Synthesis and Optimization of Pipelined Packet Processors

Cristian Soviani, Ilija Hadžić, *Member, IEEE*, and Stephen A. Edwards, *Senior Member, IEEE*

***Abstract*—We consider pipelined architectures of packet processors consisting of a sequence of simple packet-processing modules interconnected by first-in first-out buffers. We propose a new model for describing their function, an automated synthesis technique that generates efficient hardware for them, and an algorithm for computing minimum buffer sizes that allow such pipelines to achieve their maximum throughput. Our functional model provides a level of abstraction familiar to a network protocol designer; in particular, it does not require knowledge of register-transfer-level hardware design. Our synthesis tool implements the specified function in a sequential circuit that processes packet data a word at a time. Finally, our analysis technique computes the maximum throughput possible from the modules and then determines the smallest buffers that can achieve it. Experimental results conducted on industrial-strength examples suggest that our techniques are practical. Our synthesis algorithm can generate circuits that achieve 40 Gb/s on field-programmable gate arrays, equal to state-of-the-art manual implementations, and our buffer-sizing algorithm has a practically short runtime. Together, our techniques make it easier to quickly develop and deploy high-speed network switches.**

***Index Terms*—High-level synthesis, model checking, packet editing, pipelines, routers, switches.**

I. Introduction

MOST DATA communication systems, such as packet switches and routers, adopt the architecture shown in Fig. 1. The fabric is responsible for transferring data among the line cards, which do the actual network communication. Most of the switch's intelligence resides in the line cards. They are responsible for understanding protocols and deciding where to forward packets. Thus, much of the value of a switch resides in the algorithms implemented on these line cards.

Most line cards have an ingress packet processor, an ingress traffic manager, an egress packet processor, and an egress traffic manager. Packet processors parse the packet content (typically, headers), construct search keys, perform table lookups, transform the packet according to lookup results, and classify packets into flows. They also collect statistics, analyze arrival patterns, apply filtering policies, and select any packets to be dropped. Traffic managers queue packets waiting to be sent to the fabric or the network and schedule their departures based on traffic patterns, queue states, and flow information generated by the packet processors.

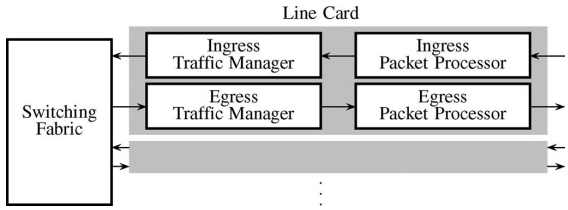

Fig. 1. Architecture of a typical switch. Line cards connected to a shared switching fabric.

Designing a packet-processing circuit for an application-specific integrated circuit (ASIC) or a field-programmable gate array (FPGA) is a challenging task for which few high-level tools are available. A designer often starts by partitioning the system into functional blocks arranged roughly, such as the flow of packets and data, and then hand-codes the blocks at the register transfer level (RTL). The gap between the "whiteboard" system and the circuit renders the development process inefficient and error prone. Contrast this to signal processing, which has tools such as Matlab for high-level model design and verification and for converting such models into RTL [1]. We want a set of analogous tools for the design of packet-processing systems.

The diversity of architectures in a packet-processing system makes developing a comprehensive synthesis system difficult. A common high-level representation for the queue-based architecture of a traffic manager and a pipelined packet processor would be unlikely. Heterogeneity can be found in tools such as Agere's Functional Programming Language. It targets pattern matching and packet parsing but does not describe the behavior of the traffic manager [2], [3].

Thus, the automatic synthesis of packet-processing components is a set of separate related problems. Each type of component is best described in its own domain-specific language and synthesized with a custom set of algorithms. Above these domain-specific synthesis procedures, we envision an integration tool enabling the designer to compose a system from synthesized blocks and explore architectural options.

The work is part of a set of development tools for packet-processing systems, not a comprehensive solution. Our goal is to simplify the creation and maintenance of packet-processing modules with performance comparable to hand-coded designs. We address packet editing functions because they provide much of a switch's value and are tedious to code manually. We propose novel methods for specifying packet transformations (Section IV), generating a synthesizable VHSIC Hardware Description Language (VHDL) code from these specifications (Section V), and sizing the buffers that connect the elements in a pipeline (Section VI).

Manuscript received March 7, 2008; revised June 17, 2008 and September 8, 2008. Current version published January 21, 2009. The work of S. A. Edwards and his group was supported in part by the NSF and in part by the SRC through an award. This paper was recommended by Associate Editor G. E. Martin.

C. Soviani is with Synopsys, Inc., Mountain View, CA 94043 USA.

I. Hadžić is with Bell Laboratories, Alcatel-Lucent, Murray Hill, NJ 07974 USA.

S. A. Edwards is with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: sedwards@cs.columbia.edu).

Digital Object Identifier 10.1109/TCAD.2008.2009168

II. Packet-Processing Pipelines

In general, a pipelined packet processor is a linear sequence of processing elements (modules). Each module performs packet editing, and every packet is processed by all modules in the same order. Logical forks and joins in the data flow are possible by using flags in the control word; a module at the fork point may decide that only a subset of downstream modules should process a packet and set the flags accordingly. This is a way to implement logical branches while preserving packet ordering and near-constant latency.

A packet usually moves through a pipeline in one of three ways: in its entirety (header plus payload); as a descriptor, where the header flows through the pipeline and the payload is stored in a separate memory; and as a series of small units (bursts) that are interleaved with those from other packets to reduce the effective speed of a single packet. Currently, our tools support the first two processing styles and would need extensive but straightforward modification to handle the interleaved burst-style processing. We used descriptors when implementing a real-world system described in detail in Section II-A. We also implemented an experimental system that consisted of an off-the-shelf traffic manager, switching fabric, and an FPGA in which we moved complete packets through the pipeline (the egress traffic manager used in that system guaranteed transmission of a full packet over the SPI4.2 bus).

A central challenge in pipelines such as ours involves nonconstant data rates, which arise because a module may change the size of the packet or need data that appear later in the packet to complete a computation. To avoid having to run modules in a lock step, we connect pairs of modules with first-in first-out (FIFO) buffers. Choosing appropriate sizes for these is important, as overly large FIFOs consume area and power while small ones unnecessarily constrain the throughput.

A. Real-World Example

We used our model to implement a packet-processing pipeline for a gigabit-capable passive optical network (GPON) [4] optical line termination (OLT) fiber-to-the-home system. Each line card serves six physical GPON ports, each supporting 2.5-Gb/s throughput in the egress and 1.25-Gb/s throughput in the ingress,[1] resulting in an aggregate line-card throughput of 15 Gb/s (egress) and 7.5 Gb/s (ingress). Each physical port is shared across 32 subscribers using a passive optical splitter in the field. We implemented traffic management and multiplexing across physical ports with an off-the-shelf Ethernet switching chip and used an FPGA for multiplexing across subscribers on the same splitter. This multiplexing is achieved by mapping the media access control (MAC) address [5] and virtual local area network (VLAN) [6] tag (Ethernet packet fields that identify the packet flow) to the PortID, a GPON-specific field used to uniquely identify a specific packet flow for a subscriber.

[1]We use "ingress" and "egress" from the perspective of the central office, i.e., facing the subscriber and synonymous with "upstream" and "downstream," respectively.

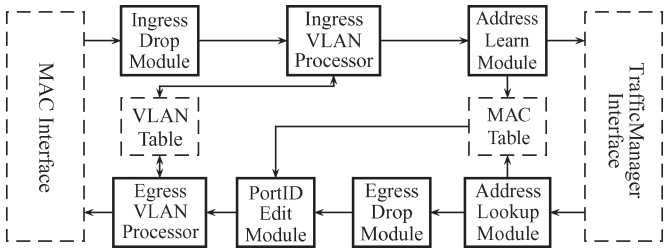

Fig. 2. Packet-processing pipeline.

Fig. 2 shows our GPON system's pipeline. We synthesized the seven highlighted modules using our techniques.

For each arriving packet, our pipeline stores the packet's payload in an external memory (not shown) and constructs an 80-byte packet descriptor that consists of the first 64 bytes of the packet's header followed by a 16-byte control word that carries information such as the packet size, a payload pointer, and various control flags. Our pipeline then transforms the packet descriptor by adding, modifying, and removing header fields; it also modifies fields in the control word. Once the packet reaches the end of the pipeline, the packet's payload is reunited with the now-modified header and is transmitted.

