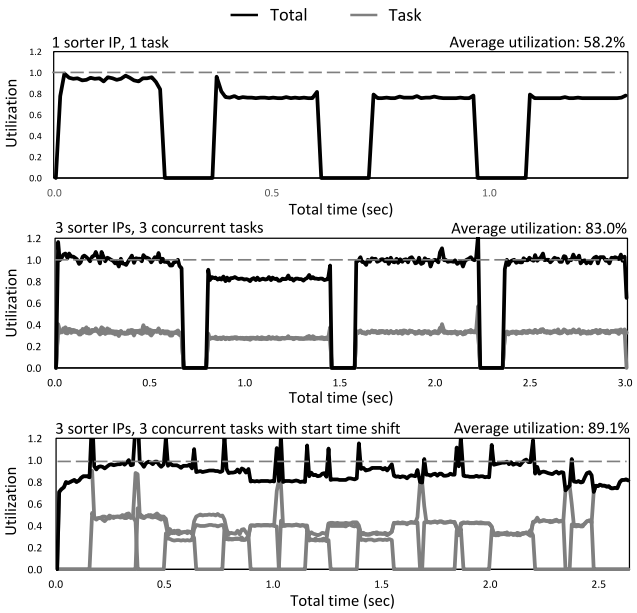


Fig. 8 Bandwidth utilization results.

7.3 Accelerator Scheduling

A sorter IP, an AES IP and a KMeans IP are scheduled on two target servers for the accelerator scheduling evaluation. The implementation details of used IPs and shells are listed in Table 3. The sorter IP is developed with HLS. The AES IP and KMeans IP are ported to hCODE from [11] and [12], respectively. The two servers were attached with different FPGA boards (one with VC707 and the other with KC705) to examine the management of the cluster composed of various HWs. In order to observe how accelerators influence each other clearly, we allocated tasks of similar execution times for IPs.

Figure 9 shows the execution time comparison of different scheduling solutions. The solution $S1$ shows the base-

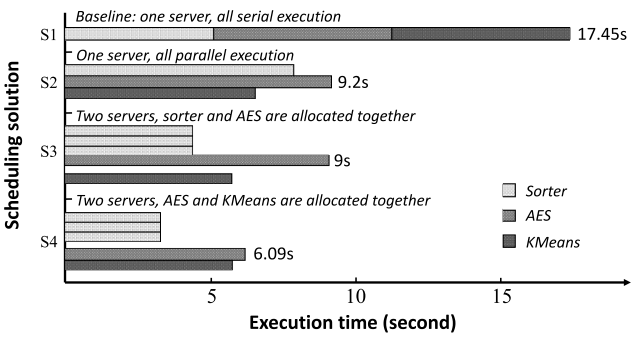


Fig. 9 Execution time of scheduling solutions.

Table 3 Implementation details of used IP and shells from hCODE repository.

	LUT	LUTRAM	FF	BRAM	DSP	Freq.(MHz)	Avg. throughput(GB/s)
ip-kvsorter (64-way merge-sorter)	20,778	3,968	40,414	128	0	150	0.58
ip-aes128	4,700	0	10,230	51.5	0	200	1.4
ip-kmeans	83,841	16,519	43,901	93	120	150	0.03
shell-vc707-xillybus-ap_fifo128	6,852	1,049	6,875	18	0	250	1.8 (Available BW)
shell-vc707-xillybus-ap_fifo128-4ch	12,787	3,548	10,352	45	0	250	1.8 (Available BW)
shell-kc705-xillybus-ap_fifo128-2ch	8,859	1,888	8,092	27	0	250	1.8 (Available BW)
VC707(Available resources)	303,600	130,800	607,200	1030	2800	-	-
KC705(Available resources)	203,800	64,000	407,600	445	840	-	-

line, for which all three IPs are implemented on the VC707-attached server and all tasks are executed serially. The overall execution time of $S1$ is 17.45s. The solution $S2$ uses the same HW as $S1$ but all tasks are executed in parallel. The overall result of $S2$ is 9.2s, which improves 1.9 times than $S1$. However, because of bandwidth conflict, the sorter and the AES tasks are 1.53 and 1.5 times slower than $S1$, respectively. This is acceptable if all tasks are from the same tenant who only cares about the overall execution time. On the other hand, if all tasks are from different tenants, the solution $S2$ will degrade service quality for customers.

