Enabling Hardware Exploration in Software-Defined Networking: A Flexible, Portable OpenFlow Switch

Asif Khan
MIT - CSAIL
Cambridge, MA, USA
aik@csail.mit.edu

Nirav Dave
SRI International
Menlo Park, CA, USA
ndave@csl.sri.com

***Abstract*—The OpenFlow framework allows the data plane of a network switch to be managed by a software-based controller. This enables a software-defined networking model in which sophisticated network management policies can be deployed. In this paper, we present an FPGA-based switch which is fully-compliant with OpenFlow 1.0, and meets the 10 Gbps line rate. The switch design is both modular and highly parametrized. It has generic split-transaction interfaces and isolated platform-specific features, making it both flexible for architectural exploration and portable across FPGA platforms. The flow tables in the switch can be implemented on Block RAM or DRAM without any modifications to the rest of the design. The switch has been ported to the NetFPGA-10G, the ML605 and the DE4 boards. It can be integrated with a Desktop PC via either the PCIe or the serial link, and with an FPGA-based MIPS64 softcore as a coprocessor. The latter FPGA-based switch-processor system provides an ideal platform for network research in which both the data plane and the control plane can be explored.**

I. Introduction

As network infrastructure becomes more integral in modern applications, the performance and correctness requirements on network packet flows become more complicated. Consider, for instance, a transcoding service in the cloud. In this application, inflows to the transcoding engine from the original video source should have minimal latency in order to get the data on time, while outflows need high bandwidth to serve all clients simultaneously. Achieving these goals effectively requires that we step away from the original self-contained, fully-autonomous network switching model to an application-aware switching model.

OpenFlow [1], an instance of software-defined networking (SDN) is a platform coupling the deep access within the network data plane, needed for SDN operations, with a simple API for network-device control. OpenFlow has gained popularity within both academia and industry [2]–[4] as a framework for both network research and implementation.

The OpenFlow community provides an open-source software package for Linux that implements a software OpenFlow switch as a reference implementation [5]. Though complete and easy to modify, software cannot meet modern line rates and as such is a non-starter for realistic exploration of the network architecture. Modern commodity switches have historically provided SDN-like functionalities at high line rate, and many efforts have been made to wrap or modify these routers to serve as OpenFlow switches. While fast, these commodity implementations do not provide the full introspection which inhibits network architecture exploration.

FPGAs provide an opportunity for flexibility with sufficient speed for realistic line rates. In fact, various OpenFlow switches [6]–[8] have been developed on the NetFPGA platforms [9], a set of FPGA boards designed to support OpenFlow and similar networking frameworks.

Naous *et al.* [6] demonstrated an OpenFlow switch implemented on the NetFPGA-1G platform. The switch has 4 ports and is capable of meeting a 1 Gbps line rate. It classifies packets into flows based on a 10-tuple which can be matched either exactly or by using field-level wildcards. It can maintain up to 64K exact match rules and up to 32 wildcard rules. Antichi *et al.* [7] improved on Naous *et al.*'s switch by defining flows in terms of patterns described as regular expressions. Their switch implementation can store up to 200K rules while still meeting line rate. Yabe [8] developed a 4-port OpenFlow switch on the NetFPGA-10G board capable of meeting a 10 Gbps line rate.

These FPGA-based switches are able to meet realistic line rates (1-10 Gbps), implement a significant portion of the OpenFlow specification and have been used for network research. The design of these switches, however, was not done with the goal of microarchitectural exploration or portability across FPGA platforms. They were designed in low-level Verilog RTL with strong timing assumptions resulting in very brittle microarchitectures. The flexibility to implement the flow tables on a different type of memory, for example, could trigger a complete re-design of the switch architecture. Portability across FPGA platforms is another highly desired feature because some FPGA boards may be more easily available or offer better performance. These switches, however, are strongly tied to the interface protocol and timing behavior of the external resources, such as DRAM, available on a particular FPGA board, severely limiting their portability.