The modules shown in Fig. 2 work together to steer packets. The ingress and egress "drop" modules perform range checks and decide if a packet should be dropped. To drop a packet, these modules set a flag in the packet descriptor's control header. A to-be-dropped descriptor continues to move through the pipeline but is ignored by other modules and discarded at the end of the pipeline, releasing its payload buffer. The two drop modules are typical of modules that do not perform any memory lookups and instead base their processing solely on packet content and their internal state.

Egress and ingress VLAN processors translate between the subscriber- and network-side VLANs. The VLAN tag of an incoming packet is used as an index into the VLAN table, which gives the transformation descriptor. The resulting packet may have one or two VLAN tags added, removed, or modified. Priority bits may also be modified, or a packet may be dropped. The need to handle all these cases and consult a lookup table renders these modules complex.

The address learn and lookup modules do not transform packets but are important nonetheless. They parse each packet to construct a search key built from the MAC address and VLAN tag and then pass this key to the MAC table. The MAC table learns addresses from packets flowing in the ingress direction and searches for information along the egress direction. These passive modules are simple and compact.

The PortID edit module uses the result of the MAC address search to edit the PortID field and hence works in concert with the address lookup module; the latter creates a query whose result is consumed by the former. To hide the latency of consulting the MAC table, we inserted a module (egress drop) whose behavior does not depend on any possible action taken by the PortID edit module. The additional delay from this module ensures that the search result will be available by the time the packet reaches the PortID edit module.

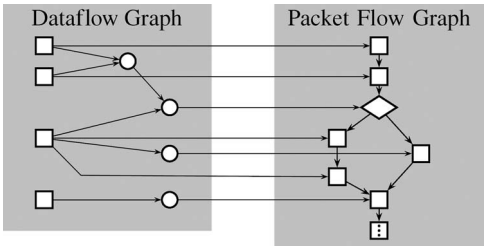

Fig. 3. Simple PEG that copies the first 2 B, and then, depending on the first 3 B, either duplicates the third or modifies it before copying a modified version of the fourth. Thin arrows denote dataflow; thick arrows denote control flow.

III. Memory Lookup Modules

Any practical packet-processing pipeline interacts with memory lookup modules that store and retrieve information from switching, routing, statistics, and metering tables. We hand-coded lookup modules for our GPON system in VHDL, although we would like to also synthesize lookup modules from high-level descriptions. Unfortunately, their behavior differs enough from pipeline modules to warrant a separate tool whose design is outside the scope of this paper. In any case, our synthesized pipeline modules need to interact with such lookup modules; hence, our synthesis technique includes provisions for doing so.

A packet editing module can extract fields from the packet header, construct a search key, and issue a search request to a memory lookup module. If a memory module is fast enough, the lookup may be direct; a single pipeline module may both issue the read request and process the result. In our GPON system, the VLAN table operates this way. More commonly, the lookup is split across two pipeline modules to hide memory latency. Instead of holding the packet while waiting for the result, the module that issues the lookup request passes the packet to the next module and immediately starts forming the request for the next packet. A module several stages downstream will consume the search result and edit the packet accordingly. The MAC table in our GPON system is implemented as a hash table; the bin is located by hashing the MAC address followed by a linear search within the bin. This can take up to 12 clock cycles.

Two conditions must be satisfied to prevent the memory lookup from becoming a throughput bottleneck. First, the time required to propagate the packet from the module that issued the search request to the module that consumes the result must be at least as long as the latency of the lookup module. Second, a lookup module must support at least one search per packet that can arrive during the search latency period.

In the methodology we propose, the designer is responsible for partitioning the function of a packet processor into modules, formally defining the function of each module, and finding the order in which the modules can be connected into a pipeline given the constraints imposed by the memory lookup latencies. Once the design problem is solved at this level, our algorithms can generate circuits that implement the details of the pipeline.

IV. PEG

One of our contributions is the packet editing graph (PEG, Fig. 3), a compiler-style intermediate representation for the operations performed by a packet header editing module. The synthesis procedure we describe in Section V starts from PEG.

PEG is meant to be derived from a human-readable programming language. The concrete syntax we proposed for it elsewhere [7] is meant as an intertool exchange format rather than a language for humans. We generated PEG from a proprietary language in our experiments; however, we could, just as easily, have started from one of the many languages targeted at specifying packet operations, such as Baker [8] or CAL [9].

We had two conflicting goals in designing PEG. It had to offer a natural level of abstraction and enable an efficient synthesis procedure. We aimed it at languages for designers who are experts in system architecture and network protocols, not necessarily ones versed in RTL digital design. Such designers see packet header editing as arithmetic and logical operations performed on fields and not as sequences of operations or state machines, and both the high-level source language and PEG need to reflect this. However, for performance, the semantics of PEG needed to be close to that of the synthesized hardware to minimize the need for sophisticated optimization algorithms.

Our solution is a compromise; we describe arithmetic and logical operations in a hardware-like style and have the tool infer and synthesize sequential behavior. In particular, a PEG description is agnostic about word length; however, we generate a circuit for a given length. Our synthesis algorithm is mechanical but not purely syntax directed. Although it does not perform optimization, we designed it to generate circuits that are well suited to optimization with common logic synthesis algorithms such as retiming.

Three things differentiate PEG from more traditional representations. Unlike the register-transfer style typically used in high-level synthesis [10], PEG is more dataflow-like in that it models memoryless pipeline stages that each consume and produce a single packet stream (registers are inferred during synthesis). However, unlike typical dataflow models, such as synchronous dataflow [11], PEG allows for data-dependent processing rates by providing a simple facility for inserting and deleting bytes flowing past. Finally, it is agnostic about word width; the same PEG specification can just as easily produce a pipeline that produces a 128-bit word each clock cycle as one that produces a single byte. We know of few other formalisms that support variable blocking factors.

A. Structure of the Graph

A PEG is a pair of directed acyclic graphs (Fig. 3). The dataflow graph describes how the packet is transformed; its sources are input nodes (rectangles), which are fields in the input packet or auxiliary inputs, e.g., from memory lookup modules. The nodes in the dataflow graph describe arithmetic and logical operators, which we draw as circles.

The sinks in the dataflow graph are nodes in the packet flow graph—the second directed acyclic graph. This graph describes how to assemble the output packet from bits generated by the dataflow graph. It contains two types of nodes: bit sequences (rectangles) and conditionals (diamonds). Bits from the dataflow graph affect how control flows at conditional nodes in the packet flow graph. The sinks in the packet flow graph are

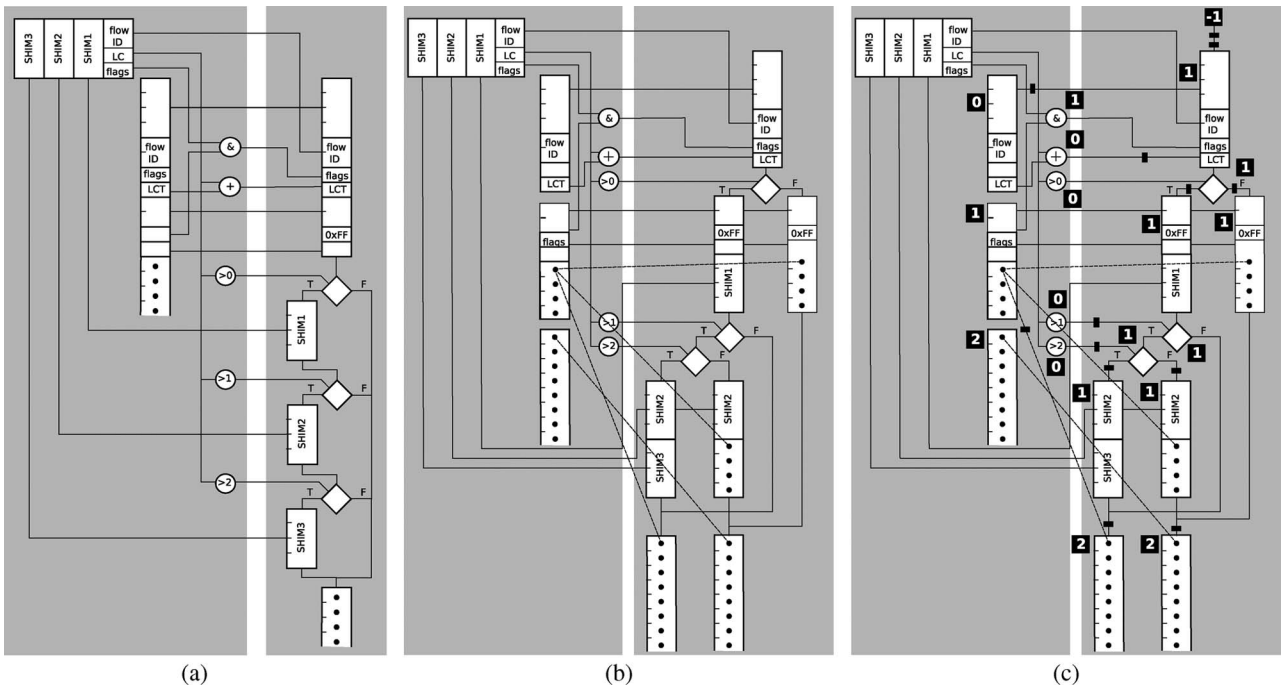

Fig. 4. Synthesizing a PEG model. (a) Initial specification labels the bits in the incoming packet header and from auxiliary inputs, processes them with an acyclic network of arithmetic and logical operations, and assembles them according to an acyclic control-flow graph. (b) Synthesis divides the packets into words (here, 64 b). (c) Assigns read cycle indexes and inserts delay bubbles.

nodes marked with dots, which copy the remainder of the input packet (i.e., its payload) to the output unchanged.

