OpenPipes: making distributed hardware systems easier

Glen Gibb and Nick McKeown
Dept. of Electrical Engineering
Stanford University
Stanford CA 94305
grg@stanford.edu, nickm@stanford.edu

***Abstract*—Distributing a hardware design across multiple physical devices is difficult—splitting a design across two chips requires considerable effort to partition the design and to build the communication mechanism between the chips. Designers and researchers would benefit enormously if this were easier as it would, for example, allow multiple FPGAS to be used when building prototypes. To this end we propose OpenPipes, a platform to allow hardware designs to be distributed across physical resources. OpenPipes follows the model of many system-building platforms: systems are built by composing modules together. What makes it unique is that it uses an OpenFlow network as the interconnect between modules, providing OpenPipes with complete control over all traffic flows within the interconnect. Any device that can attach to the network can host modules, allowing software modules to be used alongside hardware modules. The control provided by OpenFlow allows running systems to be modified dynamically, and as we show in the paper, OpenPipes provides a mechanism for migrating from software to hardware modules that simplifies testing.**

I. Introduction

Many applications require dedicated custom hardware, often to meet performance requirements such as speed or power. For example, a typical 48-port 1 Gb/s switch/router is capable of forwarding at more than 100 million packets per second [4], which requires that the device be capable of completing a routing table lookup every 10ns.

Unfortunately hardware design is a difficult and time-consuming process. Many devices today contain in excess of one billion transistors [7], [19], making design a major undertaking. Designers must ensure that not only does their design work, but that their design meets all constraints imposed upon it. Power, speed, and size are three commonly encountered constraints, and trade offs between these dimensions must be made continually throughout the design process. For example, speed may be improved at the cost of increased size by replicating circuitry or increasing the number of pipeline stages.

Designers require the ability to prototype to evaluate trade offs and to verify their ideas. Field-programmable gate arrays (FPGAs) provide a useful mechanism for prototyping, and in many situations deploying, custom hardware solutions. FPGAs are reprogrammable, flexible, and low-cost. However they're not a perfect solution, in part because they have rigid constraints on size and the resources (e.g. memories) they contain. We frequently encounter the problem within our lab of designs being too complex to fit in a single FPGA; when this occurs we're faced with the problem of either simplifying our designs or utilizing multiple FPGAs simultaneously. (Even a simple IPv4 router with a 32-entry CAM-based routing table occupies 86% of an FPGA, leaving very little room to experiment with new features.)

There's currently no easy way to take advantage of multiple FPGAs simultaneously. Most prototyping boards host only a single FPGA; those that host more tend to be prohibitively expensive and usually require considerable effort to partition the design. A designer might justifiably ask "Why can't I utilize all the FPGAs I have at my disposal?" Designers should be able to experiment easily with different system partitioning, enabling design-space exploration for system-on-chip and multi-chip designs. Furthermore, the tools should allow experimentation with more advanced ideas, such as dynamically scaling compute resources—something that could be achieved by adding FPGAs to a running system. As a thought exercise, we ask the reader "Why can't we build an IP router that's distributed across multiple chips, in which we can grow the size of the routing table by adding additional routing table lookup elements?"

This paper introduces a platform called OpenPipes as a tool to help researchers and developers prototype hardware systems. As is common in system design, we assume the design is partitioned into modules; an important aspect of OpenPipes is that it is agnostic to whether modules are implemented in hardware or software. Modules can be implemented in any way, so long as they use the same OpenPipes module interface: a module could be implemented in Java or C as a user-level process, or in Verilog on an FPGA. The benefits of modularity for code re-use and rapid prototyping are well-known.

OpenPipes plumbs modules together over the network, and re-plumbs them as the system is repartitioned and modules are moved. Modules can be moved while the system is "live", allowing real-time experimentation with different designs and partitions. Modules can be implemented in software and tested in the system, before being committed to hardware. Hardware modules can be verified in a live system by providing the same input to hardware and software versions of the same module, and checking that they produce the same output.

