Reconfigurable Router Modules Using Network Protocol Wrappers

Florian Braun John Lockwood Marcel Waldvogel
{florian,lockwood,mwa}@arl.wustl.edu

Applied Research Laboratory[⋆]
Washington University in St. Louis

Abstract. The ongoing increases of line speed in the Internet backbone combined with the need for increased functionality of network devices presents a major challenge for designers of Internet routers. These demands call for the use of reprogrammable hardware to provide the required performance and functionality at all network layers. The Field Programmable Port Extender (FPX) provides such an environment for development of networking components in reprogrammable hardware. We present a framework to streamline and simplify networking applications that process ATM cells, AAL5 frames, Internet Protocol (IP) packets and UDP datagrams directly in hardware. We also describe a high-speed IP routing module "OBIWAN" built on top of this framework.

1 Introduction

In recent years, field programmable logic has become sufficiently capable to implement complex networking applications directly in hardware. The Field Programmable Port Extender has been implemented as a flexible platform for the processing of network data in hardware at multiple layers of the protocol stack. Layers are important for networks because they allow applications to be implemented at a level where the insignificant details are hidden. At the lowest layer, networks need to modify the raw data that passes between interfaces. At higher levels, the applications process variable length frames or packages as in the Internet Protocol. At the user-level, applications may transmit or receive messages in User Datagram Protocol (UDP) messages. An important application for the network layer is routing and forwarding packets to other network nodes.

2 Background

In the Applied Research Lab at Washington University in St. Louis, a rich set of hardware components and software for research in the field of ATM and active networking has been developed. The modules described in this paper are primarily targeted to this kit, though the design is written in portable VHDL and could be used in any FPGA-based system.

[⋆] This research was supported in part by NSF ANI-0096052 and Xilinx Corp.

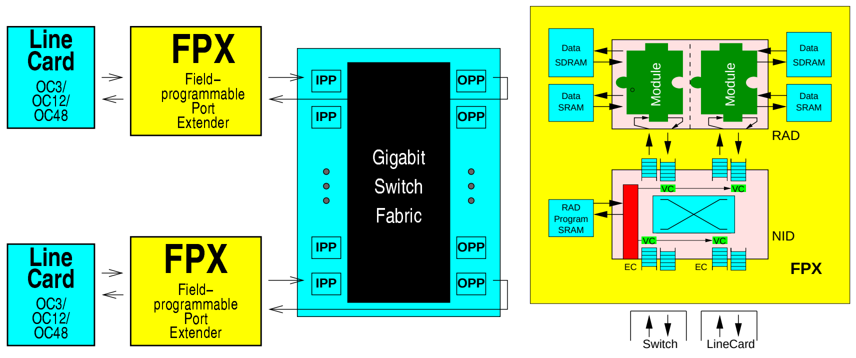
2.1 Switch Fabric

The central component of this research environment is the Washington University Gigabit Switch (WUGS, [1]). It is a fully featured 8-port ATM switch, which is capable of handling up to 20 Gbps of network traffic. Each port is connected through a line card to the switch. The WUGS provides space to insert extension cards between the line cards and the switch itself.

2.2 Field Programmable Port Extender

The Field Programmable Port Extender (FPX, [2]) provides reprogrammable logic for user applications. A configuration of the switch and the FPX is illustrated in Figure 1(a).

The FPX contains two FPGAs: the Network Interface Device (NID) and the Reprogrammable Application Device (RAD). The NID interconnects the WUGS, the line card and the RAD via a small switch. It also provides the logic to dynamically reprogram the RAD. The RAD can be programmed to hold user-defined modules. Hardware based processing of networking data is made possible that way. The RAD is also connected to two SRAM and two SDRAM components. The memory modules can be used to cache cell data or hold large tables. Figure 1(b) illustrates the major components on an FPX board.



