Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center

Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh,
Alberto Leon-Garcia, Paul Chow
University of Toronto
{naif.tarafdar, t.lin}@mail.utoronto.ca, efukuda@ece.utoronto.ca,
{hadi.bannazadeh, alberto.leongarcia}@utoronto.ca, pc@eecg.toronto.edu

ABSTRACT

We present a framework for creating network FPGA clusters in a heterogeneous cloud data center. The FPGA clusters are created using a logical kernel description describing how a group of FPGA kernels are to be connected (independent of which FPGA these kernels are on), and an FPGA mapping file. The kernels within a cluster can be replicated with simple directives within this framework. The FPGAs can communicate to any other network device in the data center, including CPUs, GPUs, and IoT devices (such as sensors). This heterogeneous cloud manages these devices with the use of OpenStack. We observe that our infrastructure is limited due to the physical infrastructure such as the 1 Gb Ethernet connection. Our framework however can be ported to other physical infrastructures. We tested our infrastructure with a database acceleration application. This application was replicated six times across three FPGAs within our cluster and we observed a throughput increase of six times as this scaled linearly. Our framework generates the OpenStack calls needed to reserve the compute devices, creates the network connections (and retrieve MAC addresses), generate the bitstreams, programs the devices, and configure the devices with the appropriate MAC addresses, creating a ready-to-use network device that can interact with any other network device in the data center.

1. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) have recently proven to be a good computation alternative in data centers due to their compute capabilities and power efficiency. One example is the Microsoft Catapult project where FPGAs are deployed in the Bing search engine [1]. With a 10% power increase they are able to see a 95% performance increase. FPGAs allow users to create customized circuitry for their application. The performance and power-savings multiply at a data center scale. Provisioning FPGA resources from a shared cloud similar to the provisioning of CPUs can be very useful to allow users to create their own FPGA computing clusters. Related works have investigated the provisioning of a single FPGA tightly coupled with a CPU [2], or the creation of static FPGA clusters within the data center [3]. The provisioning of scalable and elastic FPGA clusters from a large general heterogeneous pool of devices has yet to be investigated, or at least disclosed if it is being done in any commercial systems.

In this work, our goal is to provide an easy way to orchestrate large FPGA clusters for large multi-FPGA and heterogeneous applications. We want the user to be able to make a heterogeneous cluster with the ratio of devices (e.g CPUs to FPGAs) and cluster size they require. We also wish to make the deployment and deallocation of these clusters from the cloud quickly, on the order of seconds. Dynamic cluster topologies will be created using the data center network. Our target is to provide the user an easy framework for building a large multi-FPGA application. An example can be a database query application where the query is divided into several sub-queries. Our goal is to provide the framework for the user to describe the query in terms of their query processing engines at a logical level, and then with our framework the user can place these query processing engines on multiple FPGAs and replicate the query many times.

We abstract away the FPGA connections and present the user with a large uniform platform that sits on top of a multi-FPGA backbone. These network FPGA clusters are seen as any other network device in the data center and with the appropriate network addressing information we can send and receive data from any other device in the data center (virtual CPU, other FPGA clusters etc.). Using our prototype system we do a detailed analysis of the whole virtualization stack: starting from mapping FPGA kernels to devices, using the cloud management software OpenStack to provision FPGAs and connect their network ports, map multi-FPGA topologies onto the network by configuring the FPGAs to use the appropriate network addresses, and ways to scale up by replicating nodes and inserting schedulers to communicate with the replicated nodes. Our system abstracts away the aforementioned stack and produces the cluster based on a description of the kernels and how they are connected, and which FPGAs these kernels will map to.

The rest of the paper is as follows: Section 2 will look into related work in FPGA virtualization and cloud cluster management tools, Section 3 will describe our data center resources, Section 4 will explain our design justifications, Section 5 describe the infrastructure we provide, Section 6 explores scalability of our infrastructure, Section 7 evalu-

FPGA'17, February 22–24, 2017, Monterey, California, USA.
 ISBN 978-1-4503-4354-1/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021742.

ates our infrastructure and lastly we conclude the paper in Section 8.

2. RELATED WORK

In this section we describe previous work in virtualized FPGAs and other Cluster Management Tools in the cloud.

2.1 FPGA Virtualization and Clouds

There has been previous academic work providing FPGAs as virtualized resources within the cloud management tool OpenStack. The work presented by Byma et al. proposes FPGA resources sitting directly on the network to be allocated as OpenStack resources [4]. The hypervisor is programmed into hardware and communicates to the OpenStack controller via the network. Furthermore, the FPGA application region is split into four smaller regions allowing multiple users to share a single FPGA device. This requires modifying OpenStack to communicate to the hardware hypervisor in the FPGA.

The work proposed by Chen et al. also virtualizes FPGAs in OpenStack but does not consider FPGAs sitting directly on the network [5]. They implement the hypervisor in software by modifying KVM, which is a popular Linux hypervisor [6]. Instead, the FPGAs are coupled with a virtual machine. Similar to the previous work, this also requires modifying OpenStack to communicate to the software hypervisor.

Several industrial pursuits have started investigating provisioning FPGA resources from a cloud. One example is the Maxeler MPC-X project [7]. This project provides a virtualized FPGA resource to a user that can be implemented with a variable number of FPGAs. The user first allocates resources for the given cluster of FPGAs in the virtualized FPGA resource. Once the cluster has been made, the details are abstracted from the user during application run-time.

IBM's SuperVessel looks at providing an FPGA as a PCIe connected cloud resource with a CPU also provisioned with OpenStack [2]. In this model a single FPGA is provisioned to the user as an accelerator to which the user can upload FPGA code to be run and compiled onto the FPGA. This simplifies the process of provisioning an FPGA, but only with a single FPGA.

Microsoft has also continued their work with data center FPGAs with the second iteration of Catapult [3]. The FPGAs are connected directly to a host CPU via a PCIe link. The output of the host NIC connects to the FPGA and the FPGA connects to the network through a high-performance switch. Thus the FPGA has a direct network connection and the CPU accesses the network through the FPGA. FPGAs can communicate amongst each other through a low-overhead custom transport layer. The FPGAs in this environment are not provisioned as a part of a cloud service for external users, and are used internally within the Mircrosoft data center.