OpenPipes places several demands on the network. First, it needs a network in which modules can move around easily and seamlessly under the control of the OpenPipes platform. If each module has its own network address (e.g. an IP address), then ideally the module can move without having to change its address. Second, OpenPipes needs the ability to bicast or multicast packets anywhere in the system—we may wish to send the same packet to multiple versions of the same module for testing or scaling, or to multiple different modules for performing separate parallel computation. Finally, we want control over the paths that packets take, so we can pick the lowest latency or highest bandwidth paths—eventually, we would like a system with guaranteed performance.

The two key building blocks of the platform are OpenFlow [12], [14] and NetFPGA [11]. We use OpenFlow as the network interconnect for OpenPipes. While other network technologies meet some of the requirements, OpenFlow allows the OpenPipes controller to decide the paths taken by packets between modules. It also allows modules to move seamlessly, without changing addresses, and provides a simple way to replicate packets anywhere in the topology. OpenFlow provides a way to guarantee bandwidth (and hopefully latency in the future) between modules, and hence for the system as a whole. NetFPGA is a programmable research platform that provides a network-attached FPGA.

At a high level, OpenPipes is just another way to create modular systems, and plumb them together using a standard module-to-module interface. The key difference is that OpenPipes uses *commodity networks* to interconnect the modules. This means we can easily plumb modules together across Ethernet, IP and other networks, without modifying the module.

We are not the first to propose connecting hardware modules together using the network. Numerous multi-FPGA systems have been proposed, early examples of which include [18], [8]. These examples use arrangements of crossbar switches to provide connectivity between multiple FPGAs. Networking ideas have been making their way into chip design for a while, with on-chip modules commonly connected together by switches, and communicating with proprietary packet formats [16], [1], [17], [15]. A slightly different approach is taken by [6]: daughter cards are connected in a mesh on a baseplate, and FPGAs on each card are hardwired to provide appropriate routing. While chip design can usefully borrow ideas from networking to create interconnected modules, it comes with difficulties.

It is not clear what network address(es) to use for a module: should they use Ethernet MAC addresses, IP addresses, or something else? The usual outcome is a combination of the two, plus a layer of encapsulation to create an overlay network between the modules. Encapsulation can provide a good way to pass through firewalls and NAT devices, but always creates headaches when the packet is made larger and needs to be fragmented. It also takes us down the path of increasing complexity in the network, as we add more layers and encapsulation formats. It seems to make the network more fragile and less agile.

A consequence of encapsulation is that it makes it harder for the modules to move around. If we want to re-allocate a module to another system (e.g. to another hardware platform, or move it to software while we debug and develop) then we have to change the addresses for each tunnel.

In a modular system that is split across the network—potentially at great distance—it is not clear how errors should

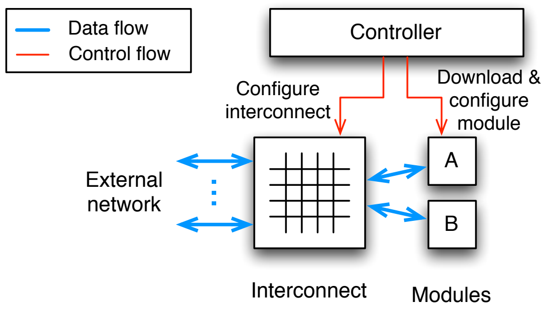

Fig. 1. Overview of OpenPipes showing the logical connection and data flow between components.

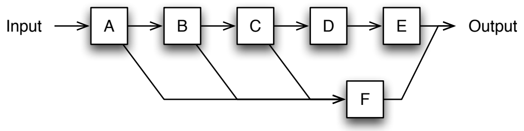

Fig. 2. Logical connection of modules within an example system. Modules A, B and C each have two downstream modules—packets leaving each of these modules may be sent to one or both of the downstream modules as determined by the application. The connection between modules is provided by the OpenFlow interconnect.

be handled. Some modules will require error control and retransmissions, whereas others might tolerate occasional packet drops (e.g. the pipeline in a modular IPv4 router). Introducing a retransmission protocol into the module interface is clearly a daunting task, as it would bloat the mechanism.

We specifically address these challenges within this paper.