(a) WUGS configuration (b) FPX components

Fig. 1. The Washington University Gigabit Switch (WUGS)

2.3 FPX Modules

User applications are implemented on the RAD as modules. Modules are hardware components with a well-defined interface which communicate with the RAD and other infrastructure components. The basic data interface is a 32-bit wide, Utopia-like interface. The data bus carries ATM header information, as well as the payload of the cells. The other signals in the module interface are used for congestion control and to connect to memory controllers to access the off-chip memory [3]. The complete module interface is documented in [4].

Usually, two application modules are present on the RAD. Typically, one handles data from the line card to the switch (ingress) and the other handles data from the switch to the line card (egress). As with the Transmutable Telecom System [5], modules can be replaced by reprogramming the FPGA in the system at any time. In the case of the FPX, this functionality occurs via partial reprogramming of the RAD FPGA. Modules can be replaced on the fly and independently of each other. A reconfiguration component performs a handshaking protocol with the modules to prevent loss of data.

3 Network Wrapper Concept

Network protocols are organized in layers. On the ATM data link layer, data is sent in fixed size cells. To provide variable length data exchange, a family of ATM Adaption Layers exists. The ATM Adaption Layer 5 (AAL5) is widely used to transport IP data over ATM networks. The Network layer uses IP packets to support routing through multiple, physically separated networks.

Components have been developed for the FPX that allow applications to handle data on different levels of abstraction. A similar implementation exists for IP over Ethernet and the corresponding network layers [6]. On the cell level, a Start of Cell (SOC) signal is given to an application module. For AAL5 frame based applications, Start-of-Frame (SOF) and End-of-Frame (EOF) signals indicate the beginning or the end of an AAL5 frame, respectively. An additional data-enable signal indicates whether valid payload data is being sent.

Translation steps are necessary between layers. A classical approach would be to create components for each protocol translation, for instance from cell level to AAL5 frame level. There would need to be a component for the reverse step as well, in our example from the frame level back to the cell level, *i.e.*, segmentation. In a new approach, we combine these two translation units into one component, which has four interfaces

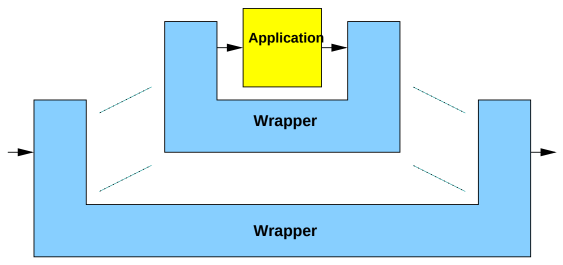


Fig. 2. Wrapper concept

as a consequence: two to support the lower level protocol and two to provide a higher level interface, respectively. Also the two units are connected to each other. This is useful to exchange additional information or to bypass the application. Latter is done in the cell processor (section 3.1).

When an application module is embedded into the new translation unit, it gets a shape like the letter U (see Figure 2). Regarding the data stream, the application only connects to the translating component, which wraps up the application itself. Therefore we will refer to the surrounding components as *wrappers*.

To support higher levels of abstraction, the wrappers can be nested. Since each has a well defined interface for an outer and an inner protocol level, they fit together within each other, as shown in Figure 2. As a result, we get a very modular design method to support applications for different protocols and levels of abstraction. Associating each wrapper with a specific protocol, we get a layered model comparable to the well known OSI/ISO networking reference model. This modularity gives application developers more freedom in their designs. They can choose the level of abstraction they needs for their specific application, while not needing to deal with the handling of complicated protocol issues, like frame boundaries or checksums.

3.1 Cell based processing

At the lowest level of abstraction, data is sent in fixed length cells. Applications or wrappers working on that protocol level typically process the ATM header and filter cells by their virtual channel. FPX Modules communicate with software via control cells, ATM cells with a well-defined structure used to perform remote configuration.

