Synthesis of High-Performance Packet Processing Pipelines

Cristian Soviani*
Columbia University, CS Dept.
New York, New York
soviani@cs.columbia.edu

Ilija Hadžić
Bell Labs, Lucent Tech.
Murray Hill, New Jersey
ihadzic@bell-labs.com

Stephen A. Edwards†
Columbia University, CS Dept.
New York, New York
sedwards@cs.columbia.edu

ABSTRACT

Packet editing is a fundamental building block of data communication systems such as switches and routers. Circuits that implement this function are critical and define the features of the system. We propose a high-level synthesis technique for a new model for representing packet editing functions. Experiments show our circuits achieve a throughput of up to 40Gb/s on a commercially available FPGA device, equal to state-of-the-art implementations.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Logic Design—*Design Aids*

General Terms

Algorithms

Keywords

Packet processors, Networking, FPGAs, high-level synthesis

1. INTRODUCTION

A packet switch (or router) is a basic building block of data communication networks. Its primary role is to forward packets based on their content, specifically header data at the beginning of the packets. As part of this forwarding operation, packets are classified, queued, modified, transmitted, or dropped.

The forwarding algorithms used in most switches are simple to facilitate efficient hardware implementation. However, they are tedious to code at the RT level because performance demands them to be deeply pipelined circuits that operate on many bits in parallel (e.g., 128 or 256) and interact with high-speed FIFOs. This makes for complicated datapaths and controllers.

We propose a high-level synthesis technique for packet editing processors. We target FPGAs, although ASICs are also possible.

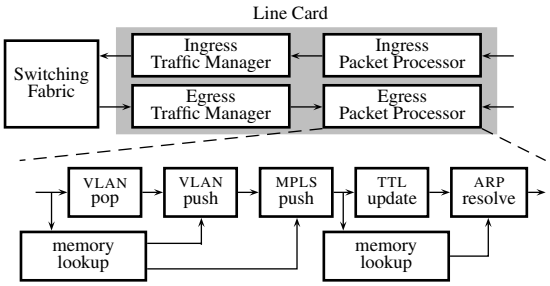


Figure 1: A switch with detail of a packet processing pipeline.

Packet editing is a pivotal function of most switches, which also include traffic managers, a switching fabric, and other components (Figure 1). We propose a novel way to model algorithms that transform input into output packets and present a synthesis procedure to translate these models into efficient VHDL code. Our technique produces packet editing blocks that are easily connected in pipelines.

We produce circuits for FPGAs that can sustain 40 Gbps throughput on industrial examples, equal to state-of-the-art switches (a 10 Gbps ASIC was novel in 2003 [6]) and are vastly easier to write and maintain than RTL descriptions.

2. PACKET PROCESSORS AND EDITING

Figure 1 is a block diagram of a packet switch consisting of line cards (we only show one) connected to a switching fabric that transfers packets. We focus on designing the line cards, which provide network interfaces, make forwarding and scheduling decisions, and, most critically, modify packets according to their contents.

Our synthesis technique builds components in the ingress and egress packet processors. A packet processor is a functional block that transforms a stream of input packets into output packets. These transformations consist of adding, removing, and modifying fields in the packet header. In addition to headers defined by network protocols, the switch may add its own control headers for internal use.

Packet processors perform complex tasks through a linear composition of simpler functions. This model has been used for software implementations on hosts [7] and on switches [4]. A less suitable architecture for hardware is a pool of task-specific threads that process the same packet in parallel without moving the packet [1].

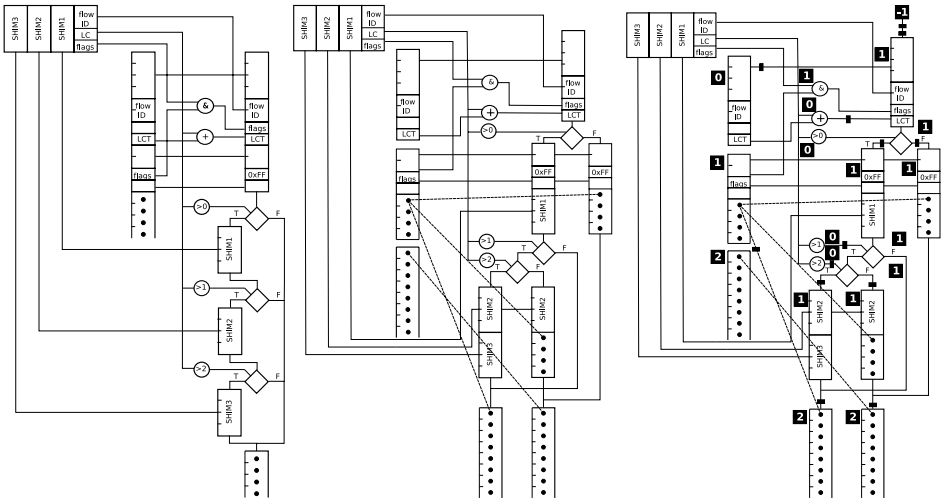
We use a unidirectional, linear pipeline model that simplifies the implementation without introducing major limitations. For example, the loops in Kohler's IP router [4] only handle exceptions. We would do this with a separate control processor.