In our diagrams, data flow from left to right, and time (i.e., bits entering and leaving the module) flows from top to bottom. Fig. 4(a) shows a more elaborate PEG. Depending on the descriptor retrieved from the memory (shown in the top left corner), this module inserts up to three new fields in the packet [labeled *SHIM* in Fig. 4(a)] and modifies the control header. We derived the function of this module (including the packet field names) from a packet-processing pipeline that pushed multiprotocol label switching [12] labels to the packet header. We use it here as an example to explain the features of the synthesis process.

V. Synthesizing the PEG

Our synthesis procedure translates a PEG into synthesizable VHDL (RTL) targeting FPGAs, although it could also be used for ASICs or even software.

The fundamental synthesis challenge is translating the purely functional PEG description into an RTL model that computes the function over multiple clock cycles. Input and output packets take multiple clock cycles to arrive and depart; how many depends on the width of the buses and the length of the packets. Consequently, different fields of the packet may arrive in different clock cycles; the module's controller has to ensure that the right operations are performed in each clock cycle.

PEG's ability to insert and remove data from the output packet further complicates things. Inserting data may demand that the input be stalled until the module is ready to pass more input data to the output. Deleting data from the output packet may mean that the output data are not available for one or more cycles. Finally, if an output word depends on a later input word,

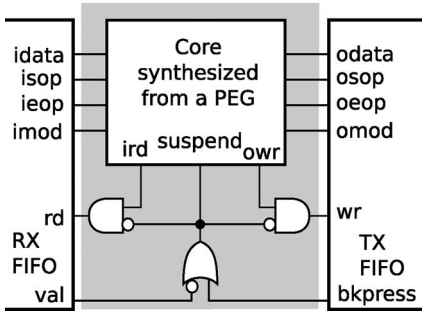

Fig. 5. A module's I/O signals and its two components.

the output must stall until the needed data have arrived at the input. In short, data do not move through the module uniformly; valid data tokens are interleaved with idle ones.

Our technique strives to keep the data flow steady (i.e., to insert as few idle cycles as possible). However, this is not always beneficial since the overall performance depends not only on the number of cycles but also on the clock period. Concentrating complex operations in a single cycle may improve the cycle count but require an unacceptable increase in the clock period.

A. Module I/O Protocol

Within the pipeline, each packet editor module interacts with the adjacent FIFOs through packet input and output interfaces and with memory lookup modules through optional auxiliary inputs and outputs, as shown in Fig. 5 (the gray area represents the module).

The module sees the input packet as a sequence of w-byte words arriving on the *idata* port; $w = 8$ (64 b) is a typical value. Similarly, the *odata* port generates w-byte words. Data

words are accompanied by two framing signals that identify the beginning and the end of the packets. "Start of packet" (*sop*) indicates the first word of the packet; "end of packet" (*eop*) indicates the last. For a packet small enough to fit in a single word, both are asserted. Since a packet may not be an exact multiple of w bytes, the last word may contain between one and w bytes. The $\log_2 w$-bit *mod* signal indicates the number of valid bytes in the last word, i.e., when *eop* is asserted. It is otherwise ignored.

We currently use the same w for the input and output of every module in a pipeline; however, alternatives would be possible, such as a $2w$-wide word in a block that would otherwise only be able to produce a word every cycle. Multiple clock domains are also possible, although on an FPGA they would consume already-scarce clock distribution resources.

The input interface consists of the *idata, isop, ieop*, and *imod* signals plus two handshaking signals: *val* and *rd*. The input FIFO generates the *val* signal when it presents valid input data. The module generates the *rd* signal to request new data from the FIFO in the next cycle. If *rd* is false and *val* is true, the FIFO holds its output. If *val* is false, the FIFO will present the next valid data word as soon as it can, regardless of *rd*.

The output interface consists of the *odata, osop, oeop*, and *omod* signals, plus two handshaking signals: *wr* and *bkpress*. The module generates the *wr* signal when it writes to the output port. When the FIFO is full, it asserts *bkpress* and ignores *wr*.

The optional auxiliary ports use the same handshaking scheme as mentioned earlier, i.e., they use *val, rd, wr*, and *bkpress*. However, these ports may have different bit widths and do not use framing flags (*sop, eop*, and *mod*). The module makes exactly one access to each auxiliary port for each packet, unlike the packet input and output ports, which transfer packets as a sequence of words across multiple cycles. For the auxiliary input, the read request *auxird* is asserted at the start of each packet, thereby requiring the value on this port to remain stable for the current packet. The module asserts the *auxowr* write request when it has computed the data.

B. Core and Wrapper

A module may have to suspend its operation when it tries to request data from an empty input FIFO or when it wants to write to a full output FIFO. Thus, we construct each of our modules to hold its state when a *suspend* signal is asserted and enclose each module in a wrapper consisting of an OR and two AND gates (Fig. 5). Our wrapper stalls the module under either condition. This is conservative since there may be cases where the module can generate data without additional input data; however, these are rare cases since the upstream FIFO is rarely empty (properly sized FIFOs can ensure this). The details are subtle; we describe a corner case in Section V-G.

C. Splitting Data Into Words

The synthesis procedure begins by dividing the input and output packets into words. Dividing the input packet is straightforward. For the (output) packet flow graph, we use the algorithm shown in Fig. 6. Fig. 4(b) shows the result of dividing Fig. 4(a)

```
Restructure(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 8w bits then                                finished word
      n′ ← build-node(v)                            next word node
      n″ ← Restructure(successor of n, (), w)
      make n″ the successor of n′
    else
      n′ ← Restructure(successor of n, v, w)
  conditional :
    n′ = copy of the conditional n
    for each successor s of n do
      n″ = Restructure(s, v, w)
      add n″ as a successor of n′
  if clean-visit then
    cache[n] ← n′
  return n′                                      the restructured node for n
```

Fig. 6. Structuring a packet flow graph into words.

into words. We restructure the packet flow graph such that conditions are only checked at word boundaries. For example, we moved the > 0 condition in Fig. 4(a) four bytes earlier in Fig. 4(b). To preserve the semantics, we made two copies of the intervening 4 B, placing one under each branch of the conditional.

The algorithm shown in Fig. 6 recursively walks the packet flow graph to build a new one whose nodes are all w bytes long (the word size). Each node is visited with a vector ν of bits that are "pending" in the current word. Output nodes are added to this vector until $8w$ bits are accumulated; a new output node n' is created by *build-node* by assembling the pending bits in ν. When the algorithm reaches a conditional node, the algorithm copies the conditional node to a new node n' and visits the two conditional's successors. The same ν is passed to each recursive call because any bits that appeared before the conditional have not yet been written out and need to be written later.

Duplication from hoisting conditionals is potentially exponential but turns out not to be a problem in practice. For example, although there are four paths in Fig. 4(b), they only lead to two final states. We handle reconvergence by maintaining a cache of already-visited nodes that we only consult when ν is empty. In such a case, when visiting a node that has already been visited (i.e., is in the cache), the traversal is stopped, and the node from the previous visit is returned.

D. Assigning Read Cycle Indexes

After splitting the packet flow graph into uniform-length words, we assign to each node in the dataflow graph a read index that indicates how many input words must arrive before the output of the node can be computed—a representation of causality. Input packet nodes are labeled consecutively starting from zero, the optional *auxin* node is also labeled zero since we assume its value is known at the beginning of the packet, and we label the root of the packet flow graph with -1.