In order to achieve higher performance for all tasks, $S3$ and $S4$ utilize two FPGA-attached servers for implementation. The $S3$ shows a scheduling solution only consider FPGA resource utilization. Because VC707 has more on-chip resources, $S3$ allocates three sorter IPs that discussed in previous section and the AES IP on it. The KMeans IP is moved to the other KC705-attached server. We can see that sorting time is reduced while AES time does not changed much because the bandwidth conflict still exists.

At last, the $S4$ shows the result of a scheduling solution implemented in hCODE, which takes bandwidth and resource utilization in consideration. This scheduler first divides high bandwidth IPs onto different FPGAs and then check the FPGA utilization. If the solution is implementable on target FPGAs then return it to the user, otherwise continue to try new solutions. In this case, because both the sorters and the KMeans request for large area, and both the sorters and the AES request for high bandwidth, the $S4$ is the best solution which allocates sorter IPs on the VC707-server, and AES IP and KMeans IP on the KC705-server. This solution improves 2.87 times overall execution times than $S1$ and also providing the best performance of each accelerator.

7.4 Development Time Discussion

Because it is difficult to show a quantitative evaluation on development time improvement with hCODE, we only give a simple discussion here. We successfully completed all evaluations in this section via hCODE commands alone. This shows that IPs on the hCODE platform can be implemented on virtualized FPGAs and deployed on an FPGA-enabled cluster without any change. In addition, any IP can be implemented with improved concurrent performance at a low development cost. Throughput improvement commonly requires a large amount of effort in IP design. However, with the help of hCODE, we can easily achieve high throughput by implementing multiple accelerators on one FPGA and using them concurrently, which requires no changes of IPs as everything is done by hCODE.

8. Conclusion

Today, most public cloud computing providers are actively using FPGAs to improve the energy-efficiency of cloud infrastructures. However, the current solution, such as the Amazon AWS F1, is still based on low efficiency server level renting. On the other hand, there is no mature open-source solution to build private FPGA-enabled cloud system. In this paper, we have proposed hCODE, to simplify FPGA virtualization and accelerator deployment at a cluster scale. By utilizing hCODE, IP developers can build and deploy accelerators at a cluster scale with low cost. The case study results show higher FPGA utilization rates and better acceleration performance can be achieved easily with hCODE. In future work, we will integrate the proposed hCODE platform with more current cluster systems, such as Apache Spark and OpenStack. We are also going to evaluate future work on public FPGA-enabled clouds.

References

[1] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," ACM Trans. Reconfigurable Technology and Systems, vol.8, no.4, Article No. 22, Oct. 2015.
[2] XILLYBUS Ltd., http://xillybus.com
[3] K. Vipin, S. Shreejith, D. Gunasekera, S.A. Fahmy, and N. Kapre, "System-level FPGA device driver with high-level synthesis support," Proc. 2013 International Conference on Field Programmable Technology(ICFPT), pp.128–135, Dec. 2013.
[4] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp.13–24, June 2014.
[5] http://opencores.org
[6] https://cocoapods.org
[7] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," Proc. 11th ACM Conference on Computing Frontiers, Article No. 3, May 2014.
[8] S. Byma, J.G. Steffan, H. Bannazadeh, A.L. Garcia, and P. Chow, "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," Proc. 22nd International Symposium on Field-Programmable Custom Computing Machines, pp.109–116, May 2014.
[9] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in Hyperscale Data Centers," 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), pp.1078–1086, Aug. 2015.
[10] M. Huang, D. Wu, C.H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," Proc. Seventh ACM Symposium on Cloud Computing, pp.456–469, Oct. 2016.
[11] A. Salah, http://opencores.org/project,aes-128_pipelined_encryption, overview, June 2016.
[12] F. Winterstein, S. Bayliss, and G.A. Constantinides, "High-level synthesis of dynamic data structures: a case study using Vivado HLS," Proc. International Conference on Field Programmable Technology, pp.362–365, Dec. 2013.