FPX Cell Processor The wrapper on the lowest level is the cell processor (see Figure 3). It performs every necessary step on the cell level that is common to all FPX modules. First of all, incoming ATM cells are checked against their Header Error Control (HEC) field, which is part of the 5 octet header. An 8 bit CRC is used to prevent errored cells from being misrouting. If the check fails, the cell is dropped.

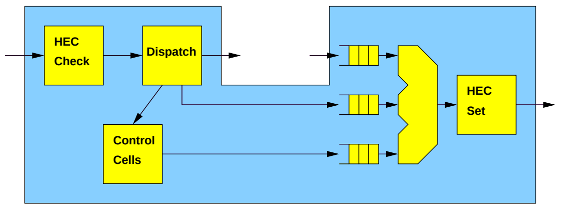


Fig. 3. Cell processor

Accepted cells are queried about their virtual channel information in the next step. The cell processor distinguishes between three different flows:

1. The cell is on the data VC for this module. In this case, the cell will be forwarded to the inner interface of the wrapper and thus to the application.
2. The cell is on the control cell VC and is tagged with the correct module ID. Control cells are processed by the cell processor itself.
3. None of the above, *i.e.*, this cell is not destined for this module. These cells are bypassed and take a shortcut to the output of the cell processor.

The cell processor provides three FIFOs to buffer cells from either of the three paths. A multiplexer combines them and forwards the cells to their last stop. Just before they leave the cell processor, a new HEC is computed.

The control cell handling inside the cell processor is designed to be very flexible, thus making it easy for application developers to extend its functionality to fit the needs of their modules. Since user applications will typically support more types of control cells than the standard types, extendibility was an important goal in the design of the cell processor. A control cell processing framework handles common cases like data integrity.

3.2 Frame based processing

To handle data with arbitrary length over ATM networks, data is organized in frames, which are sent as multiple cells. Several adaption layers have been specified, which differ in the property of being connection-oriented or connectionless, in the ability to multiplex several protocols over one virtual channel and to reorder cells during transmission.

In the ATM Adaption Layer 5 (AAL5) [7, 8] datagrams or frames of arbitrary length are put into protocol data units (PDU). A PDU's length is always a multiple of 48 octets, because a PDU is sent as a multiple of ATM cells. One bit in the ATM header, the user bit of the PTI field, is used to indicate whether a cell is the last one of a PDU. The last 8 octets of the PDU are used by a trailer, which contains information about the actual length of the frame and a 32 bit CRC to ensure data integrity.

The frame processor is a wrapper module for the FPX to handle AAL5 frame data. Its interface is designed to give application modules a more abstract view of the data. The frame processor replaces the Start-of-Cell signal with three signals, namely Start-of-Frame (SOF), End-of-Frame (EOF) and Data-Enable (DataEn). As the name indicates, SOF indicates the transmission of a new frame. The Data-Enable signal indicates valid payload data. It can be seen as an enable signal for the data processing application. It is completely independent from the cell structure. Applications can therefore resize frames or append data very easily. Also generating new frames is now more convenient. After the EOF signal, two more words are sent, which contain the option and the length field and an indication whether the frame was transmitted correctly.

3.3 IP Packet Processing

The IP processor was developed to support IP based applications. It inherits the signalling interface from the frame processor and adds a Start-of-Payload (SOP) signal, to indicate the payload after the IP header. This wrapper serves three primary functions:

1. Checking the IP header integrity, *i.e.*, the correctness of the header checksum. Corrupt packets are dropped.
2. Decrementing the Time To Live (TTL) field. As of RFC 1812 [9] all IP processing entities are required to decrement this field. Once this field reaches zero, the packet should not be forwarded any more. This is to prevent packets from looping around in networks due to mis-configured routers.
3. Recompute the length and the header checksum on outgoing IP packets.

An IP header has the length of 20 bytes, or 5 words.[1] The whole header must pass the header check before any decision about its integrity can be made. The IP Processor computes and then compares the header checksum. On a failure, the packet is not forwarded to the application. If the Time-To-Live field of an incoming packet is already zero, the packet is also dropped and an ICMP packet is sent instead. On outgoing IP packets the length field in the header and the header checksum are set accordingly. Therefore, a whole packet has to be buffered, before the actual length can be determined.