II. What is OpenPipes?

In designing the OpenPipes platform, we have four main objectives. First, we want users to *rapidly and easily* build systems that operate at line rate. Second, we want to enable a design to be partitioned across different physical systems, and to consist of a mixture of hardware and software elements. Third, we want to test modules in-situ, allowing the behavior of two or more modules to be compared. Finally, we want the system to be dynamic, i.e., we would like to be able to modify the behavior of the system while it is running, without having to halt the system for reconfiguration.

A. OpenPipes Architecture

Our architecture has three major components: a series of *processing modules*, a flexible *interconnect*, and a *controller* that configures the interconnect and manages the location and configuration of the processing modules. To create a particular system, the controller instantiates the necessary processing modules and configures the interconnect to link the modules in the correct sequence. Figure 1 illustrates the basic components and the logical connection between them; Figure 2 illustrates an example interconnection of modules that create a system.

Interconnect: OpenFlow is used to interconnect modules under the supervision of an external controller. OpenFlow is a feature added to switches, routers and access points that provides a standard API for controlling their internal flow tables. So far, OpenFlow has been added to a number of commercial switches and routers. The OpenFlow protocol allows an external controller to add/delete flow entries to/from the flow table, and so decide the path taken by packets through the network.

In OpenPipes each module has one physical input and one physical output, although a single physical output may provide multiple *logical* outputs. Each connection from a logical module output to an input is an OpenFlow "flow". Packets from an output are routed according to the flow-table in each switch along the path. A module indicates the logical output by setting header fields appropriately. OpenPipes places no restriction on which header fields are used by a module; the only requirements are that the module and the controller agree on which fields specify the logical output, and that the switch can process the fields to route packets correctly.

For example, in Figure 2, module A provides two logical outputs: one connected to module B and the other to module F. The logical output is identified by the header fields of the packet; in this case one output may be identified by "header=X" and the other by "header=Y". Relevant header fields can be any combination of fields that OpenFlow can match, such as the Ethernet address, IP addresses, and TCP ports. The connection from each logical output is controlled by the controller; the controller can choose to route data indicated by "header=X" to module B, to module F, or to module B and F simultaneously.

Processing modules: Processing modules perform work on packets. The amount of work performed by a each module is chosen by the module's designer. We expect that, in general, modules will only perform a single, well-defined function since it is well known that this tends to maximize reuse in other systems.

Processing modules require zero knowledge of upstream or downstream neighbors—they process packets without regard to the source or destination of those packets, allowing the controller to make all decisions about how data should flow within the system. The only requirement placed upon modules by OpenPipes is that all modules use an agreed-upon protocol. This ensures that modules can correctly communicate with one another, regardless of their ordering within the system, and it enables the controller to extract logical port information from each packet.

Data flow between modules: Communication between modules is performed via packet transmission over the OpenFlow interconnect: modules *source* packets which the OpenFlow interconnect routes to one or more *sink* modules. Packets consist of data to be processed along with any metadata a module wishes to communicate. Data flow is *always* uni-directional. Bi-directional information exchange between modules is achieved by establishing two uni-directional flows in opposite directions.

A module may communicate information about a packet that it sources by sending *metadata* with the packet. This metadata can be used to communicate information to a downstream module or to inform the routing decision. The method we use borrows from the internal processing pipeline of the NetFPGA [11]: with each packet we transmit metadata about that packet. Doing so eliminates the need for a separate metadata channel and the need to match metadata with the associated packet.

Addressing and path determination: Source modules do *not* address packets to destination modules; instead, modules indicate a *logical* output port for each packet. The OpenPipes controller uses the logical port together with the desired system topology to route each packet. The logical port is indicated by setting a field in each header field.

For example, a checksum validation module may use two logical outputs to indicate the validity of the checksum within a packet. One logical output is used for packets with valid checksums, the other for packets with invalid checksums. The OpenPipes controller routes each logical output to different modules: valid packets are routed to modules for further processing according the needs of the system, while invalid packets may be routed to an error reporting mechanism.

The controller is the only component in the system that knows the interconnection between modules. Individual modules are unaware of the modules they are communicating with. It is the responsibility of the controller to map the header fields on packets to logical ports, and then use this information to route to the appropriate downstream module.