Qian Zhao received his B.E. degree in the College of Automation and Electronic Engineering from Qingdao University of Science and Technology, China, in 2007. Further, he received his M.E. and D.E. degrees in Computer Science and Electrical Engineering from Kumamoto University in 2011 and 2014, respectively. He is now a postdoctoral researcher at Kumamoto University. His research interests include architecture and design methods of reconfigurable computing systems. He is a member of IEICE.

Motoki Amagasaki received the B.E. and M.E., degrees in Control Engineering and Science from Kyushu Institute of Technology, Japan in 2000, 2002, respectively. He was a design engineer at NEC Micro Systems Co., Ltd. from 2002 to 2005. He received the D.E. degree from Kumamoto University, Japan, in 2007. He has been an assistant professor in the Department of Computer Science at Kumamoto University until 2007. He has been a assistant professor in the Faculty of Advanced Science and Technology at Kumamoto University since 2016. His research interests reconfigurable system and VLSI design. He is a member of the IPSJ, the IEICE and the IEEE.

Masahiro Iida received his B.E. degree in Electronic Engineering from Tokyo Denki University in 1988. He was a research engineer at Mitsubishi Electric Engineering Co., Ltd. from 1988 to 2003. He received his M.E. degree in Computer Science from Kyushu Institute of Technology in 1997. Further, he received his D.E. degree from Kumamoto University, Japan, in 2002. He was an associate professor at Kumamoto University until 2015, and during 2002–2005, he held an additional post as a researcher at PRESTO, Japan Science and Technology Corporation (JST). He has been a professor in the Faculty of Advanced Science and Technology at Kumamoto University since January 2016. His current research interests include high-performance low-power computer architectures, FPGA computing, VLSI devices and design methodology. He is a senior member of the IPSJ and the IEICE, and a member of IEEE.

Morihiro Kuga received his B.E. degree in Electronics from Fukuoka University in 1987 and M.E. and D.Eng. degrees in Information Systems from Kyushu University in 1989 and 1992. From 1992 to 1998, he was a lecturer at the center for Microelectronic Systems, Kyushu Institute of Technology. He has been an associate professor of computer science at Kumamoto University since 1998. His research interests include parallel processing, computer architecture, reconfigurable system, and VLSI system design. He is a member of IPSJ and IEICE.

Toshinori Sueyoshi received the B.E. and M.E. degrees in Computer Science and Communication Engineering from Kyushu University in 1976 and 1978 respectively. From 1978 to 1987, he was a research associate at Kyushu University, where he received D.E. degree in 1986. From 1987 to 1989, he was an associate professor of Information System at Kyushu University. From 1989 to 1997, he was an associate professor of Artificial Intelligence at Kyushu Institute of Technology. Since 1997 he has been a professor of Computer Science at Kumamoto University. His primary research interests include computer architecture, reconfigurable computing, parallel processing. He served as Chair of the Technical Committee on Reconfigurable Systems of the IEICE, Chair of the Technical Committee on Computer Systems of the IEICE, Chair of the IEEE Computer Society Fukuoka Chapter, Chair of the IEEE CAS Society Fukuoka Chapter, and Director of the IPSJ Kyushu Chapter. Currently he also serves as Director-Elect of the IEICE Kyushu Chapter. He is a fellow of IEICE and a senior member of IPSJ.