In this paper, we detail the construction of a high-performance OpenFlow switch in Bluespec SystemVerilog (BSV) [10], a high-level HDL, and address the challenges of its flexibility and portability. Our design provides many features of interest for OpenFlow-based network research. Specifically, our switch fully meets the OpenFlow 1.0 specification [11] and achieves a line rate of 10 Gbps. Our design is

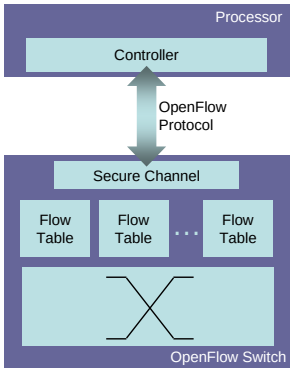

Fig. 1. Overview of the OpenFlow network architecture

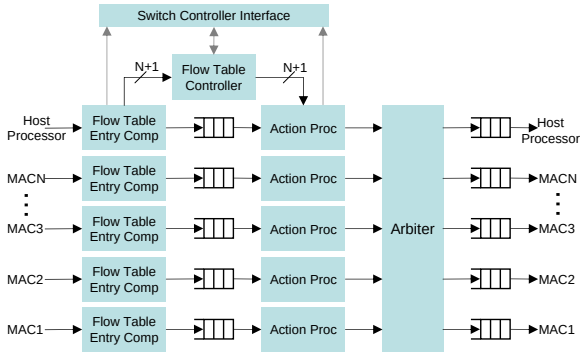

Fig. 2. OpenFlow switch architecture

highly modular and parametrized, and makes use of latency-insensitivity, split-transaction interfaces and isolated platform-specific features. These enable the relatively easy porting and retuning of the switch design across FPGA platforms. We have ported the switch to three FPGA boards: NetFPGA-10G and ML605 from Xilinx, and DE4 from Altera.

The ability to easily carry out modular refinement of the switch design can be exploited for improved switch functionality and performance. For instance, we implemented the flow tables on Block RAM or DRAM without making any modifications to the rest of the design. This enabled us to almost effortlessly explore the resource-performance tradeoff of the two flow table implementations on the three FPGA boards. We also implemented the switch in two configurations: as an FPGA accelerator communicating with a Desktop PC via the PCIe or the serial link, and as a coprocessor in an FPGA-based MIPS64 softcore [12]. This flexibility required us to implement adapters for the three switch-controller interfaces, but required no modifications to the switch design.

Paper organization: Section II describes the switch architecture and the OpenFlow network which includes the switch and a processor running the OpenFlow controller software. Section III presents an evaluation of our switch design and its FPGA implementation. Section IV summarizes our work and discusses some of the lessons learnt during its course.

II. Design of an FPGA-based OpenFlow switch and network

As shown in Figure 1, the OpenFlow network architecture includes switches, controller(s) and a secure channel based on the OpenFlow protocol which connects the switches with the controller(s). The OpenFlow switch directs packets based on flow tables, while the OpenFlow controller is responsible for modifying these flow tables to implement the high-level routing policy. This results in a separation between data and control plane functionality.

A. OpenFlow Switch Architecture

The high-level operation of an OpenFlow switch is quite straightforward. As packets stream in, the switch aggregates header information and compares it against the flow table entries. The switch then updates the packet according to the actions prescribed in the matching entry, and sends the updated packet to the appropriate egresses via a crossbar.

Our switch architecture is depicted in Figure 2. It is parametrized by N, the number of MAC ports in the switch, and W, the width of the internal data plane. These values can be freely changed to meet resource and performance requirements. In an N-port instance of the switch, there are $N+1$ ports with the $(N+1)^{\text{th}}$ port reserved for communication with a host processor. The switch also has a controller interface to allow the host processor access to the flow tables and the various registers that maintain statistics.

Since a major motivation for this design was to enable easy modular refinement and rapid microarchitectural exploration, we implemented the switch in Bluespec SystemVerilog (BSV) [10]. BSV is a high-level hardware description language built around guarded atomic actions or *rules*, and lightweight guarded modular interfaces. These representational abstractions allow the designer to encode both how modules are used, *e.g.,* when the design should enqueue a new token into a FIFO, and how their use is restricted, *e.g.,* the design cannot enqueue into a FIFO if it is full, directly and naturally.

As a switch is always permitted to drop packets due to oversubscription, it is common practice in RTL switch designs for the datapath to be a synchronous pipeline with no back pressure. This reduces some design complexity and eliminates some logic statically. However, this minor efficiency comes at the cost of less understandable compositional semantics for modules, constraining the modular refinement capability. For these reasons, we implemented all the module interfaces to provide back pressure via BSV's ready-enable micro-interface protocol to stall operations when sufficient buffering is not available. This change has a negligible area cost, but it dramatically reduces the design exploration effort.

Our switch design endorses the "fail-early" principle, dropping any packet for which it cannot guarantee end-to-end buffering. When the header flit of a packet arrives at the switch and sufficient buffering is not available, the header flit and all the subsequent flits belonging to the packet are discarded, and a failure is recorded.

The design of the switch pipeline has been divided into the following modules.