Module hosts: Modules can't exist by themselves: they must physically reside on some *host*. A host can be any device that can connect to the interconnect. Hosts are commonly programmable devices, such as an FPGA or a commodity PC, to which different modules can be downloaded. Hosts can also be non-programmable devices that hosts a fixed module.

Controller: The controller's role is three-fold: it interacts with the user, configures the interconnect, and manages the modules.

Users interact with the controller to configure the location, connection between, and configuration of each module. The configuration of the interconnect must be updated whenever the location of or connection between modules changes. Users also interact with the controller when they view the current state of the system and individual modules.

The controller interacts with modules when the user configures or views the status of a module, and when the user moves a module within the system. Moving a module within the system is implemented by creating an instance of the module in the new location, copying any relevant state from the old location, and finally rerouting flows to the new location, hence the need for interaction between controller and module when the user moves a module.

B. Related architectures

Click [13] is a modular router that, like OpenPipes, allows networking systems to be built by connecting reusable components. OpenPipes architecture differs in a number of ways including providing support for mixed hardware/software systems using the network to interconnect modules.

Clack [21] is a graphical router. It provides a way of visualizing network devices by showing the flow of traffic through a modular router. Clack is designed as a classroom teaching tool rather than a prototyping tool for high-performance network systems and as such does not address most of the objectives identified at the beginning of the section.

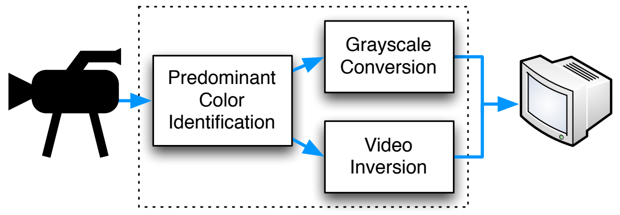

Fig. 3. Video processing application tranforms video as it flows through the system. The OpenPipes application is the region inside the dotted rectangle.

VROOM (Virtual ROuters On the Move) [20] is an architecture that allows virtual routers to move within the network, much like we allow modules to move within the network. VROOM however does not allow users to build networking systems as it is not a prototyping tool.

III. OpenPipes in action

A video processing application was built to demonstrate OpenPipes. The application itself is simple: it takes a video stream as input, performs a number of transforms on the video, and then outputs the transformed video. Despite the simplicity, the application allows us to demonstrate the main features of OpenPipes outlined earlier in the paper. This section discusses the application in detail.

A. *Video processing application*

The video processing application transforms a video stream as it flows through the system. The application provides two transforms: grayscale conversion and vertical inversion (mirroring about the central x-axis). Also provided is a facility to identify the predominant color within a video frame; information about the predominant color is used to selectively apply the transforms. A high-level overview of the application is shown in Figure 3. A screenshot of the system in action is shown in Figure 4.

Each of the transforms and the color identification facility are implemented as individual modules within the system. This enables the operator of the application to choose how video is processed by determining the connection between modules. It also allows each module to be instantiated on a separate module host, thereby distributing the application.

Input video is supplied by a video camera connected to a computer, and the output video is sent to a computer for display.

B. *Implementation*

The implementation of each of the main system components (controller, OpenFlow network, modules, and module hosts, as explained in § II-A) and a graphical user interface (GUI) for interacting with the system are discussued below.

Controller: The controller is written in Python and runs on the NOX [5] OpenFlow controller. The code is approximately 2,850 lines in length, of which only 1,800 lines are executable code. (The remainder is commenting and blank lines.) It is designed to be application-independent[1].

[1]Application independence refers to the ability to use the controller for applications other than the video processing example.

Much of the controller code is reponsible for processing commands from the GUI. Depending upon the command, the controller may be required to download a module to a module host, create or destroy flows (creation requires the calculation of a path from source to destination), or read and write state in an instantiated module. Non-GUI command processing code includes logic for learning the network topology from other NOX components and responding to events from module hosts (such as host up/down).