We label each remaining node in the dataflow and the packet flow graphs with the maximum index of all of its predecessors, thus guaranteeing that no node will be computed

earlier than its inputs. For example, the "&" node in Fig. 4(c) has two operands, namely, the *flags* field from the auxiliary input and the *flags* from the input packet. They are labeled "0" and "1," respectively; thus, the "&" node is labeled 1. Note that under these rules, two consecutive words in the packet flow graph may have the same label. We resolve such situations in the scheduling step described in the following section.

E. Scheduling

After assigning read cycle indexes, we schedule the graph to assign a clock cycle to each operation. An index may be mapped to more than one cycle.

Our scheduling algorithm inserts "bubbles" (cycle boundaries) according to two constraints. There must be at least n bubbles between an arc connected to nodes labeled k and $k+n$, and there must be at least one bubble between any two nodes in the packet flow graph. The former rule ensures that a node's data are held until it can be used; the latter ensures that the module never attempts to output two words in a cycle.

Fig. 4(c) has been scheduled according to these rules. Black squares on edges represent cycle boundaries. The first rule forced us to insert two bubbles between the top node in the packet flow graph (index -1) and the first output node (index 2); the second had us insert bubbles after the first and third conditionals.

F. Synthesizing the Controller

After assigning read cycle indexes and adding bubbles, generating the module's datapath and controlling finite state machine is easy. The datapath comes from the input nodes and dataflow graph; the packet flow graph dictates the controller.

The structure of the controller follows the structure of the packet flow graph. Fig. 7 shows the algorithmic state machine (ASM) chart generated from the scheduled graph shown in Fig. 4(c). Bubbles in the packet flow graph become states in the controller; conditionals become conditionals. For example, the topmost bubble become the *init* state in Fig. 7; the next one becomes S1, the first conditional is copied, and then, the bubbles on its outgoing arcs become states S2 and S3.

The bubbles before the sinks of the packet flow graph are merged into a common *rep* state that handles the variable-length payload and the end-of-packet condition. We describe this in the next section.

Each normal node in the packet flow graph becomes a statement in the ASM chart that sends data to the *odata* bus. Such nodes also assert *owr* to indicate valid output data. In other states, *owr* is not asserted, and *odata* is a don't-care.

The *ird* signal is asserted on each arc that joins two nodes with different indexes. The first scheduling rule ensures that, at most, one such arc is present between two states. This guarantees that two words are never read from the input in the same cycle. The *ird* signal remains de-asserted for state transitions where the index remains constant, effectively stalling the input FIFO.

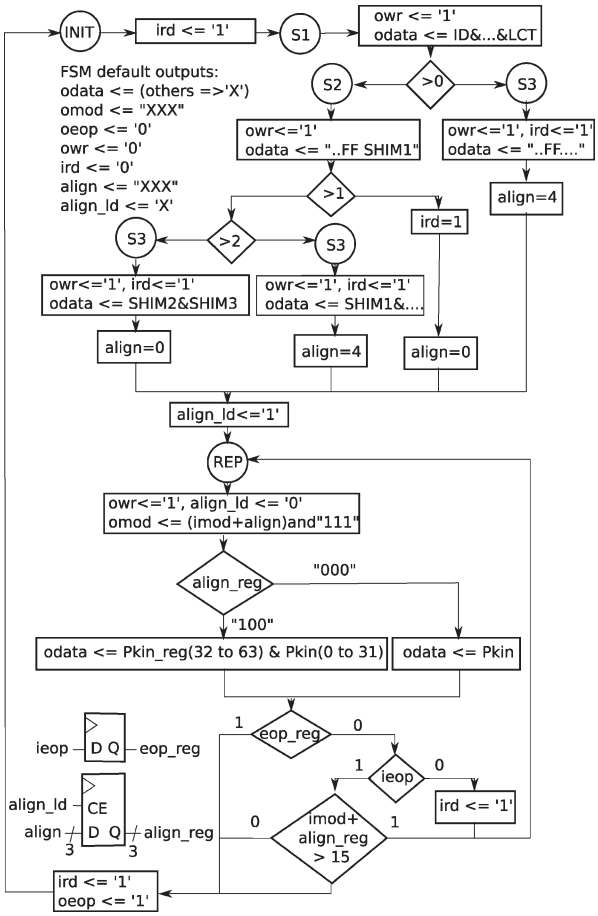

Fig. 7. ASM chart for the controller synthesized from Fig. 4(c).

G. Handling the End of a Packet

The sinks in the packet flow graph make the module directly copy its input (often the packet's payload) to its output. The *rep* state in the ASM chart handles this.

The PEG shown in Fig. 4(c) has two such sinks. The right one runs when zero or two SHIM fields are inserted in the header. Since $w = 8$ in this example, the output alignment is the same as the input. The left leaf corresponds to the case when one or three fields (4 or 12 bytes) are inserted. Here, the input data are misaligned for the output, and the module has to assemble each output word from two consecutive input words.

Rather than generate a separate state for each leaf, our algorithm generates an *align_reg* register and a common state (the *rep* state shown in Fig. 7) that handles all the alignment cases. In the *rep* state, the generated state machine first tests *align_reg* and assembles the output word. When the packet is misaligned, it combines part of the input word from the previous clock cycle with part of the input word from the current clock cycle.

When *ieop* is asserted, the current input word is the last in the packet; thus, the module must output the pending bytes plus the valid bytes from the current word before going to the *init* state. Two cases are possible. If the remaining data can be sent in a single word, the module goes directly to the *init* state; otherwise, a second cycle is needed to finish the transfer.

The *eop_reg* 1-byte register helps to handle the second case. It is set in the cycle *ieop* is first detected and it forces the

controller to the *init* state in the next cycle, preventing the machine from looping forever.

It may appear that a module could stall on the last word of a packet when no new packet is waiting on the upstream FIFO; but this turns out not to be the case. The details of this corner case are worth explaining in part to illustrate the challenge of constructing such circuits by hand. The worrisome scenario is this: The module is processing the last word of the "last" packet (i.e., there is no packet immediately following it in the upstream FIFO) and is thus about to suspend itself; however, the length of the packet has extended into another word so that another clock cycle is needed to produce the final word of the output packet. In this case, the module recognizes that it must continue to output words without consuming its input; hence, it does not assert *ird* in the second-to-last cycle. This prompts the FIFO to continue to assert *val* in the next (final) cycle, thus preventing the module from being stalled. Only when the module knows that it has finished emitting the last word of the output packet does it assert *ird*, which may prompt the FIFO to deassert *val* in the next cycle and suspend itself. The bottom states in Fig. 7 induce this behavior.

H. Synthesizing the Datapath

We directly translate the nodes in the dataflow graph into combinational logic to form the datapath, which also includes the output multiplexer that generates the *odata* output word.

Our algorithm transforms each bubble on an arc in the dataflow graph into parallel registers, one for each bit of the arc. This guarantees that each node has valid data starting in the clock cycle corresponding to its read cycle index.

The "bubble" registers need load-enable signals since a single read cycle index may correspond to multiple clock cycles. We add logic to compute the load-enable signals; however, most of this logic becomes trivial after logic optimization.

Although many datapath registers could be removed during this step, we do not do so because they help in retiming during logic synthesis. Our simple as-soon-as-possible scheduling policy is almost certainly not optimal for balancing computation across multiple clock cycles; however, doing better requires a detailed understanding of low-level logic details. Instead, we let logic synthesis address this problem by generating circuits that are well suited for retiming.

VI. Pipeline Analysis

By design, connecting our modules to form pipelines is straightforward because we synthesize modules that speak matching protocols on their input and output ports. While it would be possible to connect modules directly, doing so would force them to run in lock step and greatly reduce throughput; hence, modules are invariably connected with FIFO buffers. Such buffering smooths out variations in data flow rate among the modules; sufficiently large buffers isolate each adjacent module, allowing the pipeline performance to be improved by improving the performance of each module separately.

In this section, we address the challenge of selecting appropriate FIFO sizes. Making them too small reduces throughput by constraining the behavior of adjacent modules; making them too big needlessly consumes area and power.

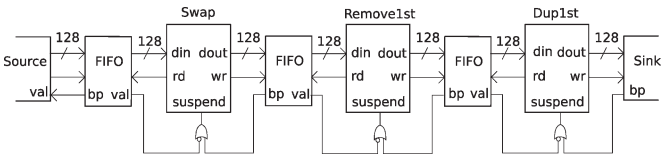

Fig. 8. Packet pipeline. How big should the FIFOs be?