2.2 Cloud Cluster Management Tools

A key aspect of this project is to provide orchestration of clusters within our cloud environment. Heat [8] is a component in OpenStack that can orchestrate clusters using an orchestration template, which describes the virtual machines and networking within the desired cluster. This allows the creation of interesting network topologies within a cluster. Heat can be combined with user applications that can modify these clusters using other metrics such as performance, resource utilization, and CPU usage.

Other tools exist that combine orchestration and load balancing using the aforementioned metrics. The usual workflow for these tools are as follows. The tool first reserves a set of resources from a larger pool of compute nodes for a certain application. The allocated resources are then connected for the application and monitoring. The monitoring is used for user statistics as well as fault tolerance within the cluster.

These tools are helpful for getting reliable performance on a cluster as well as debugging a cluster. Debugging a cluster can be a daunting task as there are many variables within the cluster. These tools monitor events to gauge the status of different processes within an application and present the problem to the user in an easy to understand representation.

Most of these tools currently work for CPU clusters (e.g Apache Mesos [9]) and GPU clusters (e.g NVidia Management Library) [10]. Our challenge is to extend clustering capabilities to FPGAs by developing our own orchestration tool and then to investigate monitoring and updating our clusters using FPGA metrics, which will differ from the current set of CPU and GPU metrics current tools use.

3. BACKGROUND

This section describes the infrastructure underlying our platform, including the SAVI data center and SDAccel FPGA Programming Platform.

3.1 Smart Applications on Virtualized Infrastructure (SAVI) Testbed

The SAVI testbed is a Canada-wide multi-tier heterogeneous testbed [11]. Figure 1 shows the architecture of SAVI. This testbed contains various heterogeneous resources such as FPGAs, GPUs, Network Processors, IOT sensors and CPU-based servers. The virtualization of these resources is one focus of the research on SAVI, where our work focuses on the use of FPGAs. Resources other than CPUs such as GPUs and network processors, are provisioned to the user either by providing the entire server without virtualization or with the use of PCIe-passthrough.

The multi-tier property refers to the network architecture of SAVI. SAVI can be seen as multiple cloud networks. The Core consists of a large number of CPUs that provide the main compute resources of the data center. The Core is then connected to several Edges dispersed around Canada. Each of these Edges is a smaller cloud network that also contains the heterogeneous devices. Many of these heterogeneous devices are connected directly to the network through high performance 10 GbE switches. These devices are treated the same way any CPU would be treated as many of these devices are assigned network ports with valid MAC and IP addresses. These devices are addressable by any other node (CPU or other device) on the network, once they are registered to the network. This allows, for example, an IoT sensor in Toronto to send data to an FPGA cluster in Victoria and then have the data be accessible by a CPU cluster in Calgary. Furthermore the multi-tier architecture allows a lot of the processing to be done on the Edge network close to the heterogeneous devices before being sent to the Core where there are more compute resources.

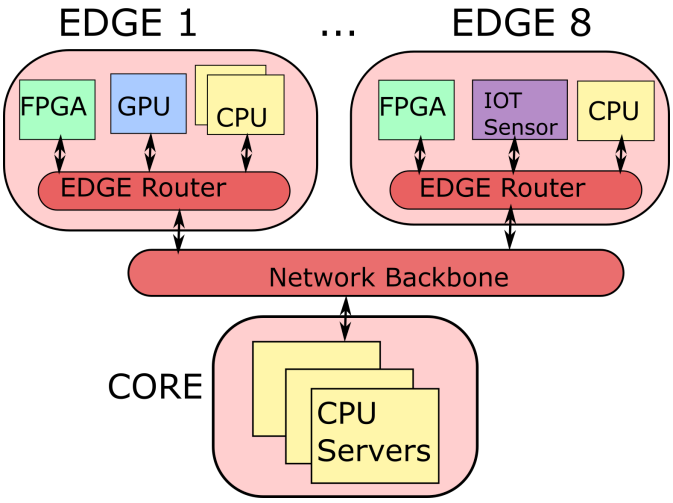


Figure 1: System diagram of the SAVI multi-tier architecture that has a CORE with many CPU Compute Servers and Edges physically dispersed around Canada. Each Edge is made up of compute CPUs and other heterogeneous devices (e.g FPGAs, GPUs, IOT Sensors).

3.1.1 OpenStack

OpenStack is the cloud management tool used by SAVI [8]. OpenStack is divided into several services. The two main services that we employ in our platform are Nova and Neutron, and these services are typically interfaced with a client machine. Nova is responsible for the deployment of compute infrastructure from the platform. This involves the generation of virtual machines on physical machines. The client machine specifies two fields when requesting a virtual machine: a software image, and the flavor. The software image refers to all the software that is to be installed on the virtual machine, including the operating system and any other applications that we want initialized on our virtual machine. These images are typically kept in a repository and can be updated by users of the testbed. The flavor refers to the physical specifications of the virtual machine, such as number the of CPU cores, RAM, and hard drive space.

Neutron is responsible for the provisioning of network resources. We can create network ports within our cluster, and these ports are assigned MAC addresses and IP addresses that will be valid within the cluster. When creating virtual machines, these ports are created implicitly, but we can explicitly create additional ports for non-virtual devices or non-CPU devices.

3.2 FPGA Hypervisor

In our design we use the Xilinx SDAccel [12] platform as an FPGA hypervisor, where the hypervisor is used to provide some basic services. The FPGA in this model is a PCIe-connected device and the platform first provides a driver to communicate to the FPGA. This is done through OpenCL, which provides the API to communicate to and manage devices.

OpenCL is both a programming language for heterogeneous devices and a programming API for a host application (conventionally run on a CPU) to manage and communicate to OpenCL compatible devices [13]. This environment gathers all the OpenCL devices connected to the processor usually locally via PCIe. In the SDAccel Platform, as shown in Figure 2, the OpenCL API communicates to a driver provided by Xilinx called the Hardware Abstraction Layer (HAL) that provides driver calls to send/receive data from the FPGA and program the Application Region, in the FPGA. The Application Region is programmed using partial reconfiguration, and the region around the Application Region is the Hypervisor in our model. In this platform the kernels within the Application Region can be OpenCL kernels, Vivado HLS kernels, or even hand-coded Verilog/VHDL kernels.