We have also implemented a processor to handle User Datagram Protocol (UDP) packets. This wrapper gives an indication to the embedded application when the data contains the payload of such a packet and handles the UDP checksum and length field. The UDP processor is used for an application that is not described in this paper.

4 IP router OBIWAN

To demonstrate the functionality of our framework, we will now present a fully functional Internet Protocol router, which we call OBIWAN (Optimal Binary search IP lookup for Wide Area Networks). The router uses one external SRAM module and an internal bitmap. Routing entries can be configured with control cells as mentioned in section 3.1. The router extracts the destination IP address of incoming IP packets, which are encapsulated in AAL5 frames, buffers the data while the IP lookup is performed and forwards the packets with the VCI being replaced according to the next hop information. The WUGS switches the packets to one of the eight ports according to the new VCI. OBIWAN can operate at full line speed, *i.e.*, 2.4 Gbps. It is designed to even work at the worst case, which is one IP packet in every ATM cell, or one lookup every 16 clock cycles. This provides a number of up to 6.25 million IP packets per second.

4.1 Lookup Algorithm

The lookup algorithm that we used for our implementation is a binary search over prefix lengths using hash tables. This algorithm is documented in detail in [10, 11]. The basic algorithm uses a hash table for each prefix length. A hash key and a value can be determined from the prefix and will be stored in the table. When a lookup is requested, the basic algorithm performs a binary search for the best matching prefix. It starts with the prefix length 16, and depending on a match it continues the search with a longer, *i.e.*, 24 bits or a shorter, *i.e.*, 8 bits length, until the longest matching prefix length has

[1] This applies to the vast majority of IP packets that do not contain any IP options.

been determined. The number of iterations, *i.e.*, the depth of the binary search tree of the basic scheme is five.

Though the algorithm given above performs very well already, in terms of memory accesses, it still requires 3 simultaneous lookups to guarantee line speed operation. Reducing the number of memory accesses is our main goal in improving the algorithm above. This can be done by reducing the depth of the binary search tree. First, the algorithm combines two adjacent prefix length by expanding the shorter prefix to two more specific routing entries, which reduces the maximum depth of our search tree by one.

Many routing entries have a prefix length shorter or equal to 16 and our analysis of network traffic has shown, that around 50% of all routed IP packets will take one of these routes. These results led us to implement a bitmap for the first 16 prefix bits, which indicate, whether a longer prefix exists in the hash tables. This does not only reduce the binary search tree by another level, but also improves the lookup speed for every second IP packet significantly.

To reduce the overall memory consumption of the hash tables and to reduce the probability of collisions, we only use two relatively large hash tables. One for all prefix lengths from 17 to 24, and another one for the range 25 to 32. Our final configuration gives us two hash tables with four buckets per key each. The table for the shorter prefixes contains 64k entries with four buckets per word, the other 32k entries with two buckets per word. The key and value functions use a simple bit extraction algorithm, which is very efficient in hardware. Tests have shown, that this configuration is useful with real routing tables and does not give many collisions. Despite the fixed number of buckets per hash table entry, collisions can still be resolved by expanding a prefix to more entries, thus moving them to other locations in the table.

For each match in a hash table, the lookup algorithm should determine a next hop information. Since this information is only necessary at the very end of the lookup, we separated it from the hash tables and put them in a shadow-table to avoid unnecessary memory bandwidth usage.

4.2 Implementation

When IP packets first come into the router, the IP destination address is extracted and forwarded to the actual IP lookup engine. At the same time, the whole packet is stored in a FIFO (figure 4). The packet can only be forwarded when the next hop information, which is a new VCI, is available from the lookup engine. While the packet is written to the FIFO, the payload words are counted. The number is then put in a separate queue. On the output side of the router, the IP packets are forwarded, with the VCI being replaced, as soon as the next hop information is available from the IP lookup. There is also a queue for next hop information, in case the output port is congested and IP packets cannot be forwarded fast enough.