Flow Table Entry Composer: Each input port of the switch

receives packets as a sequence of fixed-size flits. For each input port, there is an associated flow table entry composer which aggregates the packet header and decodes it into an internal flow table entry tag representation. This entry is forwarded to the flow table controller as a query. The composer also forwards the entire packet to the corresponding action processor.

Flow Table Controller: The flow table controller is responsible for maintaining the flow tables entries and the per-flow statistics, and arbitrating the requests to access the flow tables.

The controller is implemented as a flexible pipeline capable of tolerating variations in the flow table latency. It can receive up to $N+2$ queries in each cycle: 1 from the switch controller interface, and $N+1$ from the flow table entry composers. Every cycle, the controller, via a configurable priority scheme, selects a request and pushes it into the pipeline. The controller handles requests serially due to resource constraints. Since only one request is made per packet, this is not a performance bottleneck, and has the additional benefit of providing intra flow table consistency.

The flow table controller maintains two tables, an exact match table implemented on either Block RAMs or DRAM, and a wildcard match table implemented as a CAM. Each flow table entry consists of three components: a compressed representation of packet header information which serves as a tag for matching against requests, a list of actions determining the output ports and any modifications that need to be made to the matching packet, and flow-specific statistics, *e.g.,* the number of packets in the flow, the number of bytes sent, and the time when the last matching packet was received. The data layout has a one-to-one correspondence with the C-struct in the OpenFlow controller software.

The flow table controller pipeline issues a request to both the exact match and the wildcard match tables in parallel, prioritizing the response from the exact match table. If a match is found, it forwards the action list obtained from the matching flow table entry to the action processor module of the corresponding port. If, however, a match is not found, it instructs the action processor to either drop the packet or send the packet to the OpenFlow controller.

Action Processor: The action processor buffers the unmodified packet until it has received the action list from the flow table controller. It updates the destination ports and the packet header, as required by the action list. It can modify the fields of the data link, the network and the transport layers. It also updates the checksum for the network and the transport layers.

Arbiter: The arbiter is an $(N+1) \times (N+1)$ crossbar. Every cycle, it selects 1 flow based on a configurable scheduling policy, and forwards the selected flow to the output queues of all the associated destination ports. This selection is maintained until the entire packet is transmitted. The arbiter can direct multiple flows simultaneously.

Switch Controller Interface: The switch controller interface module provides an address-mapped interface for the OpenFlow controller to the flow tables and the statistics. In addition to the necessary logic for marshaling the accesses over the controller-switch communication link, it has additional interlock logic to guarantee that flow table updates are applied atomically. This allows us to reason about the functionality of the switch at the granularity of packet transfers.

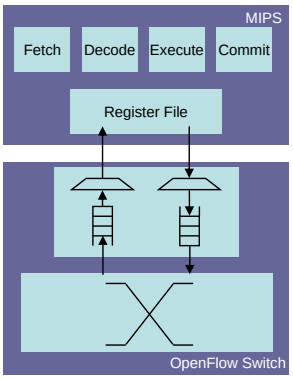

Fig. 3. OpenFlow network architecture with a switch integrated into an FPGA-based MIPS64 softcore as a coprocessor

B. OpenFlow Network Architecture

The use of generic interfaces facilitates the integration of the switch with any processor that runs the OpenFlow controller software. The processor interface can be wrapped to translate its protocol into the split-transaction protocol of the switch interface. Using this technique we have two instances of our switch differentiated by the interface to the host processor.

In the first instance, the OpenFlow controller software runs on a desktop PC which communicates with the switch via either the PCIe DMA engine or the serial link, resulting in highly variable latency. This corresponds to a standard implementation of the OpenFlow network architecture.

In the second instance, the OpenFlow controller software runs on an FPGA-based MIPS64 softcore, and the switch is attached to the softcore as a coprocessor (see Figure 3). This organization provides a low-latency link with a response time guarantee. The softcore communicates with the switch using the MIPS64 coprocessor move instructions (MFC and MTC). Data is communicated between the switch and the softcore in 64-bit chunks, and is translated into a dynamically-sized tagged union representation by the interface adapter.

III. Evaluation

Making use of the flexibility described in the previous section, we successfully ported our switch to three FPGA platforms: NetFPGA-10G and ML605 from Xilinx, and DE4 from Altera. Figure 4 provides a comparison of the utilized resources, the operational clock speed and the consumed power of the switch on the three boards. The three switch implementations only differ in the number of MAC ports. On the NetFPGA-10G board, which has 4 SFP+ transceivers, the switch operates at 160 MHz and has a 64-bit datapath, meeting the 10 Gbps per lane performance requirement.