OpenFlow network: Multiple OpenFlow switches are used in a non-structured topology (see Figure 4). The topology was primarily dictated by the location of switches within our local network. We did however deploy one remote switch in Los Angeles; this switch is connected to our local network via a MAC-in-IP tunnel. (The controller sees the remote switch connected directly to the local switches.) The distributed non-structured topology clearly demonstrates OpenPipes' ability to geographically distribute hardware. The switches in use are a mixture of NEC IP8800 switches and NetFPGA-based OpenFlow switches.

Module hosts: A mixture of NetFPGAs and commodity PCs are used as module hosts. Hardware modules are hosted by the NetFPGAs and software modules are hosted by the commodity PCs. All module hosts are located locally except one NetFPGA deployed in Houston and connected to the OpenFlow switch in Los Angeles via a dedicated optical circuit.

Each host (NetFPGA and commodity PC) runs a small client utility that communicates with the controller. The utility notifies the controller of it's presence, provides the ability to download modules to the host, and processes reads and writes of module state by the controller. The utility is written in Python and consists of approximately 150 lines of shared code, and an additional 100–300 lines of platform-specific code.

Modules: Six versions of modules are implemented: frame inversion (hardware), grayscale conversion (software, hardware, and hardware with a deliberate error), predominant color identification (harware), and comparison (software). The comparison module is used in verification as outline in § III-C.

Each hardware module is implemented as a separate bitfile. As a result, only one hardware module can be active at any instant in a NetFPGA module host.

GUI: The graphical user interface (GUI), which is built on ENVI [2], provides a simple mechanism to enable user interaction with the controller. The GUI provides a view of the system showing the active network topology and the connections between module hosts and the network. Users can instantiate modules on the module hosts by dragging and dropping modules onto the individual hosts. Connections between modules, the input, and the output, are added and deleted by clicking on start- and end-points. Individual modules may also be selected to view the active state of the module. Figure 4 shows an image of the GUI.

C. *Testing*

OpenPipes provides a mechanism for verifying the correctness of a module: a known good instance is run in parallel with a version under test and the output of the two versions

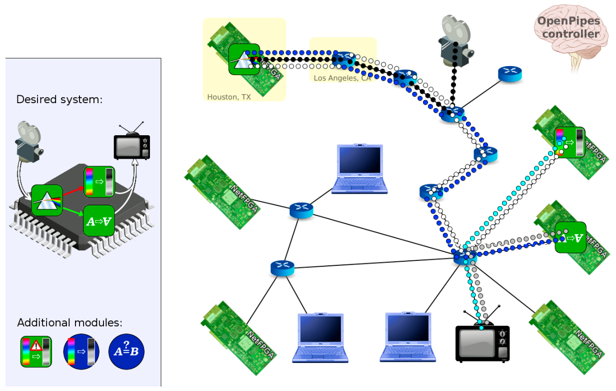

Fig. 4. Video processing application GUI.

is compared. The demonstration application provides three versions of the grayscale conversion module and a comparison module to demonstrate this. The software version of the grayscale module is considered to be the “correct” module and the hardware versions are considered as versions under test. (In a real-life scenario, a software version may be developed first to prove an algorithm and then converted to hardware to increase throughput.) The input video stream is bicast to the software module and the version under test, and the output of both modules is fed into the comparison module. The comparison module reports differences between frames when comparing the output of the fautly hardware module with the reference.

One point to note is that the comparison module in use is written specifically for this application. It performs frame-by-frame comparison by using explicit knowledge of the packet format to extract and compare the same frame from both streams. Application-specific comparison modules use knowledge about the two streams to synchronize the streams; a generic comparison module would require the use of some explicit syncronization mechanism.

D. Limitations

The application was written expressly for demonstration purposes. Two important design decisions were made during implementation to simplify the process that should be highlighted.

First, only *one* module may be instantiated per host. OpenPipes *does* support the instantiation of multiple modules in a single host. The transforms used in the application are simple enough that multiple transforms could be instantiated inside a single FPGA. § IV-C outlines how to enable multiple modules in a single FPGA.