While pipelines in general have been analyzed extensively, ours have unique properties that warrant specialized analysis. Uneven data flow is their key distinguishing feature; modules can insert and remove data based on state and input data. This is what makes them algorithmically rich and difficult to analyze. However, our pipelines are simple in other ways. Throughput, not latency, is the figure of merit because overall latency in a switch tends to be dominated by queuing and forwarding; a few extra clock cycles in the packet-processing pipeline do not matter. Moreover, our pipelines are linear. Finally, the function of each stage in our pipelines is fairly simple because of high performance requirements.

We describe how to compute the performance of a module in isolation, how to combine these results to compute the overall performance of a pipeline with FIFOs that are big enough to isolate the modules, and, finally, how to determine the smallest FIFOs that deliver such isolation. We propose two algorithms for the latter problem: one exact and one heuristic. While both are fairly costly to run (they repeatedly call a model checker), we believe that they are practical because FIFO sizes are typically chosen only once during development, and the answer only affects pipeline performance and not its function.

A. Example

Consider the pipeline shown in Fig. 8, which processes packets that start with one or more 112-bit Ethernet headers followed by a payload. The first module can swap the first two headers, the second can remove the first header, and the third can duplicate the first header. Each module only modifies packets with certain headers and passes others unmodified. This pipeline can modify packets in eight different ways. Although a single module could do all this, our multimodule architecture is representative of real packet-processing pipelines, which eschew complex modules for performance reasons.

These simple module processors have complex data rates. While the first module does not change the packet length, swapping headers means that the output must stall until the second header arrives at the input, then the input must stall while the output writes the first header. The second module stalls the output while the first header is being discarded, and the third stalls the input while it makes a second copy of the first header. Such data- and state-dependent behaviors combined with FIFOs makes pipelines like this complex sequential systems.

We want to answer two questions. First, what is the pipeline's worst-case throughput? We have observed many engineers use a questionable rule of thumb and simply assume that any pipeline can only process data every other clock cycle on average;

however, our experiments suggest this is often wasteful and even optimistic for certain pipelines. Second, for an achievable throughput, how small can the FIFOs be for them to still guarantee that performance? Most designers use intuition, often leading to overprovisioning. Even for simple pipelines, these questions are difficult to answer manually.

Our methods take less than 30 s to tell us that the worst-case throughput of this pipeline is 0.6 and the FIFO sizes of 4, 5, and 5 can guarantee this. Furthermore, if we reduce the throughput requirement to 0.5, FIFOs of sizes 4, 4, and 4 suffice, as do 4, 5, and 3.

B. Analyzing an Isolated Module

We first analyze the performance of a module in isolation and assume that later, we will add enough buffering to the pipeline to make it behave as if it is isolated. Later, we show how to combine these measures to analyze a complete pipeline.

For performance analysis, we abstract the interface of each module to an input and an output bus; two control signals *rd* and *wr*, which indicate when the module wants to communicate; and a *suspend* signal that stalls the module when input data are unavailable or the downstream FIFO is full.

An ideal module always asserts *rd* and *wr*; but, most modules are not ideal. For example, the *Remove1st* module shown in Fig. 8 does not write in cycles where the second header arrives at the input. In general, the sequence of values on *rd* and *wr* depends on the function of the module and the arriving packet.

Let r_i^p, w_i^p, and $i = 0, 1, 2, \ldots$ be the sequences of values observed on the *rd* and *wr* signals for an input data pattern p. In general, a module reads and writes data at different rates; thus, we define separately the worst-case read throughput ρ and the worst-case write throughput ν.

For the input, the worst case arises from sequences r_i^p with the smallest number of ones in a given time

$$\rho = \min_p \left(\lim_{t\to\infty} \frac{1}{t} \sum_{i<t} r_i^p \right). \tag{1}$$

The limit—actually the mean of a Bernoulli sequence—exists because $r_i^p \in \{0, 1\}$ so that $0 \le \sum_{i<t} r_i^p \le t$. In fact, $0 \le \rho \le 1$.

This ρ is useful for guaranteeing that a module can process any input flow at a given rate (assuming sufficient input buffering). For example, if $\rho = 0.4$, the pipeline must be clocked at 500 MHz to guarantee a 200-MS/s input throughput for any data pattern since $500 \times 0.4 = 200$. Similarly

$$\nu = \min_p \left(\lim_{t\to\infty} \frac{1}{t} \sum_{i<t} w_i^p \right). \tag{2}$$

Computing ρ and ν from (1) and (2) is impractical because the limits are over sequences of unbounded length. Instead, we analyze an abstraction of a module's behavior—the state transition graph (STG) of the controller—and compute the lower bounds by considering all simple cycles in the graph.

An STG (Fig. 9) is a nondeterministic finite state machine (V, E, r, w), where V is the set of states, $E \subseteq V \times V$ is the set of transitions, and $r, w : E \to \{0, 1\}$ represents the values of the *rd* and *wr* signals on each transition. While each module is deterministic, the STG abstracts data computations, making it nondeterministic. In Fig. 9, the reset state is on the left; branching is due to different operations, and the state with the self-loop handles the packet payload (see Section V-F).

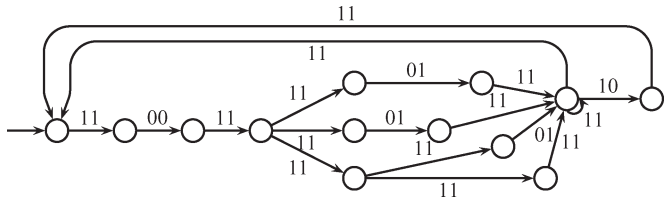

Fig. 9. STG for the DwVLANproc module.

A simple cycle is a sequence of transitions, which forms a nonintersecting path whose tail connects to its head. Considering that our STGs are finite and strongly connected, every path in them follows simple cycles, i.e., if C is the set of all simple cycles

$$S = C \cup C^2 \cup C^3 \cup \cdots = \bigcup_{i=1}^{\infty} C^i \tag{3}$$

contains all finite and infinite paths through the STG, including many that cannot occur on real data because of abstraction.

Each path $s_i \in S$ has the associated read throughput R_i with it. Define a set $\Re$ of all read throughputs that can be achieved by traversing the STG

$$\Re = \left\{ R_i \middle| s_i \in S \wedge R_i = \frac{\sum_{j=1, c_j = s_i(j)}^{|s_i|} \sum_{e \in c_j} r(e)}{\sum_{j=1, c_j = s_i(j)}^{|s_i|} |c_j|} \right\} \tag{4}$$

where $\sum_{j=1, c_j = s_i(j)}^{|s_i|}$ denotes a sum over the elements of a sequence s_i and $|c_j|$ denotes the number of transitions in a simple cycle c_j. The minimum throughput (i.e., the minimum element of set $\Re$) is

$$R = \min_{c \in C} \frac{\sum_{e \in c} r(e)}{|c|}. \tag{5}$$

To see this, consider the well-known inequality $\sum a_i / \sum b_i \ge \min(a_i/b_i)$. It follows that $\forall R_i \in \Re$, $R_i \le R$. Further, since $C \subset S$, it follows that $R \in \Re$.

By a similar argument, the worst-case write throughput is

$$W = \min_{c \in C} \frac{\sum_{e \in c} w(e)}{|c|}. \tag{6}$$

Note that the minimum-weight path in (5) and (6) may be impossible when the module runs with actual data; thus, in reality the worst-case throughput defined by (1) and (2) may be higher. However, this only makes our computations more conservative; we may report the need for larger FIFOs than necessary.

Finally, we define a related metric, namely, the read/write ratio. This is useful for relating the input rate to the output rate,

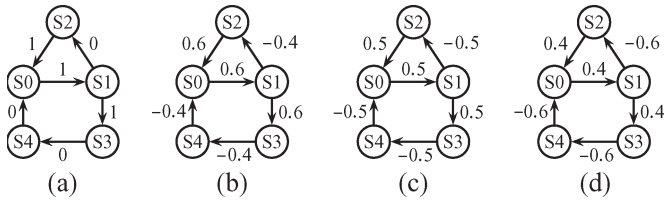

Fig. 10. An STG to illustrate computing R. (a) Edges labeled with $r(e)$. Edges labeled with weights for $\alpha = 0.4$, (b) 0.5, (c), and 0.6 (d). Since $\alpha = 0.5$ is the largest α with no negative-weight cycles, it follows $R = 0.5$.

as discussed later in Section VI-C

$$T = \min_{c \in C} \frac{\sum_{e \in c} r(e)}{\sum_{e \in c} w(e)}. \tag{7}$$

Graph metrics such as (5)–(7) are known as minimum-cycle means [13]. Our formulation matches that of Dasdan [14]; however, we use a slightly different notation.