The PCIe Module is a master to a DMA engine to read/write to off-chip DRAM. This is used to communicate data to the Application Region. The PCIe Module is also a master to an ICAP module (not shown) responsible for programming the Partial Reconfig region with a bitstream sent from the user in software. The HAL driver provides an API that abstracts away the addresses required to control the various slaves of the PCIe master.

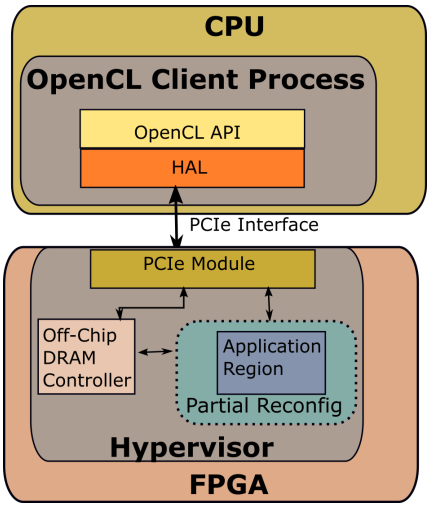


Figure 2: System diagram of the SDAccel platform.

For our cluster model we modified this base platform by adding a 1 GbE Ethernet core. For simplicity we do not use partial reconfiguration for the Application Region choosing instead to synthesize the application with the Hypervisor as a single design. This makes the programming flow simpler as there is only one bitstream to synthesize. The low-speed Ethernet core and the absence of a partially reconfigurable Application Region are limitations of our current modified version of the platform and will be addressed in the future.

4. DESIGN JUSTIFICATION

Our system looks at the creation of multi-FPGA clusters from a high-level kernel description. Our system can be broken down according to the infrastructure stack shown in Figure 3. We provide a high-level of abstraction where a user describes logical clusters independent of FPGA mappings, and our system provisions the cluster from a pool of resources accordingly. We take this approach as this is quite analogous to the modern FPGA system design approach that can be seen in Xilinx Vivado IP Integrator [14] or Altera Quartus QSys [15]. These tools are used to con-

nect modules together to create larger systems. Our system provides a similar design space but the space is across multiple FPGAs. The interfaces between FPGAs are abstracted from the user, where the user's logical cluster description has no notion of which FPGA each user module will map to. We wish to provide a familiar environment to large system designers while providing abstractions to provide ways to scale to large multi-FPGA designs. Our system also provides ways to scale up nodes within a cluster by replicating those nodes, or entire clusters with the use of a directive.

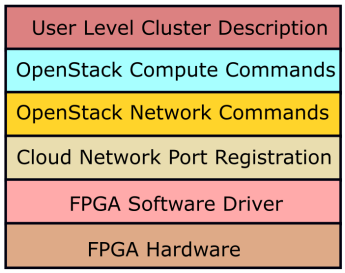


Figure 3: Infrastructure stack, most of this is abstracted from the user.

The multi-FPGA design cluster is created on the fly from a pool of resources in SAVI using OpenStack. Our system first builds single FPGA nodes, and then connects the network ports on the FPGAs in topologies that are determined from the logical cluster description the user provides. OpenStack is used to provision the single FPGA nodes, and to acquire network ports for the FPGAs in the SAVI data center. We use OpenStack because this is an open-source cloud management tool that is adopted in large data centers such as SAVI. Our placement in SAVI gives us access to the multi-tier infrastructure with the large pool of heterogeneous resources, which can be used to create large-scale heterogeneous applications.

5. INFRASTRUCTURE OVERVIEW

In this section we describe our infrastructure by examining each part of the infrastructure stack. First we will look at how a single FPGA is provisioned, then we will look at how clusters are provisioned and scaled.

5.1 Single FPGA Environment

The first step is to be able to provide an FPGA within a virtual machine. PCIe passthrough gives a virtual machine full access to a subset of the PCIe devices within a physical compute node. This can allow us to have multiple virtual machines on top of the physical machine able to access different PCIe devices. Figure 4 shows two virtual machines attached to PCIe devices using PCIe passthrough.

In our environment we use the Alpha Data ADM-PCIe-7V3 cards, which have a Virtex 7 FPGA. On these FPGAs, the Xilinx SDAccel static bitstream (the bitstream describing the Hypervisor) is programmed onto the flash memory. In our environment all the FPGAs are programmed with the same Hypervisor bitstream as OpenStack running on the server uses the PCIe Module connection to determine the type of PCIe-connected device. This PCIe Module might not be consistent across different FPGA Hypervisors and will not be recognized as the same device to OpenStack. This

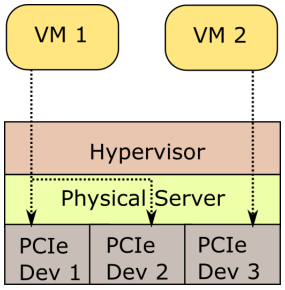


Figure 4: This figure shows PCIe passthrough. Two VMs have direct access to PCIe devices. The Hypervisor grants the first virtual machine full access to the first two PCIe devices and the second virtual machine has the third PCIe device.

static bitstream includes the PCIe Module that is detected by the physical compute node, and then registered with OpenStack Nova by including it in its PCIe passthrough White-List, which specifies the devices that can be attached to virtual machines.

The last step for the setup is the creation of a new Flavor, which defines a set of specifications for virtual machines. A Flavor can specify the type of PCIe device as well as the number of PCIe devices of that specific type (e.g there can be Flavors for for one FPGA, two FPGAs, one GPU, two GPUs etc.). In addition to PCIe devices, a Flavor also defines other machine specifications such as memory size and hard-disk space. Once these Flavors are created, they can be used to create multiple virtual machines described by the specifications of the Flavor.

We have been using this environment to provide an SDAccel service since Fall 2015, before IBM started to offer it in April 2016. We have Flavors that have simulated FPGAs and physical FPGAs.

5.2 Multi-FPGA Infrastructure

Our infrastructure provides a cluster of network-connected FPGAs to the user given a description of what a cluster of kernels will look like. The user provides a logical description of their desired FPGA cluster that describes how different FPGA kernels are to be connected together. The user also provides an FPGA mapping that specifies the number of FPGAs the user requires and places the kernels on the appropriate FPGAs. Kernel connections within a single FPGA are simple as they are directly connected, whereas kernel connections between FPGAs are implemented via the network. Furthermore kernels may also fan out to schedulers instead of directly connecting to other kernels.