Our switch design is very compact, and can be readily modified. It comprises of approximately 2400 lines of BSV code. In

	NetFPGA-10G	ML605	DE4
Ports	5×5	2×2	5×5
LUTs	24009	12062	11131
Flip Flops	29326	15469	40287
Block RAMs	159	85	1.1 Mb
Clock Speed	160 MHz	100 MHz	100 MHz
Power	876 mW	275 mW	442 mW

Fig. 4. Comparison of our OpenFlow switch implementations on different FPGA boards

comparison, Yabe's OpenFlow 10G switch implementation [8] is approximately 10K lines of Verilog RTL.

We implemented the exact match flow tables on both Block RAM and DRAM on the three FPGA boards. Figure 5 presents the resource-performance tradeoff between Block RAM and DRAM for implementing the flow tables on the three boards. The switch architecture, when the exact match flow tables are implemented on Block RAMs, has a pipeline latency of 19 cycles for a packet to travel from ingress to egress.

We implemented both "Desktop PC-switch" and "MIPS64 softcore-switch" network architectures on all three boards. In the former architecture, the switch responds to controller requests in $530K - 560K$ cycles, when the serial link is used, and in $6.6K - 7.3K$ cycles, when the PCIe is used. In the latter architecture, the switch responds with a fixed latency of 9 cycles.

Our switch was tested using a regression suite maintained by the OpenFlow community. The suite tests for OpenFlow functionality, and verified that the switch met the OpenFlow specification v1.0.0.

IV. Conclusion

Our aim was to provide a complete FPGA-based OpenFlow switch implementation which was both flexible for architectural exploration and portable across FPGA platforms so that new and improved switch functionalities can be easily incorporated to facilitate research in software-defined networking. We have successfully accomplished our goal, and hope that our switch is widely adopted by the OpenFlow community.

To conclude, we mention some of the important lessons learnt during the course of this work.

1) A modular design in which modules communicate through generic interfaces, and tolerate variations in latency of the modules that they communicate with, greatly facilitates robust microarchitectural exploration and portability across FPGA platforms.

FPGA Board	Block RAM		DRAM	
	Entries	Latency	Entries	Latency
NetFPGA-10G	14.7K	1 cycle	2.9M	25 cycles
ML605	18.9K	1 cycle	5.8M	20 cycles
DE4	20.8K	1 cycle	11.6M	18 cycles

Fig. 5. Resource-performance tradeoff between Block RAM and DRAM on different FPGA boards, when used entirely for implementing the exact match flow tables

2) Isolating platform-specific features, such as Block RAMs and DSP slices, not only facilitates portability, but also reduces the inconsistencies of FPGA synthesis tools. Inline instantiation of these features often results in tools silently eliminating the control logic around them, resulting in a broken design.

3) FPGA board vendors often do not provide open source versions of IP blocks required for external communication, such as memory controllers and PCIe drivers. The development of these IP blocks for each board requires a substantial amount of effort. A central venue for the user community of each board will promote sharing and ease the development burden.

Acknowledgements

The authors would like to thank Muhammad Shahbaz and Andrew Moore for the loan of the NetFPGA-10G board and their support during the debugging phase, and Robert Norton for his help in bringing up the MIPS processor. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the United States Air Force, under Contract No. FA8750-11-C-0249. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or the United States Air Force.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[3] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, October 2009.

[4] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, oct. 2010, pp. 408 –415.

[5] "OpenFlow Reference Linux Software – Soft Switch," `http://www.openflow.org/wp/tag/soft-switch`.

[6] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08, New York, NY, USA, 2008, pp. 1–9.

[7] G. Antichi, A. Di Pietro, S. Giordano, G. Procissi, and D. Ficara, "Design and development of an openflow compliant smart gigabit switch," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, dec. 2011, pp. 1 –5.

[8] "OpenFlow implementation on NetFPGA-10G – Design Document," `https://docs.google.com/document/d/1ZwHXQZocKwQls6Ted8VZO8h9MjBtu9WxV2fAY44eOgE/edit`.

[9] M. Blott, J. Ellithorpe, N. McKeown, K. Vissers, and H. Zeng, "FPGA Research Design Platform Fuels Network Advances," *Xilinx Xcell Journal*, September 2010.

[10] *Bluespec SystemVerilog Version 3.8 – Reference Guide*, Bluespec, Inc., Waltham, MA, November 2004.

[11] "OpenFlow Switch Specification Version 1.0.0," `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`, December 2009.

[12] R. Watson, P. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. Moore, S. Murdoch, P. Paeps, M. Roe, and H. Saidi, "CHERI: a research platform deconflating hardware virtualization and protection," in *Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, March 2012.