Dasdan assigns each edge two weights $\omega(e)$ and $\tau(e)$, which we derive from our $r(e)$ and $w(e)$. To compute R, $\omega(e) = r(e)$, and $\tau(e) = 1$; for W, $\omega(e) = w(e)$, and $\tau(e) = 1$; and for T, $\omega(e) = r(e)$, and $\tau(e) = w(e)$.

We compute R, W, and T using a Bellman–Ford-based algorithm proposed by Lawner (see Dasdan [14]). We chose it for its simplicity.

Let $G = (V, E)$ be a directed graph with edge weights w_e for $e \in E$. The $O(VE)$ Bellman–Ford algorithm can determine whether there is any negative-weight cycle in G. By cleverly choosing the weights and performing a binary search, we can quickly estimate R, W, and T.

Consider computing R. From (5), we have

$$R = \max(\alpha) \quad \text{s.t.} \quad \forall c \in C,\ \alpha \leq \frac{\sum_{e \in c} r(e)}{|c|}. \tag{8}$$

Assigning $w_e = r(e) - \alpha$, it follows that

$$\frac{\sum_{e \in c} r(e)}{|c|} \geq \alpha \Leftrightarrow \sum_{e \in c} r(e) - |c| \cdot \alpha \geq 0 \Leftrightarrow \sum_{e \in c} w_e \geq 0. \tag{9}$$

Therefore, we have to find the maximum α such that all cycles are positive. Bellman–Ford can verify the absence of negative cycles for a given α; thus, α can be approximated with a binary search to any desired accuracy.

Similarly, to compute T, let $w_e = r(e) - \alpha \cdot w(e)$ and note

$$\frac{\sum_{e \in c} r(e)}{\sum_{e \in c} w(e)} \geq \alpha \leftrightarrow \sum_{e \in c} r(e) - \alpha \cdot \sum_{e \in c} w(e) \geq 0 \leftrightarrow \sum_{e \in c} w_e \geq 0. \tag{10}$$

Fig. 10 shows the computation of R for a small STG. Fig. 10(a) shows the STG with edges labeled with $r(e)$. This graph has only two simple cycles, namely, (S0, S1, S2) and (S0, S1, S3, S4). Their read/time ratios $\sum_{e \in c} r(e)/|c|$ are $2/3 \approx 0.666$ and $2/4 = 0.5$. Thus, from (5), $R = 0.5$.

For more complex graphs, we use Bellman–Ford as described earlier. Fig. 10(b) shows the edge weights for $\alpha = 0.4$. Since both cycles have a positive weight, we conclude that $R > 0.4$. Fig. 10(d) shows the weights for $\alpha = 0.6$. Here, the small cycle has a positive weight but the large one has a negative weight $(0.4 + 0.4 - 0.6 - 0.6 = -0.4)$; thus, we conclude that $R < 0.6$. Fig. 10(c) has a positive- and a zero-weight cycle, confirming that $R = 0.5$.

$R \leftarrow 1$
for $i \leftarrow n \ldots 1$ **do** $R \leftarrow \min(R_i, R \cdot T_i)$
return R

Fig. 11. Throughput (R) under optimally sized FIFOs.

C. Analyzing a Pipeline of Isolated Modules

It is easy to compute the throughput of a pipeline by combining the values of R, W, and T computed for isolated modules. The analysis remains valid if we assume that the modules are interconnected using finite-size FIFOs, provided that they are large enough to avoid causing a performance bottleneck. Further, we show that such a set of FIFOs exists for any set of pipeline modules. First, note that packet lengths are bounded in all practical network protocols. By construction, no module will start reading the next packet until it has finished writing the previous packet; so FIFOs large enough to accommodate two packets (one being read, one being written) never block a module because of insufficient capacity. Such large FIFOs are wasteful and unnecessary in practice; we discuss how to compute the exact bound in Section VI-D.

Consider computing the throughput of the pipeline shown in Fig. 8. Let R_{123} be the overall throughput (because the pipeline consists of M_1, M_2, and M_3). First, compute R and T for each of the modules, following Section VI-B. Now, consider the "module" M_{23} obtained by merging M_2, M_3, and their connecting FIFO. We assume that the FIFO never limits the throughput; hence, either M_2 or M_3 is the bottleneck. If M_2 writes slower than M_3 can read, M_2 is the bottleneck, M_3 is effectively an ideal sink, and $R_{23} = R_2$. If, however, M_3 reads slower than M_2 writes, the FIFO will eventually fill up. The fraction of time M_2 is not stalled is exactly the ratio between the read of M_3 and the write rate of M_2. T_2 is the ratio of the read rate of M_2 to its write rate; hence, the overall throughput of the pipeline is $R_{23} = T_2 R_3$. Overall, then, $R_{23} = \min(R_2, T_2 R_3)$. Following the same reasoning, $R_{123} = \min(R_1, T_1 R_{23})$. In general, this process can be applied from output to input—Fig. 11—to compute the throughput of the whole pipeline.

D. Analyzing a Pipeline of Interacting Modules

In this section, we use a model-checking technique to establish how small the FIFOs can be and still guarantee the performance we computed in the previous section. Small FIFOs force upstream modules to wait on stalled downstream modules even if the downstream modules have sufficient average throughput. The problem can be avoided by using very large FIFOs; the objective is to find their minimum sizes.

The problem is that the minimum FIFO size depends on the cycle-by-cycle behavior of the modules and not just on aggregate measures, such as R. For example, a single-element buffer is sufficient to connect a module that (at worst) writes

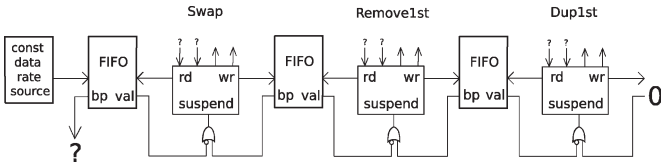

Fig. 12. Pipeline model suitable for model checking.

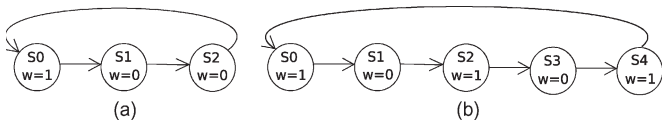

Fig. 13. Data source state machine for (a) $T = 1/3$ and (b) $T = 2/5$.

every other cycle to one that reads every other cycle. On the other hand, connecting a module that stalls for ten cycles then writes for ten cycles would demand a larger FIFO, although its average throughput is the same. Such interactions grow complicated quickly when they involve multiple modules.

For these reasons, we resort to performing model checking to analyze the behavior of our pipelines. To make it practical, we abstract away all data behavior by only analyzing STGs for the modules and the FIFOs. This makes it possible to analyze small pipelines exactly in reasonable time; however, an exact approach does not scale so that we also propose a heuristic that runs much faster and comes reasonably close to the ideal.

Broadly, our algorithms work by building a model of the pipeline that can answer whether the pipeline operates smoothly at a particular throughput by feeding it a specific pattern of valid and invalid words, and then asking a model checker whether the model ever asserts the "backpressure" signal. Fig. 12 shows such a model. We connect the output of the pipeline to an ideal sink that is always able to accept another word. We could also only allow a sink with 50% throughput, for example, by connecting a different module at the sink.

We connect the input of the pipeline to a constant-rate data source—a simple STG such as those shown in Fig. 13—that supplies a repeating pattern of valid and invalid data words to the pipeline. For example, to test 33% throughput, the driving STG writes for one cycle and stalls for two. This approach only allows us to consider rational rates.

Overall, model checking is very different from simulation which only tests a specific series of data *values*. Instead, our approach tests *all possible data values* but constrains the throughput by supplying a fixed pattern of valid words.

Fig. 14 shows a representative STG for a three-element FIFO. For efficiency, our FIFOs are Moore machines; their inputs can only affect their outputs after a cycle.

E. FIFO Size Monotonicity

Since we use costly model checking in the inner loop of the search algorithm to determine whether a pipeline configuration can achieve a given throughput, we make the following observation to reduce the search space.