5.2.1 Logical View of Kernels

The kernels in this system are streaming kernels and they use the AXI stream protocol for input and output. The AXI stream interface our system uses has the following fields: A 32-bit data field, 8-bit dest field, a 1-bit last field, a 1-bit ready field and a 1-bit valid field.

All kernel inputs to the system are addressed by a specific dest entry. Logically speaking, unless otherwise stated, any kernel output can connect to any input. This can be seen as all kernels being connected to a large logical switch. These kernels may be mapped to the same FPGA or to different

FPGAs. Furthermore these kernels can be replicated with directives in the input scripts and they can be scheduled in different ways with the use of schedulers.

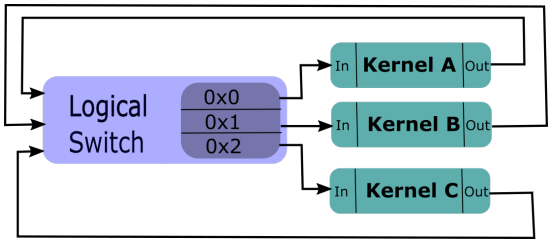


Figure 5: This highlights the simple logical view of a kernel cluster. In this situation all the kernels output to a switch and their inputs come from the switch.

Sub-Clusters

In Figure 5 we show three kernels connected via one logical switch. All kernels are connected to each other in a fully connected network. Edges can be removed if we directly connect kernels. Figure 6 shows four kernels with direct connections between some of the kernels. Such sub-clusters are then connected to the logical switch.

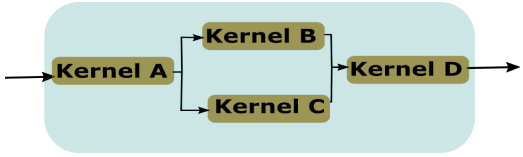


Figure 6: This is an example of a directly connected sub-cluster that would be connected to the logical switch.

We can also have our own schedulers where the output of a kernel might not be connected to all the other kernel inputs but to a subset of kernel inputs arbitrated by a scheduler. This type of sub-cluster is shown in Figure 7 and explained in more detail in Section 6.1.

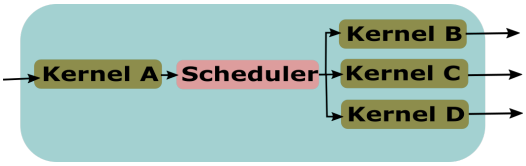


Figure 7: This is an example of a sub-cluster where a kernel is connected to a local scheduler that arbitrates between 3 kernels within the sub-cluster.

5.2.2 *Physical Mapping of the Kernels*

Each kernel in the logical topology is mapped to a physical FPGA. More than one kernel can be mapped to an FPGA. Direct kernel connections on the same FPGA are simply connected within the FPGA. Kernels with connections that cross an FPGA boundary are wrapped with logic to help with the crossing.

When connections on the large logical switch are divided across multiple FPGAs, the logical switch is implemented as physical switches on each of the FPGAs. Figure 5 shows three kernels fully connected with a logical switch. Now let's consider the following scenario: Kernels A and B are on FPGA 1 and Kernel C is on FPGA 2. The physical mapping is shown in Figure 8.

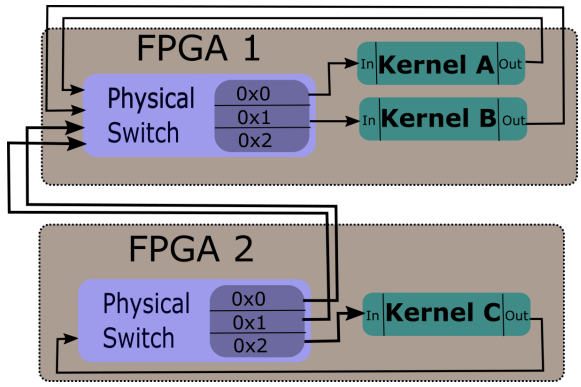


Figure 8: This figure translates the logical cluster shown in Figure 5 into a physical cluster with two FPGAs.

Figure 8 shows the logical switch split into two physical switches. The inputs to the respective kernels on the two FPGAs always come from the physical switch on the FPGA. The first FPGA sends all packets addressed to Kernel C to the switch in the second FPGA. Furthermore the second FPGA's switch sends all packets dedicated for Kernels A and B to the first FPGA. The output of each of the kernels feed into the physical switch on that FPGA. The physical switch can determine the destination FPGA of each packet.

For edges between kernels that are not connected to the large logical switch (sub-clusters), the direct connections must also be facilitated between FPGAs.

5.2.3 *FPGA Application Region*

The FPGA Application region includes helper modules for the User Kernel to interface directly with the network through the Ethernet interface. The helper modules are responsible for filtering packets, formatting packets and arbitrating for the network port. The Application Region is shown in Figure 9.

The configuration bus is used to configure the input and the output modules. These signals are driven by the PCIe Module on the FPGA, which receives signals from the PCIe-connected virtual CPU.

Input Module

All the packets that the FPGA receives via the Ethernet are forwarded to the input module. The packets that are received at the network port follow the Ethernet packet convention with a 14-byte header. On top of this we add our own protocol by appending two bytes (Kernel Address) to specify the destination kernel for the packet, as we may have multiple kernels on the FPGA that are requesting input packets.

Figure 10 shows the protocol details used by our FPGA infrastructure. Each FPGA in our infrastructure is assigned a MAC address within the SAVI infrastructure. The process

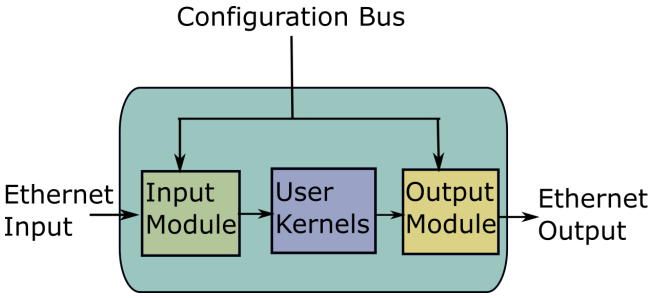


Figure 9: This figure shows the details of the Application Region. The input and output modules are both configured by the configuration bus.