Second, the modules are implemented without any flow or rate control. Flow/rate control is not needed for this application as the modules are able to process data at a much higher rate than the video source rate. (The video is a 25 Mb/s stream and the transforms are simple enough that even the software-version of the grayscale module is able to process this without dropping frames.) § IV-A outlines how to support flow and rate control.

E. Demonstration

A video of the video processing application in action may be viewed at: http://openflow.org/wk/index.php/OpenPipes

IV. Plumbing the depths

Applications vary considerably in terms of their requirements from the OpenPipes system. Examples of requirements include flow control and error control. This section addresses issues to be considered when constructing and deploying OpenPipes applications.

A. Flow control

Flow control is required by many applications. It is needed when an application cares about data loss due to congestion (overflowing of buffers) caused when a source module generates data at a rate faster than a destination can process it.

Flow control was not required for the video processing example as all modules process data at a rate much greater than the source video rate. (The source video rate was 25 Mb/s.)

Rate limiting and credit-based flow control are two schemes that are appropriate for applications requiring flow control. Applications may use one scheme or the other exclusively, or may choose to use a mixture of the two.

Rate limiting: Rate limiting is a simple mechanism that is applicable for certain applications. The output rate of each module is limited to a set maximum rate; data loss is prevented by ensuring the maximum output rate of each module is less than the rate at which downstream modules can process data.

The simplicity of rate limiting is due to it’s “open-loop” nature. There is no feedback from downstream modules to upstream modules to report the current congestion state.

Rate limiting is most appropriate when each module is able to process data at a nearly-constant rate. (Or at least for the given output rate it is able to process data at a nearly-constant rate.) It performs poorly in situations where the processing rate varies considerably based upon the input, as the maximum rate needs to be set based upon worst-case performance.

Responsibility for assigning the maximum output rates lies with the controller. Each module reports its maximum input rate to the controller, allowing the controller to assign rate limits based upon the active configuration. In addition to ensuring that the maximum input rate is never exceeded for any module, care must be taken to ensure that no links in the network are oversubscribed.

Care must also be taken to “propagate” information about maximum input rates of modules backward through the system. For example, if modules A, B, and C are connected in a chain, then the output rates of modules A *and* B must be limited to ensure that the maximum input rate of module C is never exceeded. This concept is illustrated in Figure 5. Note that modules should report their relation between input and output rate to allow the correct calculation of propagated rate limits.

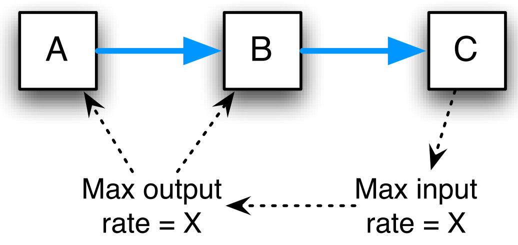

Fig. 5. Rate limit information must "propagate" backward through the system. Module C has a maximum input rate of X, therefore the maximum output rates of modules A and B should be limited to X. Buffer overflow would occur in B if A's rate was not limited to X. (Note: this assumes that the output rate of B is identical to it's input rate.)

Credit-based flow control: Credit-based flow control [9] is a mechanism in which a downstream element issue credits to an upstream element. An upstream element may transmit as much data as it has credits for; once it exhausts it's set of credits it must pause until the downstream element grants more credits.

Credit-based flow control is more complex than rate limiting as the system runs "closed-loop": downstream modules must communicate with upstream modules. The mechanism is appropriate in scenarios where there is considerable variation in a module's processing rate depending upon input data. Simple rate limiting is non-ideal in such situations as the system would need to operate under worst-case assumptions for processing rates for each module.

Credit-based flow control requires a sizable buffer at the input of each module to ensure that buffer never empties due to the upstream module waiting to receive credits. If a module is capable of processing data at a rate BW, with a round-trip-time between modules of RTT, then the required buffer size is $BUF = BW \times RTT$. This equates to 125 KB if $BW =$ 1Gb/s and $RTT =$ 1ms. The buffer size requirement increases if the downstream module doesn't immediately issue credits to the upstream module.

Challenge: connections other than one-to-one: Connections other than one-to-one complicate both flow control mechanisms.