For two pipelines with the same modules M_i but different FIFO sizes f_i and f_i',

$$\forall i,\ f_i < f_i' \quad \text{implies} \quad R \le R' \tag{11}$$

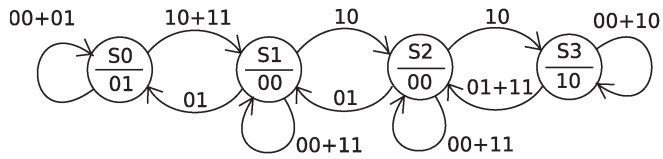

Fig. 14. STG of a three-place FIFO. Inputs: *wr, rd*. Outputs: *bp, val*.

```
GeThanAny(F, list)
  for F' ∈ list do
    if F ⪰ F' then
      return true
  return false

LeThanAny(F, list)
  for F' ∈ list do
    if F ⪯ F' then
      return true
  return false

MinSize(list)
  s ← ∞
  for F' ∈ list do
    s' ← Σ_i F_i
    if s' < s then
      F ← F'
      s ← s'

Succ(F)
  S ← ∅
  for 0 ≤ i < n do
    S ← S ∪ (F_0, ..., F_{i-1},
             F_i − 1, F_{i+1}, ...)
  return S

SearchMinFIFO
  s ← 1
  F ← (1, 1, ..., 1)
  repeat
    BAD.push_front(F)
    s ← s + 1
    F ← (s, s, ..., s)
  until ThroughputAchieved(F)
  GOOD.push_front(F)
  for F' ∈ Succ(F) do
    TRY.push_front(F')
  while TRY.size > 0 do
    F ← TRY.pop_front
    if GeThanAny(F, GOOD) then
      ok ← true
    else if LeThanAny(F, BAD) then
      ok ← false
    else
      ok ← ThroughputAchieved(F)
    if ok then
      GOOD.push_front(F)
      for F' ∈ Succ(F) do
        TRY.push_front(F')
    else
      BAD.push_front(F)
  return MinSize(GOOD)
```

Fig. 15. Exact algorithm for computing minimum FIFOs.

because increasing the size of a FIFO can never decrease the throughput; decreasing the throughput requires more backpressure; however, a larger FIFO never induces any.

This is not a total ordering, e.g., it does not discriminate when $f_i < f_i'$ and $f_j > f_j'$ for some $i \neq j$. Nevertheless, it helps reduce the search space when trying to find overall minimum FIFO sizes. When $\forall i, f_i < f_i'$, we write $F \prec F'$.

F. Exact Depth-First Search Algorithm

Fig. 15 shows our exact algorithm for determining minimum FIFO sizes. It is a variant of a depth-first search that uses (11) to prune the search space (the GeThanAny and LeThanAny functions). It is slow—later, we present a heuristic variant—but it can be practical. We also used it to evaluate our heuristics.

The core function is ThroughputAchieved (not shown), which calls the VIS model checker [15] to determine if the pipeline with a given set of FIFO sizes can achieve the desired throughput. The algorithm first considers pipelines with all FIFOs of size 1, then all of size 2, etc., until a feasible one is found; this is the starting place for the search.

We maintain three lists of FIFO size assignments: *good, bad*, and *try*, which contain assignments that have worked, failed, and not been checked, respectively. The Succ function returns the next points in the search space if the current state works. The depth-first behavior arises by adding and removing elements from the beginning of the *try* list in a stack-like fashion.

```
GreedyMinFIFO
  s ← 1
  repeat
    s ← s + 1
    F ← (s, s, ..., s)
  until ThroughputAchieved(F)
  H ← (0, 0, ..., 0)
  while H ≠ (1, 1, ..., 1) do
    m ← −1
    for i ← 1, ..., n do
      if H_i = 0 and (m = −1 or F_i > F_m) then m ← i
    F_m ← F_m − 1
    if not ThroughputAchieved(F) then
      H_m ← 1
      F_m ← F_m + 1
  return F
```

Fig. 16. Greedy search for minimum size FIFOs.

The MinSize function returns the best solution found; our cost metric is simply the sum of FIFO sizes, reflecting their area. Other metrics would be easy to accommodate.

G. Heuristic Search Algorithm

The exact algorithm we presented earlier is often too slow to be useful; so we modified it to use a heuristic: Fig. 16. Like the exact algorithm, it begins by increasing FIFO sizes uniformly until it finds a solution, and then, it decreases the largest FIFO until doing so would prevent the pipeline from operating at the desired throughput.

Once the size of a particular FIFO has been decreased to the point that it violates the throughput constraint, we mark the FIFO as "held" (the members of the H array) and do not attempt to further modify its size. The algorithm terminates when every FIFO has been marked.

This algorithm can miss the optimal FIFO sizes because it assumes that the FIFO sizes are independent, which is not always true. Relations between FIFO sizes can be complex. For example, increasing the size of one may allow two or more FIFOs to shrink (this does not violate the monotonicity result). Nevertheless, we find our heuristic works well in practice.

VII. Experimental Results

Using PEG, we implemented modules from the GPON OLT system (Section II), as well as a subset of modules from another (experimental) system. We synthesized them using the techniques of Section V, assembled them into a variety of pipelines, and used the techniques of Section VI to determine appropriate FIFO sizes. The data bus width was 128 b; we targeted a Xilinx Virtex 4 XC4VLX40-FF668-10 FPGA.

Table I shows the size and performance of synthesized modules. The top half of the table shows the modules from the experimental system; the bottom half shows the modules from the GPON packet-processing pipeline described in Section II. The row labeled "Full Pipeline" is the result for a fully assembled GPON pipeline, including the interconnecting FIFO buffers.

We measured the performance of each module using the Xilinx post-routing static timing analyzer. Each delay is of the longest register-to-register path plus register setup time. For area, we report the number of lookup-table primitives and of flip-flops used in the Virtex 4 device for the module minus its wrapper. The rightmost column shows the estimated module throughput, which is the product of the module bit width and the frequency in the previous column. This is an overapproximation that does not consider for what fraction of cycles the module is actually producing or consuming data; however, it is still a reasonable indicator of performance. For both case studies, synthesized modules are consistently showing the throughput between 25 and 40 Gb/s. We observed no performance penalty between modules placed and routed in isolation and those in a pipeline.

TABLE I
PEG Module Synthesis Results

Module	Core Size		Delay	Throughput
	LUTs	FFs		
MPLSpush	556	107	3.8 ns	33 Gbps
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
IngrsDrop	287	406	3.7 ns	34 Gbps
IngrsVLANproc	767	378	4.9	26
AddrLearn	80	154	3.7	34
AddrLookup	48	140	3.7	34
EgrsDrop	283	401	3.9	32
PortIDedit	29	138	3.8	33
EgrsVLANproc	489	475	3.9	32
Full Pipeline	4854	4603	5.0 ns	25 Gbps

In the production system, the GPON pipeline runs at 166 MHz—a 6-ns period. All synthesized modules are capable of running considerably faster and, thus, were not the bottleneck. Instead, the external packet memory interface was the main system bottleneck; at no time during the development process did we see a timing violation in the pipeline modules.

Wide buses let us achieve the required throughput—an obvious technique for overcoming FPGA clock frequency limitations. However, it is exactly these kind of designs that make the hand-coding at the register transfer level lengthy, tedious, and error prone. The results demonstrate the merit of our proposed techniques; the throughput requirement of the system was easily met (and, in some cases, exceeded), while the design time and code maintainability were vastly improved.

To try to quantify "maintainability," we surveyed the code repository of the GPON design and focused on modules written by a single developer. Twenty-two modules were handwritten; nine were synthesized using our tools. The handwritten modules had a total of 131 check-ins (5.95 per module, on average), 41 of which (31%) were bug fixes. The nine automatically synthesized modules had 35 check-ins (3.88, on average), five of which (14%) were bug fixes. While these results are limited (e.g., we did not have time to do independent hand implementation of any modules), they suggest that our technique improves productivity.