While the logical flow can fork and join, we implement only linear pipelines that can use flags to emulate such behavior. Non-linear pipelines are more complicated and would not improve throughput.

*This work was done while Soviani was at Bell Labs.

†Edwards and his group are supported by an NSF CAREER award, gifts from Intel and Altera, and by the SRC.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.



(a) The initial specification (b) Split into 64-bit words (c) Read cycle indices and delay bubbles added

Figure 2: Steps in synthesizing a PEG model for the MPLS push module in Figure 1.

```
Restruct(node n, pending bits v, word size w)
  clean-visit ← true if v is empty
  if clean-visit and cache contains n then
    return cache[n]
  case type of node n of
  output data :          ≥ 1 bytes, one successor
    append n to v        put n in current word
    if v is w∗8 bits then      finished word
      n′ ← build-node(v)       next word node
      n″ ← Restruct(successor of n, (), w)
      Make n″ the successor of n′
    else
      n′ ← Restruct(successor of n, v, w)
  conditional :
    n′ = copy of the conditional n
    for each successor s of n do
      n″ = Restruct(s, v, w)
      Add n″ as a successor of n′
  if clean-visit then
    cache[n] ← n′
  return n′              the restructured node for n
```

Figure 3: Structuring a packet map into words

If a packet needs to be dropped or forwarded to the control processor, we set flags in the control header and perform the action at the end of the pipeline. This guarantees every processing element sees all the packets in the same order; packet reordering is usually done in a traffic manager, a topic beyond the scope of this paper.

Figure 1 shows a packet processing pipeline that edits the Virtual Local Area Network (VLAN) tag of an Ethernet packet [3] and adds a Multi Protocol Label Switching (MPLS) label [8], based on unique flow identification (FlowID). We assume a previous stage has performed flow classification and prepended a control header with a FlowID.

Both the VLAN push and MPLS push modules insert additional headers after the Ethernet header, while the Time-To-Live (TTL) update and Address Resolution Protocol (ARP) resolution modules only modify existing packet fields. The VLAN pop module removes a header. While this pipeline is simple, real switches just perform more such operations, not more complicated ones.

Thus, packet processing amounts to adding, removing, and modifying fields. Even the flow classification stage, which often involves a complex search operation, ultimately produces a modified header. We refer to these operations as *packet editing*; it is the fundamental building block of a packet processor.

In addition to the main pipeline, Figure 1 shows two memory lookup blocks. These blocks store descriptors that define how to edit the headers (e.g., how many MPLS labels to add). Here, the FlowID is a index into descriptor memory. A memory lookup module is any component that takes selected fields and produces data for a downstream processing element (e.g., IP address search, present in all IP routers, is a form of generalized memory lookup). Flow classification is thus packet editing with memory lookups.

Modules that use memory lookup assume a previous pipeline stage issued the request, which is processed in parallel to hide memory latency. We do not synthesize memory lookup blocks, but can generate requests and consume results. Because our pipelines preserve packet order, simple FIFOs suffice for memory interfaces.

Hence, we model packet processors as linear pipelines whose elements have four types of ports: input from the previous stage, output to the next, requests to memory, and memory results. Each processing element must be capable of editing packets based on their content and data from memory.

3. RELATED WORK

Kohler et al. [4] propose the CLICK domain-specific language for network applications. It organizes processing elements in a directed dataflow graph. CLICK specifies the interface between elements to facilitate their assembly. Although originally for software, Kulkarni et al. [5] propose a hardware variant called CLIFF, which represents its elements in Verilog. Schelle et al.'s [9] CUSP is similar. Brebner et al. [1] add speculative multi-threaded execution.

Unlike CLIFF/CUSP, we synthesize our modules instead of assembling library components. CLICK, furthermore, defines fine-grained elements whose connection has substantial handshaking overhead; our larger processors minimize this problem.