The actual IP lookup engine (see Figure 5) takes a destination IP address on its input and delivers a next hop information (VCI) on its output after some time. The IP lookup engine works strictly in order. The IP address is first checked against the internal bitmap, which is located in on-chip memory. As mentioned earlier, the bitmap contains the information, if the longest prefix for this address has at most 16 bits. Therefore the

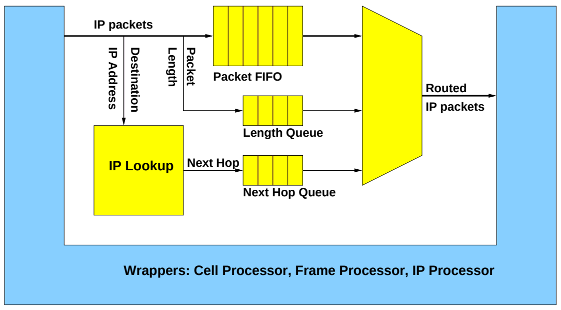


Fig. 4. IP router module OBIWAN

bitmap is $2^{16} = 64kb$ in size. If there is no longer prefix than 16 bits, the next step, the binary search over hash table lookups, is skipped and a next hop address is forwarded. A next hop address encodes the location of the next hop information in SRAM. Otherwise the IP address is forwarded to one of two engines, which perform the binary search.

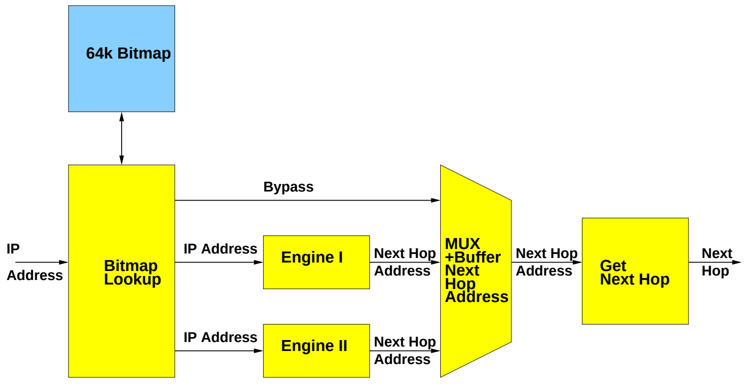


Fig. 5. IP routing lookup (block diagram)

The binary search units (see Figure 6) start with a prefix length of 24 bits, compute a key/value pair for the hash table and set the address for memory access. Because of buffers at the chip boundaries and inside the lookup engine, it takes 6 clock cycles, until the data from memory can be evaluated. During that time, some pre-computations are done: the next possible longer and shorter prefix lengths are determined, the corresponding key/value pairs and memory addresses are computed and stored. When the data words are finally available, the values of all buckets are compared to the value corresponding to the current IP address. On a match, the address for the longer prefix is selected for the next iteration, otherwise the address for the shorter prefix is chosen.

A match also updates the next hop address for the best matching prefix, which initially points to the 16 bit prefix of the IP address. After a total of 3 iterations, the next hop address is forwarded to retrieve the next hop information for the current IP address.