Figure 10: The Ethernet protocol plus our custom protocol to differentiate the kernel.

by which we get the MAC address is discussed in Section 6.2. The destination MAC address should match the MAC address assigned to the particular FPGA. The source MAC address will be the source MAC address of the FPGA or of the virtual machine within SAVI that is sending the FPGA data. The next two bytes, according to the Ethernet frame protocol, are the ether-type that we hardcoded to 0x7400, and the last field is the address of the kernel within the FPGA.

The Input Module consists of an Input Bridge and an Input Demultiplexer. The Input Bridge is configured after the FPGA is programmed with the bitstream and before the application can run. The Input Bridge behaves as both a firewall and converts a packet from an Ethernet Packet into an AXI Stream packet. The Input Bridge's firewall is configured with the MAC address assigned to the FPGA. The Input Bridge also drops the Ethernet header and adds a dest field as part of the AXI stream. The dest field corresponds to the Kernel Address specified within the header. This Input Demultiplexer either outputs to kernels on this FPGA that are expecting Ethernet input, or it outputs to kernels on a different FPGA; in this case all packets matching the corresponding dest field will be sent straight to the output module. The input to the switch comes from both the Ethernet module and all other user kernels that can output to any other kernel on the FPGA. An example of an Input Module is shown in Figure 11. For details refer to Section 5.2.2.

Output Module

This module receives streams from the User Kernels and from the Input Demultiplexer. The Output Module consists of Packet Formatters (PF) and an Output Switch. Each stream (either from the User Kernels or from the Input Module) needs a Packet Formatter before it can be sent out to the network. Each stream is formatted with the appropriate MAC headers. The source MAC address is that of the FPGA. The destination MAC address is of the destination FPGA or virtual machine. The ether-type is 0x7400 as it was in the input stream and then we append the dest of the

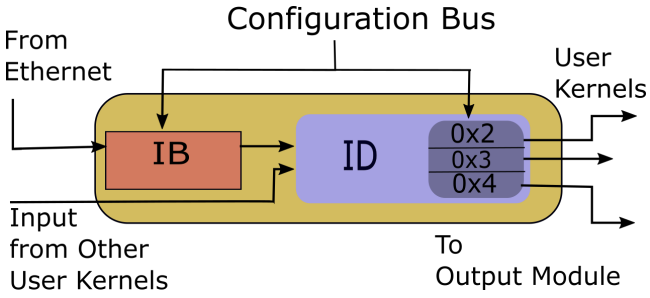


Figure 11: The input module consisting of the Input Bridge (labelled IB) and the Input Demultiplexer (labelled ID). In this example the dest fields 0x2, 0x3 feed into different User Kernels on this FPGA and 0x4 feeds into another FPGA by going through the Output Module.

stream into the header of the packet. All the Packet Formatters are fed into an output-switch that arbitrates using the last field of the AXI stream. The output switch uses a round-robin scheduling algorithm. The output module is shown in Figure 12. The input to the Packet Formatter is an AXI stream with a dest field. The formatter uses the dest field as the kernel address when it is outputting to the network.

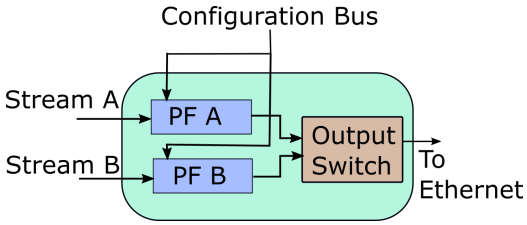


Figure 12: The output module for two streams consisting of Packet Formatters (labelled PF) for each stream that needs to be output.

6. SCALING UP FPGA CLUSTERS

A major feature of our infrastructure is the ability to scale the cluster. We can treat a whole cluster as a single processing unit and it can be replicated with a single directive within the script. For example, Figures 5 and 8 show a logical mapping transformed into a physical mapping without any replication. If this was to be replicated three times there would be a total of six FPGAs. The original FPGA mapping file listed only two FPGAs but six FPGA MAC addresses would be returned to the user.

6.1 FPGA Schedulers

Nodes within the cluster can be replicated as well without replicating the entire cluster. Replicating a node within the cluster will require all nodes that fan-in to that specific node to now include a Scheduler. The Schedulers currently support any-cast, which uses a round-robin scheduler, or broadcast. Figure 13 shows how a node is replicated within a cluster and where a Scheduler is inserted.

The Schedulers are also FPGA kernels. If the replicated kernels span across multiple FPGAs the scheduler will be placed on the FPGA with the most replications of that ker-

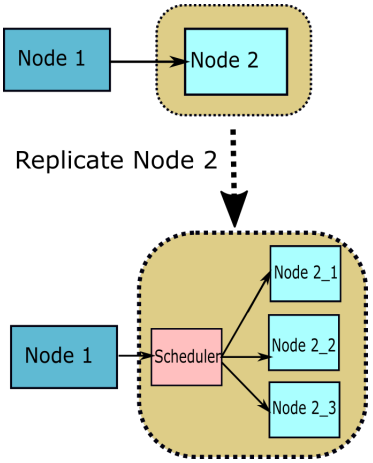


Figure 13: This shows the replication of Node 2. The replicated nodes are Node 2_1, Node 2_2 and Node 2_3. Node 1 has a Scheduler that fans out to the replicated nodes.

nel to reduce latency for the more common case. For example, in Figure 13, if two of three replications are on FPGA 1, and the other is on FPGA 2, then the script will place the Scheduler on FPGA 1. The script will then create connections from the Scheduler to the replicated nodes and one connection to the Output Module on FPGA 1. The remaining replicated kernel will be connected to the Input Module on FPGA 2. Figure 14 illustrates this scenario.

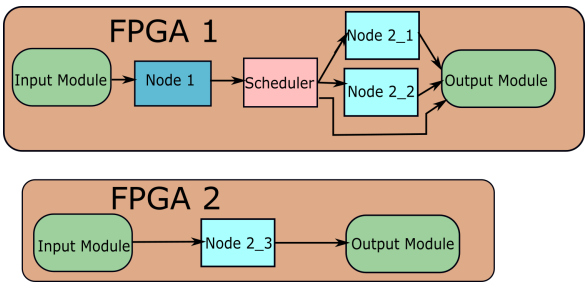


Figure 14: The physical configuration if Node 1, Node 2 _1 and Node 2_2 are on FPGA 1 and Node 2_3 is on FPGA 2.