One-to-many scenarios, in which one upstream module is directly connected to many downstream modules, requires the processing limits of *all* downstream modules be considered. This is not a significant challenge when using rate limiting: the upstream module is limited to the input rate of the *slowest* downstream module. Doing so ensures that the upstream module never transmits faster than any of the downstream modules can process.

One-to-many scenarios complicate credit-based flow control as many downstream modules send credits to the upstream module. The upstream module must track the credits issued by *each* downstream module, restricting it's output based upon the downstream module with the fewest issued credits. This increases the amount of state and the complexity of the enable/pause logic within each module.

Many-to-one situations, in which many upstream modules are connected to one downstream module, cause challenges for both flow control schemes. The combined output of all upstream modules must not overload the downstream module. The current solution is to split the rate or credits equally between all upstream modules. This prevents overloading but also results in underutilization when the quantity of data sourced by different upstream modules differs significantly. The reason is that bandwidth unutilized by one module can not be given to another module.

B. Error control

Error control within OpenPipes is not required by all applications: applications may provide end-to-end error control or be tolerant of errors. The video processing application is tolerant of errors as dropped or corrupted data packets simply appear as small visual glitches in the output video. Two types of error control can be used depending upon needs: error detection and error correction/recovery.

Error detection is handled via mechanisms like checksums. It may be part of a recovery mechanism, as discussed below, or it may be used simply to prevent erroneous data from progressing through the pipeline. The application must decide how to respond when errors are detected (assuming it's not being used as part of an error recovery system); this will not be discussed further as it is application specific.

Error correction is provided by mechanisms that introduce redundancy in the data, such as error-correcting codes [10]. Provided that errors are relatively minor, the correction mechanism can repair data as it is received by each module.

Error recovery is handled via a combination of error detection and retransmission [3]. Retransmission requires buffering of data at the point of transmission, waiting for a confirmation of reception (or lack of notification of an error) before the data is discarded.

Error control and flow control mechanisms may be used simultaneously.

C. Multiple modules per host

The example video application presented in § III places only one module within a host at any given time. This is *not* a limitation of OpenPipes, it was done to simplify the example.

Multiple modules are supported in a single host via two mechanisms: an OpenFlow switch is instantiated in the host alongside the modules, or each module is associated with a separate physical interface.

Instantiating an OpenFlow switch allows the use of a single connection between the host and the OpenFlow network. The internally instantiated switch sits between external switches and the modules, extending the network into the host. The internal switch does not need to support the entire set of OpenFlow matches and actions, only those that are used by OpenPipes.

Use of multiple physical interfaces eliminates the need to dedicate resources on the host to implementing an additional switch. In this scenario, the number of physical interfaces limits the number of modules that can be instantiated on the host (in addition to the limits imposed by the computation resources).

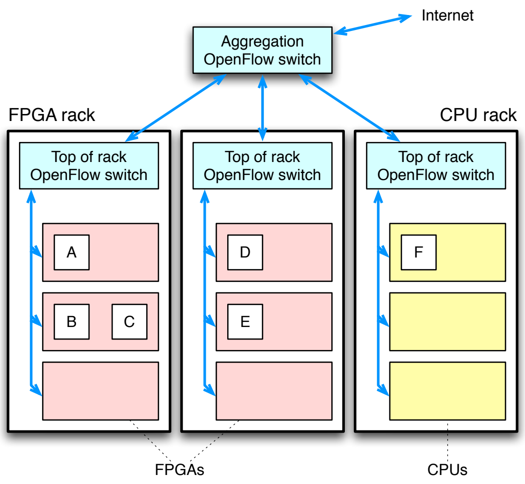
Internet
Aggregation OpenFlow switch
FPGA rack
CPU rack
Top of rack OpenFlow switch
Top of rack OpenFlow switch
Top of rack OpenFlow switch
A
B
C
D
E
F
FPGAs
CPUs

Fig. 6. Example data center style deployment. FPGAs or CPUs within a rack are connected to a top-of-rack switch. Racks are connected to an aggregation switch which may provide external connectivity.

