OpenPipes: making distributed hardware systems easier

Glen Gibb [1], Nick McKeown [2]

Dept. of Electrical Engineering, Stanford University
Stanford CA 94305
USA
[1] grg@stanford.edu
[2] 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

[1] Many applications require dedicated custom hardware, often to meet performance requirements such as speed or power. Unfortunately hardware design is a difficult and time-consuming process. Many devices today contain in excess of one billion transistors [12], 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.

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 dilemma of either simplifying our designs or utilizing multiple FPGAs simultaneously.

[1] A more detailed version of this paper appears as a technical report (TR10-MKG-170910): http://yuba.stanford.edu/techreports/TR10-MKG-170910

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 a 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, then ideally the module can move without changing 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 key building block of the platform is OpenFlow [7, 8], which is used as the network interconnect. While other network technologies meet some of the requirements, OpenFlow allows the OpenPipes controller to decide the paths taken by packets between modules, allows modules to move seamlessly without changing addresses, and provides a simple way to replicate packets anywhere in the topology. It also provides a way to guarantee bandwidth (and hopefully latency in the future) between modules, and hence for the system as a whole.

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 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 [5, 11]. 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 [1, 9, 10]. A slightly different approach is taken by [4]: 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 for interconnecting 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 MAC and IP, plus a layer of encapsulation to create an overlay network between the modules. Encapsulation solves some problems but results in increased complexity and tends 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.

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.

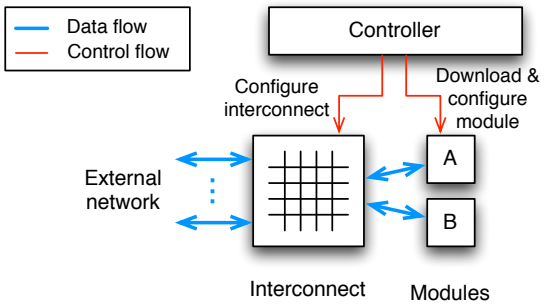

Fig. 1. Overview of OpenPipes showing the logical connection and data flow between components.

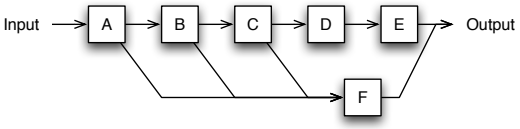

Fig. 2. Logical connection of modules within an example system. Modules A, B and C each have two downstream modules—packets leaving each of these modules may be sent to one or both of the downstream modules as determined by the application. The connection between modules is provided by the OpenFlow interconnect.

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 network, 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. 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. One logical output may be indicated by "header=X", the other by "header=Y".

Processing modules: Processing modules perform work on packets. 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.

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.

As mentioned above, modules should use an agreed-upon format to ensure that modules can correctly communicate with one another.

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.

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 host 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.

The controller interacts with modules when the user configures 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 and then copying any relevant state from the old location, hence the need for interaction between controller and module.

III. OpenPipes in action

A video processing application was built to demonstrate OpenPipes. The application itself is simple, but despite the simplicity, the application allows us to demonstrate the main features of OpenPipes. This section discusses the application in detail.

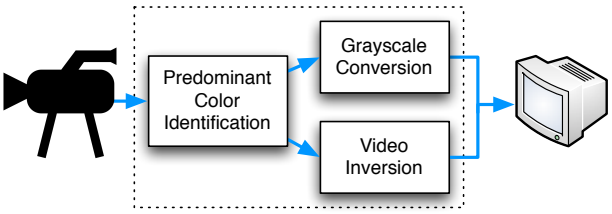

Fig. 3. Video processing application tranforms video as it flows through the system. The OpenPipes application is the region inside the dotted rectangle.

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; this information 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

Controller: The controller is written in Python and runs on the NOX [3] OpenFlow controller. It is designed to be application independent[2]. Much of the code is responsible for processing commands from the GUI; other elements of the code perform functions like learning the topology and responding to events from module hosts.

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 the local network via a tunnel. 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 [6] 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

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

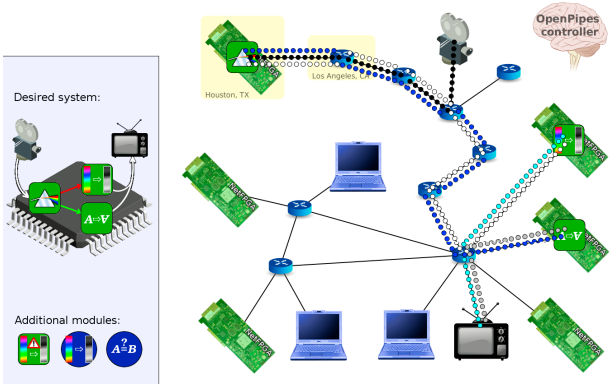

Fig. 4. Video processing application GUI.

OpenFlow switch in Los Angeles via a dedicated optical circuit.

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.

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. 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 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.

D. Demonstration

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

IV. 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.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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