6.2 FPGA Software Drivers and Network Connections

Each virtual machine with an FPGA is responsible for sending control signals to the FPGA. These control signals are to configure the Input Module and the Output Module with the appropriate MAC addresses. We choose to use the software to configure the Input and Output Modules because the alternative is to encode the MAC addresses in hardware, which will require resynthesizing FPGA bitstreams for different physical FPGAs when replicating the cluster. Our approach allows us the option to generate our cluster with one set of FPGAs and then replicate the clusters with the same bitstreams to more FPGAs.

The software drivers can configure the Input Bridge and the Packet Formatters in the hardware because the PCIe module in the hardware is a master (a driver of signals) to these modules. This means that writing to a certain address on the PCIe module can be used to send data to the Input Bridge or the Packet Formatter. We can write to different addresses of the PCIe module with the HAL driver that was provided in the SDAccel tool kit. When a virtual machine with an FPGA is booted, the software driver is accepting bitstreams. Once a bitstream is received it will be programmed with the HAL and the Input Bridge and Packet Formatters will also be configured by the HAL. Our justification to provide the Packet Formatters as software configurable blocks is due to scalability. If we wish to scale up our cluster with more network FPGAs, the MAC address of each FPGA can be configured by software instead of synthesizing bitstreams on a per FPGA level.

Each FPGA obtains a network connection by first receiving a network port from the OpenStack service, Neutron. Each network port consists of a MAC address, and IP address. This port is then registered with the physical port on the network switch that has the FPGA connection. Our scripts can determine the physical switch port of a particular FPGA connection by observing which physical server hosts the virtual machine containing the PCIe server. In our setup we have one FPGA per physical server. If this were to change we would need a new mechanism to infer the physical network port of a particular FPGA. Once the port returned by Neutron is registered with the physical port, it is now accessible on the network from any other device in the SAVI data center, including other virtual CPUs, IoT devices and FPGA clusters.

6.3 Tool Flow

We summarize the use of our system by describing the tool flow. First the user submits a logical cluster description and FPGA mapping file to a global FPGA parser. Eventually, these could be generated by a higher-level framework or application. OpenStack calls are generated to create virtual machines, which are light-weight CPU virtual machines connected to an FPGA, and one virtual machine dedicated to synthesize bitstreams. Subsequent OpenStack calls are generated to create network ports, each with valid MAC and IP addresses. These ports are registered with the SAVI switch and now all packets sent to these addresses will be forwarded to the right switch port. After all the OpenStack calls are generated, the individual FPGAs are synthesized on the large virtual machine dedicated to synthesizing bitstreams. Once the bitstreams are synthesized they are forwarded to the individual FPGAs to be programmed onto the FPGA. Once programmed, the Packet Formatters are configured by the FPGA software driver running on the light-weight CPU attached to the FPGA via PCIe. After the user submits the initial cluster description files, the rest of the calls are automatically generated by our infrastructure.

7. EVALUATION

This section first examines the overhead of the infrastructure our design introduces and compares it to the SDAccel platform. This will quantify the overhead we introduce to support our multi-FPGA cluster. Then we test the latency and throughput of the input and output modules using a set of microbenchmarks. Finally, we test a full application using a database acceleration application. The designs are implemented on the Alpha Data 7V3 card, which has the following specifications: a Xilinx Virtex 7 XC7VX690TFFG-1157 FPGA (433200 LUTs, 866400 Flip Flops, 1470 BRAM Tiles), two 8GB ECC-SODIMM for memory speeds up to

1333MT/s and Dual SFP+ cages for high speed optical communication including 10 Gigabit Ethernet.

Our network infrastructure connects the 10 GbE SFP ports using 10 GbE to 1 GbE transceivers to a network switch. The switch can support 10 GbE links, but due to the 1 GbE FPGA core that is in our FPGA Hypervisor we have to use a 1 GbE cable. The goal of the evaluation is to demonstrate that our FPGA network modules add little overhead with respect to throughput and very little latency overhead. The absolute latency and throughput numbers are limited by the 1 GbE network connection but the infrastructure we have built can be used on 10 GbE, or better, systems where we would expect these numbers to be better. We also wish to highlight the scalability of our infrastructure with a case study, demonstrating that by simply changing a directive in the script, our clusters can replicate with the throughput scaling accordingly.

7.1 Resource Overhead

The resource overhead from our infrastructure is shown in Table 1. Absolute numbers are given with the percentage of the device total shown in brackets.

Table 1: Resource Overhead of our System

Hardware Setup	LUTS	Flip-Flops	BRAM
SDAccel Base	53346 (12.3 %)	64550 (7.45 %)	228 (15.5 %)
SDAccel Base with Ethernet Support	62344 (14.4 %)	76124 (8.79 %)	228 (15.5 %)
Input Module			
Input Bridge	87 (0.02 %)	170 (0.019 %)	2 (1.36 %)
Input Demultiplexer with 16 outputs	82 (0.019 %)	124 (0.014 %)	0 (0 %)
Output Module			
Ethernet FIFO Controller	26 (0.006 %)	12 (0.014 %)	2 (1.36 %)
Output Switch with 16 inputs	517 (0.119 %)	138 (0.016 %)	0 (0 %)
Packet Formatter (one per network output stream)	230 (0.053 %)	252 (0.029 %)	2 (1.36 %)
Total Available	433200	866400	1470

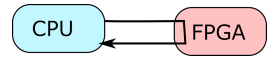
The SDAccel Base refers to the standard SDAccel environment that has no network connection for the FPGA. The SDAccel Base with Ethernet Support includes a 1 Gb Ethernet port. We can see that the addition of the Ethernet port requires only 2.1% of the resources of the whole device. The Input Module is divided into a Input Bridge and the Input Demultiplexer. The size of the Input Bridge is independent of the number of network input streams. The size of the Input Demultiplexer is dependent on the number of streams. Table 1 shows the overhead corresponding to a 16-port switch. The Output Module is divided into the Ethernet FIFO Controller, the Output Switch and the Packet Formatter. The Ethernet FIFO Controller overhead is independent of the number of output streams. The Output Switch size, analagous to the Input Demultiplexer size, is dependent on the number of output streams. The number of Packet Formatters we have on our FPGA is dependent on the number of output streams. It can be seen that the resource usage of the Input Bridge, Input and Output Modules and Packet Formatter is small relative to the device and not significant in terms of resources.