D. Routing

The current implementation uses shortest-path routing between sources and destinations. Applications are not restricted to using shortest-path routing; OpenFlow allows them to perform routing based upon their individual needs. An application may choose high-bandwidth paths over low latency paths for example.

Applications should take into account the utilization of links when choosing routes. Care should be taken to prevent oversubscription of individual links.

E. Additional challenges

Some applications may require the synchronization of modules within the system, other applications may have strict latency constraints. We have deliberately ignored these issues in this paper to focus on the challenges that we believe are applicable to a wider set of applications.

F. Enabling large deployments

We believe that the architecture enables large deployments as illustrated in Figure 6. It should be possible to distribute the OpenFlow interconnect over a number of racks and populate the racks with a large number of FPGA and CPU-based systems. Users of such a setup would be able to use OpenPipes to build a highly-scalable networking system.

V. Conclusion

OpenPipes is a new platform for building distributed hardware systems. The system to be built is partitioned into modules, all of which may be physically distributed; what makes OpenPipes unique is that it uses an OpenFlow network as the interconnect between modules. Use of an OpenFlow network affords a number of benefits: (i) any network-attached device can host modules, allowing the use of varied hardware devices and the inclusion of software modules within the system, (ii) running systems may be dynamically modified by instantiating new modules and updating routes within the interconnect, and (iii) development and verification is simplified via a migration path from software to hardware modules in which software modules may be used as references against which to test hardware modules.

Distributing hardware does pose a number of challenges. This paper addresses what we see as the major challenges, such as flow control and error control. While we don't yet have perfect solutions for all challenges, we do lay down a useful foundation that allows many applications to be built using OpenPipes. OpenPipes allows designers to easily partition their systems across multiple physical devices, allowing them to better make use of the resources at their disposal.

References

[1] W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM.

[2] ENVI: An Extensible Network Visualization & Control Framework. http://www.openflowswitch.org/wp/gui/.

[3] G. Fairhurst and L. Wood. Advice to link designers on link Automatic Repeat reQuest (ARQ). RFC 3366 (Best Current Practice), Aug. 2002. http://www.ietf.org/rfc/rfc3366.txt.

[4] Force10 Networks: S-Series. http://www.force10networks.com/products/sseries.asp.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards and operating system for networks. In *ACM SIGCOMM Computer Communication Review*, July 2008.

[6] S. Hauck, G. Borriello, and C. Ebeling. Springbok: A Rapid-Prototyping System for Board-Level Designs. In *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[7] Intel to deliver first computer chip with two billion transistors, Feb. 2008. http://afp.google.com/article/ALeqM5ipelkeZwHqz3cqmha_jD7gNhB98A.

[8] M. Khalid and J. Rose. A novel and efficient routing architecture for multi-FPGA systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(1):30–39, Feb 2000.

[9] N. Kung and R. Morris. Credit-based flow control for atm networks. *Network, IEEE*, 9(2):40 –48, mar/apr 1995.

[10] S. Lin and J. D. J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 2004.

[11] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA—an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[13] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 217–231, New York, NY, USA, 1999. ACM.

[14] The OpenFlow Switch Consortium. http://www.openflowswitch.org.

[15] J. D. Owens, W. J. Dally, R. Ho, D. N. Jayasimha, S. W. Keckler, and L.-S. Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[16] C. L. Seitz. Let's route packets instead of wires. In *AUSCRYPT '90: Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 133–138, Cambridge, MA, USA, 1990. MIT Press.

[17] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Design Automation Conference, DAC '01*, June 2001.

[18] M. Slimane-Kadi, D.Brasen, and G.Saucier. A fast-FPGA prototyping system that uses inexpensive high-performance FPIC. In *Proceedings of the ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[19] UMC delivers leading-edge 65nm FPGAs to Xilinx, Nov. 2006. http://www.umc.com/english/news/2006/20061108.asp.

[20] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. *SIGCOMM Comput. Commun. Rev.*, 38(4):231–242, 2008.

[21] D. Wendlandt, M. Casado, P. Tarjan, and N. McKeown. The clack graphical router: visualizing network software. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 7–15, New York, NY, USA, 2006. ACM.