Our approach differs from classical high-level synthesis (c.f., De Micheli [2]) in important ways. For example, we always use as-soon-as-possible scheduling for speed; classical high-level synthesis considers others. Furthermore, most operations are smaller than the muxes needed to share them, so we do not consider sharing.

Our technique differs most from classical high-level synthesis in its choice of computational model. Rather than assume data are stored and retrieved from memories, we assume data arrives and departs a word at a time from FIFOs, thus our scheduling mostly considers the clock cycle in which data arrive and can leave.

4. THE PACKET EDITING GRAPH

Although the behavior of a node in a packet editing pipeline can be modeled at the RT level, doing so is awkward for deeply pipelined circuits that operate on many bits in parallel. By contrast, our Packet Editing Graph (PEG) model describes such nodes in an implementation-independent way that is much easier to design and modify, and it can be synthesized into efficient circuitry.

Figure 2a shows a PEG for a simplified MPLS push module. The MPLS protocol adds a label to the beginning of a packet that acts as a shorthand for the IP header. When another switch receives the packet, it uses separate rules to forward the packet. The module in Figure 2a inserts up to three MPLS labels according to an in-memory descriptor. It also updates the label count (LCT) to reflect any added labels, replaces the FlowID field with the one from the descriptor and updates various other flags. Replacing the FlowID maps a set of MPLS tunnels to the set of next-hop destinations.

A PEG is an acyclic, directed graph consisting of inputs (the packet itself and data from a memory lookup block, drawn as rectangles in the top left section of Figure 2a); arithmetic and logical operators (the circular nodes in the middle of the figure); outputs (an output packet map, shown on the right side of the figure; and data used to generate memory lookup requests, not shown in this example) and the connections among those. In the figure, time flows from top to bottom and data flows from left to right.

The packet map—the control-flow graph on the right—is the most novel aspect of a PEG. The bits of the output packet are assembled by following a downward path. A diamond-shaped node is a conditional: control flows to one of its successors depending on the value of the predicate. Conditionals allow bits to be inserted and deleted from the output packet. The final node, marked with dots, copies the remainder of the input packet to the output.

5. THE SYNTHESIS PROCEDURE

The challenge in synthesizing a circuit from a PEG is converting the flat, bit-level specification into the sequential word-level implementation needed for performance. This is tricky because things generally do not fall on word boundaries and some results may depend on bits that arrive later. Moreover, a PEG allows conditional insertions and removals, so there is not always a simple mapping between the word in which a byte is received and when it is sent.

Our synthesis procedure analyzes the PEG, establishes the necessary mapping, and builds a datapath and controller.

5.1 Wrappers and the Module Interface

We create synthesizable RTL by instantiating a manually-written wrapper around a core synthesized from a PEG. The wrapper adapts the core interface to the specific FIFO protocol.

Figure 5 illustrates a typical wrapper. Here, we do not show any memory input/output ports, which are also handled by the wrapper. They transfer exactly one word per packet, so the core sees the input port as a parameter and the output port as a register.

Cores receive and send packets over a w-byte parallel interface ($w = 16$ is typical). The module sees the input packet as a sequence of w-byte words arriving on the *idata* port (Figure 5). Similarly, the output is generated as a sequence of w-byte words on the *odata* port. Three flags on each port indicate packet boundaries: *sop* denotes the start, *eop* the end, and the *mod* signal indicates the number of bytes in the final word in a packet.

A core communicates with the wrapper through three more signals. *rd* and *wr* request data from the input and indicate when data are written to the output. *Suspend* instructs the module to stall. The wrapper in Figure 5 simply stalls the module when input data are not available or when the output cannot accept new data.

5.2 Splitting Data into Words

Our synthesis procedure begins by dividing the input and output packets on word boundaries using the procedure of Figure 3. Dividing the input packet is straightforward; reshaping the output packet map is complicated because of conditionals. Figure 2b shows the result of this on the MPLS example of Figure 2a.

We restructure the packet map so conditions are only checked at the beginning of each word to guarantee that only complete words are generated in every cycle (except the last, a special case). For example, the > 0 condition in Figure 2a has been moved four bytes earlier in Figure 2b and the intervening four bytes have been copied to the two branches under the conditional to maintain I/O behavior.