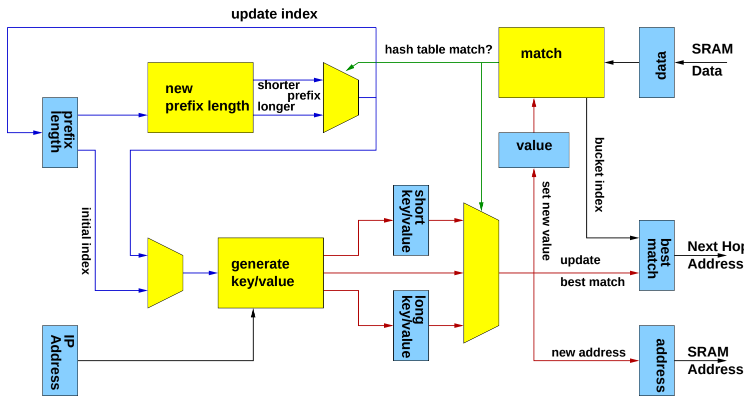


Fig. 6. Binary search on hash tables

Before we can read the next hop information from memory, we have to reorder the next hop addresses, coming from three different locations. Since some next hop addresses bypass the binary search, they may arrive out of order. The next hop information is read from a shadow table in memory.

5 Implementation Results

Our framework and the IP router are designed for the FPX. The system clock on the FPX is 100 MHz and the FPGA used is a Xilinx Virtex E 1000-7. The following table gives the size (in lookup tables) and the maximum speed of our components on the FPX hardware.

Wrapper/Module	LUTs	Speed (MHz)
Cell Processor	760	118
Frame Processor	312	116
IP Processor	680	109
OBIWAN	1196	111

6 Conclusions

We have presented a framework for IP packet processing applications in hardware. Although our current implementation was directed for use in the Field Programmable Port Extender, the framework is very general and can be easily adapted to other platforms.

We introduce the concept of U-shaped wrappers, where each handles a particular protocol level. Unlike the traditional pre-/postprocessor separation, the U shape allows to put components logically together, increasing the flexibility and reducing the number of cross-dependencies. The common interface between layers also lowers the learning curve.

The framework is useful for application developers, who are designing in the area of IP networking and ATM. We also presented a fully functional IP router, which runs at 2.4 Gbps (OC-48) and is based on our framework. The entire router consisting of all components (control cell processor, memory interface, frame processor, and OBIWAN router) have been synthesized for a Xilinx Virtex and fit within 17% of an XCV1000E FPGA. This indicates that the layering employed does not adversely affect size or performance. Besides the basic router, there is still plenty of space for future, application-defined functions, such as hardware-accelerated active networking or security processing.

References

1. Tom Chaney, J. Andrew Fingerhut, Margaret Flucke, and Jonathan S. Turner. Design of a gigabit ATM switch. Technical Report WU-CS-96-07, Washington University in St. Louis, 1996.
2. John W. Lockwood, Jonathan S. Turner, and David E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *Proceedings of FPGA 2000*, pages 137–144, Monterey, CA, USA, February 2000.
3. John W. Lockwood, Naji Naufel, Jonathan S. Turner, and David E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *Proceedings of FPGA 2001*, Monterey, CA, USA, February 2001.
4. David E. Taylor, John W. Lockwood, and Sarang Dharmapurikar. Generalized RAD Module Interface Specification on the Field Programmable Port Extender (FPX). http://www.arl.wustl.edu/arl/projects/fpx/references, January 2001.
5. Toshiaki Miyazaki, Kazuhiro Shirakawa, Masaru Katayama, Takahiro Murooka, and Atsushi Takahara. A transmutable telecom system. In *Proceedings of Field-Programmable Logic and Applications*, pages 366–375, Tallinn, Estonia, August 1998.
6. Hamish Fallside and Michael J. S. Smith. Internet connected FPL. In *Proceedings of Field-Programmable Logic and Applications*, pages 48–57, Villach, Austria, August 2000.
7. Juha Heinanen. Multiprotocol encapsulation over ATM adaptation layer 5. Internet RFC 1483, July 1993.
8. Peter Newman et al. Transmission of flow labelled IPv4 on ATM data links. Internet RFC 1954, May 1996.
9. Fred Baker. Requirements for IP version 4 routers. Internet RFC 1812, June 1995.
10. Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM ’97*, pages 25–36, September 1997.
11. Marcel Waldvogel. *Fast Longest Prefix Matching: Algorithms, Analysis, and Applications*. Shaker Verlag, Aachen, Germany, April 2000.