7.2 Microbenchmarks

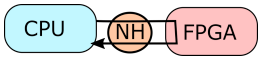
Our microbenchmarks consist of an application that is a direct connection between the Input Module and the Output Module of an Application Region. The goal of this is to show the overhead of our Input and Output Modules and to show that they can handle packets at line-rate as all of the modules are of single-cycle latency.

7.3 Microbenchmark Setup

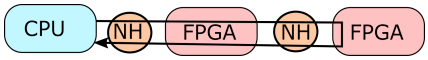
For Microbenchmark 0 the CPU is directly connected to the FPGA. The CPU sends packets to the raw network interface and the FPGA echoes them back. The packets traverse through the Input Module, the Application Region FIFO and exit through the Output Module back into the CPU. The CPU for this data-point is not a virtual machine and the specifications of it are as follows: Intel Xeon 3.5 GHz CPU E5-2637, four cores with hyperthreading, 32 GB RAM.



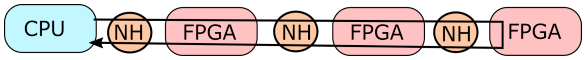
(a) Microbenchmark 0 is a CPU directly connected to an FPGA (not through network switch).



(b) Microbenchmark 1 is a CPU connected with a network switch to an FPGA Chain of length 1.



(c) Microbenchmark 2 is a CPU connected with a network switch to an FPGA Chain of length 2.



(d) Microbenchmark 3 is a CPU connected with a network switch to an FPGA Chain of length 3.

Figure 15: Microbenchmarks 1 to 3 have a network hop (NH). Each network hop travels to the network switch connected to all the FPGAs. Microbenchmark 0 does not use a virtualized CPU, where as the others use virtual CPUs provisioned in SAVI.

This is compared to three microbenchmarks using SAVI. These microbenchmarks consists of one virtual machine sending data to a chain of FPGAs. The chain of FPGAs is either a single FPGA, two FPGAs, or three FPGAs. The traversal through the FPGA chain requires packets to travel to the network switch to be routed. Figure 15 shows the setup of the four microbenchmarks. The specifications of the virtual machine sending data to the FPGA chain are as follows: QEMU Virtual CPU 2.0 GHz, two cores, 4 GB RAM.

7.3.1 Latency

The round-trip latencies are shown in Figure 16. There is no switch latency and no virtualization overhead for Microbenchmark 0. However after that point we notice a linear

progression as we increase FPGAs. Each extra FPGA on the path requires two trips to the switch.

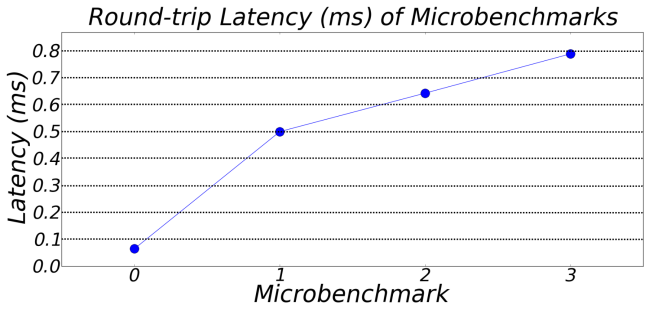


Figure 16: Round-trip Latency of the four microbenchmarks.

7.3.2 Throughput

Figure 17 shows the throughput for the different microbenchmarks. The red line is the bandwidth limit of the network cable. The throughputs of Microbenchmark 0 to 3 are measured with the iperf tool [16]. This is a network tool used to measure throughput of network connections. Microbenchmark 0 does much better because of the faster CPU as it approaches the theoretical maximum of 1 GB/s, which is the current speed of the Ethernet module in the SDAccel framework. When we look at Microbenchmarks 1 to 3, as expected the throughput remains consistent as more FPGAs are added to the chain. Figure 17 shows two additional data points. The first is the throughput between two virtual machines in the SAVI network (Microbenchmark 4). The second additional Microbenchmark is the calculated throughput within the FPGA (Microbenchmark 5). The internal FPGA bandwidth shows that the bottleneck observed is not within the FPGA but due to the virtual machine feeding the FPGA. The internal FPGA throughput is calculated by using the bus width, which is 4-bytes wide and multiplying that by the clock speed, which is 125 MHz. The network switch is designed to switch at 40G rates and therefore is not the bottleneck of our system.

Both the Input and Output Modules work with single-cycle latency. The Input Module needs a four-cycle warm-up period before it bursts the rest of the packet and the Output Module requires a five-cycle warm-up period. These warm-up periods are accommodated with additional FIFOs, which adds to the latency but does not affect the throughput.

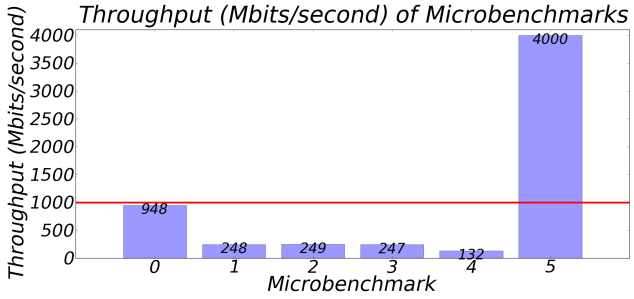


Figure 17: Throughput of the four microbenchmarks with two additional datapoints.

7.4 Application Case-study

Our application case study is a database query accelerator. Several works, such as [17] have shown FPGAs are a good target for such applications as they can perform low-latency, high-throughput applications. Furthermore, frameworks such as Apache Drill have shown that distributed clusters are a good way to accelerate database services [18]. The combination of those observations suggest that a distributed FPGA cluster is ideal for a database query accelerator.

The application we have built is a naive implementation of a query. The query is broken down into several sub-queries. Even though it is a naive implementation, the purpose of the infrastructure is to show that laying out the circuit is easy, and so is replication of that circuit (changing one number in the logical cluster file).

7.5 Query Implementation Details

The query is composed of five streaming components connected as a chain:

1. SQL Read: This component is responsible for reading SQL columns and outputting the data in a format that enables the rest of the components to process the data.