The algorithm in Figure 3 recursively walks the packet map to build a new one whose nodes are all w bytes long (the word size). Each node is visited with a vector v that contains bits that are "pending" in the current word. Output nodes are added to this vector until $w * 8$ bits are accumulated, at which point a new output node is created by build-node, which assembles the saved bits in v. The algorithm handles conditionals by copying the condition to a new node n', which is placed at the beginning of the current word, and visiting the two successors under the conditional. The same v is copied to each recursive call, effectively duplicating the rules for the bits that appeared before the conditional in the current word.

This has the potential of generating an exponentially-large tree, but in practice protocols are designed to avoid this. For example, there are four paths in Figure 2b, but we find they lead to only two different states: the one- and three-label cases converge since they require the same alignment; the zero and two cases are similar.

We handle reconvergence by maintaining a cache of nodes that can be reused. If a node visit is "clean," i.e., the pending vector is empty, the cache is checked for an earlier visit that can be reused.

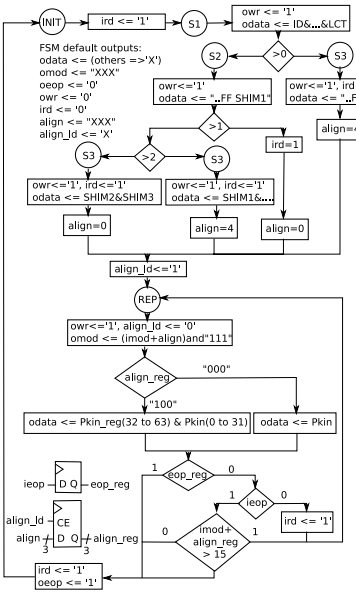


Figure 4: Controller synthesized from the packet map in Figure 2c

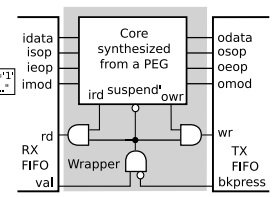


Figure 5: A core module and wrapper.

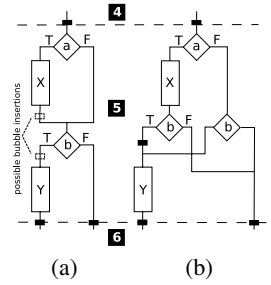


Figure 6: Scheduling within a read cycle.

5.3 Assigning Read Cycle Indices

After splitting the packet map into words, we label each node with the logical cycle in which its data are available. These indices (black boxes in Figure 2c) are like clock cycles, but practically an index may map to several clocks if the controller causes a stall.

The first input word index is zero, the second is one, etc. The rest are computed from causality: the index of a node is the highest index of all its predecessors. Constant nodes and memory inputs, assumed to be present in all cycles, are therefore ignored.

5.4 Scheduling

Once read cycle indices are assigned, we insert "bubbles" that correspond roughly to pipeline stages. We draw these as black rectangles in Figure 2c. We insert them according to the following rules: if two indices differ by $k > 0$, at least k bubbles are needed between them; and any two output nodes in the packet map, even with the same index, require at least one bubble between them.

In Figure 2c, two bubbles were inserted between the top-most node and the first output node because the difference between their indices is two. This follows the first rule. The first word cannot be output in cycle 0 because it depends on the *flags* field, which

becomes available in cycle 1. Following the second rule, bubbles were also added after the first conditional because these arcs are between two output nodes. These bubbles are necessary because it is not possible to write two words simultaneously, even though the information for building the second word is available earlier.

To comply with the first rule, exactly k bubbles are inserted on any arc between nodes with different indices. It is harder to comply with the second rule. Consider the example in Figure 6a. We can output X, Y, both, or none, depending on conditions a and b. If both a and b are true, we need two physical cycles; otherwise, one cycle will suffice. If both a and b are false, the output will idle for one cycle, as the data to follow will not be available.

Following the second rule, we may insert a bubble in the two positions in Figure 6a. But if we insert it under X, if a is true and b is false, we spend two cycles instead of one. The solution is to reshape the graph by duplicating the second condition (Figure 6b).

The bubble-insertion algorithm is straightforward. For each node in the original graph, two copies are built: "empty" and "full," which handle control flow when the current cycle has and has not been used for data output respectively. For most nodes, only one copy remains after a sweep that removes unconnected nodes.

5.5 Synthesizing the controller

Once read cycle indices and bubbles have been added, we synthesize the control machine (FSM). Its structure follows the packet map. Bubbles in the packet map become states; we replace them with registers, giving a one-hot encoding. Bubbles adjacent to the leaves are special states that copy data until *ieop*, after which the FSM returns to the initial state.