2. SQL Where: This operation is used to match column predicates and values with respect to a boolean operation (equal, greater than, less than, etc.)

3. SQL Like: This operation is used on a string column data and is used to match a string using a substring.

4. SQL Group: This operation aggregates different records using a grouping operation, such as counting.

5. SQL Write: This component is responsible for separating the stream coming out of SQL Group into columns.

Our infrastructure allows us to easily replicate the number of query processing engines, even across multiple FPGAs. When considering the number of processing engines, we first observe the resource usage of one replication of this processing engine, which is as follows: LUTs 11567 (2.7 %), Flip Flops 17176 (1.9 %), Block RAM 504 (34.3 %).

The Block RAM utilization limits our replication so we are limited to two query processing engines per FPGA (each query processing engine requires 35 % of the available BRAM). In our logical FPGA cluster file we would specify this as six replications and in our FPGA mapping we would divide the kernel nodes onto three FPGAs. We do the replication with a scheduler. The scheduler is located on one FPGA and forwards the data to either the replicated engines on the same FPGA or to another FPGA. This would send all the data to one destination and then the scheduler would be responsible for forwarding the data to the appropriate query processing engine. The first FPGA has a scheduler connected to two replicated query processing engines. The second and third FPGAs also have two replicated query processing engines connected directly to the Input Module as opposed to a Scheduler. The Scheduler on the first FPGA is responsible for scheduling work to all six replicated query processing engines across three FPGAs. This makes it simpler for the user since they do not have to change their interface to the cluster as they change the number of replications.

7.6 Case Study Evaluation

Our evaluation compares the throughput of one replication versus six replications across three FPGAs. As expected Figure 18 shows that the throughput increases as the replications increase and we expect it to continue to increase until it reaches the maximum of the FPGA chains observed earlier at about 240 MB/s. This example also highlights the scalability of our system. This would be at about 12 replications, which would require six FPGAs. The throughput limit of 240 MB/s is due to the speed of the CPU inputting table data into the FPGA chain. With a faster CPU we could theoretically saturate the network cable throughput limit of 1 GB/s, which can be increased with a faster network.

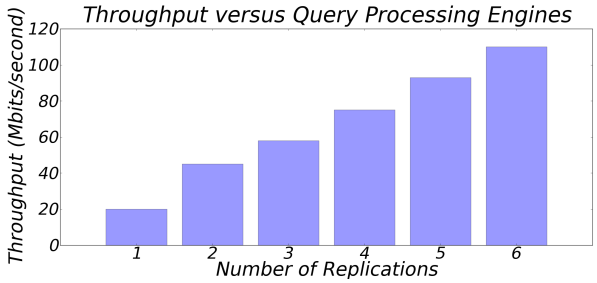


Figure 18: The throughput as we scale up the number of query processing engines.

8. CONCLUSION AND FUTURE WORK

The use of FPGA clusters can be useful as projects like the Microsoft Catapult project have shown. Our infrastructure provides a lightweight cluster provisioning tool. This tool, with a logical cluster description and FPGA mapping, can generate scalable clusters from a heterogeneous cloud. Moreover, these clusters are connected to the network as network devices ready to interact with other network devices. Our infrastructure makes it easy to scale up as with a simple directive we saw throughput scale almost linearly from one to six replicated processing units in our database case study.

The performance limitations in our experiments are due to the limits of our networking infrastructure. One area of future work is to address the slow 1 Gb Ethernet links by upgrading the Ethernet core. Our cluster example remains small but we also wish to upgrade our physical infrastructure to many more nodes so that we can demonstrate a large scale application. Our current case study application is limited due to the number of FPGAs available.

Another area of future work is to build true virtualization on top of this infrastructure. This can involve automatic placement of these kernels so the user will no longer provide an FPGA mapping, or the concept of making a large virtual FPGA out of an FPGA cluster. In both scenarios the cluster details are hidden from the user. Our infrastruture provides simple provisioning of FPGA clusters and can be the platform for using FPGA virtualization.

9. ACKNOWLEDGEMENTS

The authors would like to thank the SAVI testbed for providing the infrastructure, as well as NSERC, SAVI, Xilinx, and CMC/emSYSCAN for providing the equipment and funding for this project.

10. REFERENCES

[1] Andrew Putnum et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[2] IBM Research. OpenPOWER Cloud: Accelerating Cloud Computing. `https://www.research.ibm.com/labs/china/supervessel.html`, 2016.

[3] Adrian Caulfield et al. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, October 2016.

[4] Stuart Byma et al. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2014.

[5] Fei Chen et al. Enabling FPGAS in the Cloud. In *Computing Frontiers*, 2014.

[6] KVM. Kernel Virtual Machine. `http://www.linux-kvm.org`, 2015.

[7] Maxeler Technologies. MPC-X Series. `https://www.maxeler.com/products/mpc-xseries`, 2015.

[8] Omar Sefraoui et al. OpenStack: toward an open-source solution for cloud computing. In *International Journal of Computer Applications*, 2012.

[9] Apache Software Foundation. Apache Mesos. `https://mesos.apache.org`, 2015.

[10] NVidia Inc. NVidia Cuda Zone, Cluster Management Library. `https://developer.nvidia.com/cluster-management`, 2015.

[11] Joon-Myung Kang et al. SAVI Testbed: Control and Management of Converged Virtual ICT Resources. In *IFIP/IEEE International Symposium on Integrated Network Management*, pages 664–667. IEEE, 2013.

[12] Xilinx Inc. SDAccel Development Environment. `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2016.

[13] The Khronos Group. OpenCL Standard. `https://www.khronos.org/opencl/`, 2015.

[14] Xilinx Inc. Accelerating Integration. `http://www.xilinx.com/products/design-tools/vivado/integration.html`, 2016.

[15] Altera Corporation. Qsys - Altera's System Integration Tool. `https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-qsys.html`, 2016.

[16] Iperf. Iperf – The TCP/UDP Bandwidth Measurement Tool. `https://iperf.fr`, 2014.

[17] Christopher Dennl et al. Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration. In *Field Programmable Custom Computing Machines (FCCM)*, pages 25–28, 2013.

[18] Apache Software Foundation. Apache Drill. `https://drill.apache.org/`, 2015.