Figure 4 shows the packet map of Figure 2c translated into an FSM. When an output node is encountered, the data are steered to the output, and the *owr* signal is asserted. The second scheduling rule ensures that at most one output node is found on any path between two states. For paths with no output nodes, *owr* remains deasserted. For each arc index increase, the *ird* signal is asserted to read the next word from the buffer. The first scheduling rule ensures that at most one word will be read between two states. For paths with no index increase, *ird* remains deasserted.

5.6 Handling the end of a packet

All paths in the packet map finally reconverge to no more than w different states, corresponding to no shifting necessary to shifting $w-1$ bytes. Our algorithm merges the end of each path to a common *REP* state that is accompanied by an auxiliary *align* register of size $\log_2(w)$. Any transition that leads to *REP* loads *align*.

The *REP* state performs two tasks. First, it aligns the data using a multiplexer. In Figure 2c, *align* can take only two values, 0 and 4, demanding a two-input multiplexer. Second, if *eop* is active, the FSM can use *align* and *imod* to decide whether an extra cycle is required for the last word and the input FIFO must stall.

5.7 Synthesizing the data path

Combinational nodes translate directly into combinational logic to form the datapath. Bubbles become registers that guarantee any node with read cycle index i has a valid value in the matching cycle. A read cycle index may correspond to several clock cycles, so registers must be able to hold their values. We take a simple approach: values are held when the present state equals the next state. The data path can be stalled by asserting *suspend* signal (Section 5.1) that causes the core module to hold all control and datapath registers. This is a brute-force solution that could be more clever: only the output registers are compelled to stall; data could still propagate within the process unless it would overwrite existing data.

Table 1: Synthesis results for selected modules

Module	Core size		Delay	Throughput
	LUTs	FFs	ns	Gbps
MPLSpush	556	107	3.8	33
TTLEXPupdate	43	20	2.9	44
VLANfilter	11	12	2.9	44
VLANedit	505	125	4.0	32
PPPoEfilter	410	151	3.7	34
PPPoEterm	819	322	4.0	32

6. EXPERIMENTAL RESULTS

We synthesized some 128-bit wide modules from industrial designs for a Xilinx Virtex 4 xc4vlx40–ff668–10 FPGA. We generated VHDL with our synthesis method and fed it to the Xilinx ISE tools for RTL synthesis, placement, and routing.

Table 1 shows the size and performance of each module. We instantiate a module between two FIFO buffers, place and route the circuit and report the delay of the longest register-to-register path; based on this result we calculate the resulting packet processing throughput. For area, we only report the size of the core modules with no pipelining (i.e., not including the wrapper circuitry) in terms of the required number of flip-flop and lookup table primitives for the Virtex 4 device family. As only packet headers flow through the pipeline, we are able to sustain 40 Gbps throughput with a realistic distribution of packet sizes, something only very high-end switches currently achieve.

7. CONCLUSIONS

Establishing a strict formalism for describing packet editing operations (our packet editing graph) allowed us to construct a high-performance hardware synthesis procedure that can be used to create packet processors. The performance of circuits synthesized by our procedure is comparable to the performance of circuits in state-of-the-art switches, while the design entry is done at a much higher level of abstraction than the RTL usually used. The direct benefit is improved designer productivity and code maintainability. Experimental results on modules extracted from actual product-quality designs suggest that our approach is viable.

8. REFERENCES

[1] G. Brebner, P. James-Roxby, E. Keller, and C. Kulkarni. Hyper-Programmable Architecture for Adaptable Networked Systems. In *Application-Specific Arch. and Proc.*, 2004.

[2] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[3] Virtual Bridged Local Area Networks. IEEE 802.1Q, 2003.

[4] E. Kohler et al. The Click Modular Router. *ACM Trans. Comp. Systems*, 18(3):263–297, August 2000.

[5] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a domain specific language to a platform FPGA. In *Proc. Design Automation Conference*, pp. 924–927, San Diego, CA, 2004.

[6] M. V. Lau et al. Gigabit Ethernet switches using a shared buffer architecture. *IEEE Comm. Mag.*, 41(3):76–84, 2003.

[7] S. O'Malley and L. Peterson. Dynamic Network Architecture. *ACM Trans. Comp. Sys.*, 10(2):110–143, 1992.

[8] E. C. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, IETF, January 2001.

[9] G. Schelle and D. Grunwald. CUSP: A modular framework for high speed network applications on FPGAs. In *Proc. Field-prog. gate arrays*, pp. 246–257, Monterey, CA, 2005.