To evaluate the FIFO sizing algorithms, we arbitrarily selected four modules from Table I, synthesized them, and ran our algorithms on various combinations. Table II shows throughputs for a 32-bit bus. For some modules, the reported throughput may appear too conservative, e.g., the VLANedit module does not enforce the minimum payload size and the low throughput results from analyzing short packets that do not appear in practice. A more detailed model that included

TABLE II
STATISTICS FOR MODULES IN FIFO SIZE EXPERIMENTS

Id	Name	States	Transitions	R	T
A	VLANedit	24	33	0.600	0.643
B	IngrsVLANproc	18	40	0.530	0.563
C	EgrsVLANproc	13	17	0.909	1.000
D	VLANfilter	1	2	1.000	1.000

TABLE III
EXPERIMENTAL RESULTS FOR FIFO SIZING

Modules	Throughput		Heuristic Alg.		Exact Alg.	
1 2 3 4 5	Ideal (R)	Used	Size	Time	Size	Time
ABC	0.329	0.250	10	6 s	9	18 s
CBA	0.337	0.333	11	8 s	11	38 s
BCD	0.511	0.500	10	2 s	10	6 s
DCB	0.530	0.500	8	1 s	8	2 s
ABCB	0.191	0.166	16	1 m	13	22 m
ABAB	0.123	0.111	16	5 m	15	68 m
ACCA	0.385	0.333	15	32 s	13	8 m
CBAC	0.329	0.250	11	11 s	11	1 m
BBCB	0.167	0.166	17	7 m	15	88 m
AAAA	0.159	0.142	18	19 m	15	326 m
ABCDA	0.217	0.200	20	3 m	17	150 m
DCBAD	0.337	0.333	13	7 s	13	1 m
AABBC	0.119	0.111	24	409 m		
BBCCD	0.288	0.250	17	1 m		
CCDDA	0.600	0.500	10	1 s	10	1 s
DDAAB	0.219	0.200	13	31 s	13	5 m

padding states would be more accurate; however, our goal was to evaluate our algorithms and not to analyze specific pipelines.

We narrowed the bus to 32 bits to increase the sequential complexity and challenge the FIFO sizing algorithm. This also illustrates our maintainability argument; we were able to regenerate the modules by changing one parameter and recompiling. Manually rewriting a VHDL implementation for a new bus width would be a lengthy painful task.

The pipelines in our experiments (Table III) are "random" combinations of three and five modules. Although their functionality is nonsensical, their complexity is representative.

The "Ideal Throughput" column lists R as computed by the algorithm in Fig. 11. We ran the greedy (Fig. 16) and exact (Fig. 15) algorithms with the slightly smaller throughput than ideal (listed in the "Used Throughput" column) because it must be a ratio of small integers. The last four columns list the total size of all FIFOs in the pipeline, as well as the time needed to obtain this solution. Although the results vary substantially, it appears that shorter pipelines and those containing simple modules have a higher throughput and require smaller FIFOs. Not surprisingly, our algorithm runs faster on such examples.

The results show that module sequence is important. For example, BCD requires ten words of storage in the FIFOs; DCB only requires eight. Thus, the interaction of adjacent modules is a critical factor that cannot be ignored for an accurate analysis.

VIII. RELATED WORK

This paper intersects three areas: domain-specific languages for network applications, hardware synthesis of those applications, and pipeline performance analysis and provisioning.

The architecture of a typical packet processor is some combination of deep pipelining and multithreading [16]. Brebner *et al.* [17] have shown how to automate the synthesis of multithreaded architectures on FPGAs. Their systems are state machines that execute a pool of threads that collectively process a single packet. By contrast, we focus on pipelined architectures, specifically the algorithms for high-level synthesis of modules that implement the pipeline stages. Which architectural style is superior depends on the application, its performance requirements, and so forth. We do not attempt to answer this question; if the pipelined architecture is chosen, our tool spares the designer from much low-level coding.

A. Languages for Networking

Our PEG notation is meant to be generated from a language for networking applications. We do not consider the design of such a language; others have. For example, the Click architecture of Kohler *et al.* [18] allows router algorithms to be described as directed dataflow graphs, much like our pipelines. However, they consider only software and code their modules in C++. While its details are very different from this paper, it validates the pipeline-centric semantics of our PEG.

Kulkarni *et al.*'s Cliff [19] is an embedding of Click in Verilog for FPGAs that uses a library of predesigned components. Their contribution is in defining an interface that allows such components to be assembled and a compiler that does so. However, we feel designing the components is the major challenge in implementing router algorithms in hardware.

Mihal *et al.* [20] use a Click-like formalism to synthesize and program networks of horizontally microcoded processors for packet-processing tasks. Like us, they have the goal of generating customized hardware for packet processing from high-level descriptions; however, their approach requires the designer to supply the structure of the datapath. Moreover, unlike our word-by-word approach to processing packets, they consume an entire packet header in one cycle (their sample uses a 344-bit datapath) and process a few cycles before producing the entire packet header. Although we have not made any experimental comparison of the two techniques, we suspect theirs would require more hardware to achieve the same throughput, although they should be able to support more complex operations.

Like Cliff, Schelle and Grunwald's [21] CUSP assembles prewritten hardware blocks. It improves upon Cliff by allowing multiple modules to run simultaneously and permits them to execute speculatively. Our pipelines appear to provide ten times the throughput, probably because of wider datapaths.

B. High-Level Synthesis

Our technique to generate pipeline modules can be considered high-level synthesis but differs in many ways from the classical approach described by De Micheli [10] and others. For example, scheduling and binding are two key operations in classical high-level synthesis; yet, we deliberately use only basic versions of each because of our model of computation. We use as-soon-as-possible scheduling because we expect detailed scheduling decisions to be made by a retiming procedure during

logic synthesis. Since our computational model is loop-free, retiming our circuits is very effective.

Since most of our operations are small (e.g., not multipliers), we do not consider resource sharing; overhead from additional multiplexers and wiring would be self-defeating. Unlike the arithmetic-intensive applications usually considered in high-level synthesis, our modules mostly shuffle and forward data. However, even for costly operations, we would not consider resource sharing since we want the highest performance; the area of our elements is often dwarfed by intermodule buffering.

Our computational model differentiates us from classical high-level synthesis. Instead of assuming that data are held in memories, we assume that data arrive and depart a word at a time. This leads to the main problem that our algorithm addresses; our PEG specifications are functional so that we must schedule operations into clock cycles based on when data arrive and can be sent. The main challenge here is making sure everything is performed as early as possible. Delaying an operation for a cycle may cause a periodic pipeline stall. For a module that takes just three cycles to operate, such a stall could reduce the throughput of the pipeline by 33%; thus, avoiding stalls is key.

C. Pipeline Performance Analysis

Since performance is the point of pipelines, it is not surprising that many have considered analyzing their performance. It is a difficult problem in general; hence, practical approaches tend to either approximate the solution or simplify the problem by constraining the systems being considered. Considering that we are working with data-dependent pipelines and want to compute reasonably tight worst-case bounds, no existing technique is satisfactory, although many address very similar problems.

The asynchronous logic community has addressed this problem more than any other. There, buffering is termed "slack." Manohar and Martin [22] determine slack elasticity, i.e., when buffering can be added and removed without affecting behavior. This is trivial for linear pipelines such as ours; they consider fan-in and cycles, which complicate things.

Modeling pipelines stochastically is usually easier. Traditional queuing theory is helpful if packets arrive with a distribution such as the Poisson; however, we want to know the worst-case behavior. Similarly, Xie *et al.* [23] analyze large stochastic Petri nets to report average delays, which are not our concern.

Our need to model pipeline elements with choice is unfortunate; many existing techniques do not consider choice to simplify the analysis. For example, although synchronous data flow [11] graphs permit arbitrary topologies, not just the linear pipelines to which we restrict ourselves, they are fairly easy to analyze. For example, Murthy and Bhattacharyya [24] show how input and output buffer spaces can be shared when exact data lifetimes are known. Similarly, Le Boudec and Thiran's [25] network calculus, which provides an effective mathematical framework for analyzing the throughput and FIFO sizing for pipelines of modules with variable delays, requires each module to have constant read and write throughputs.

The latency-insensitive design approach of Carloni *et al.* [26] models systems with marked graphs—a choice-free subset of Petri nets. In this choice-free setting, Lu and Koh [27], [28] attack a buffer-sizing problem much like ours and propose a mixed-integer-linear-programming solution. The work of Casu and Macchiarulo [29] is representative of more floorplan-driven approaches; their buffer sizes are driven by expected wire delays, something we do not consider.

Most in the asynchronous community also prohibit pipelines with choice. Nielsen and Kishinevsky [30] only compute performance. Holgaard and Amon [31] do so as well but enable parametric analysis by adding symbolic element delays. Kim and Beerel [32] go a step further and consider where to insert delays in such pipelines—an operation similar to retiming. Prakash and Martin [33] and Beerel *et al.* [34] consider how much buffering to add to systems for maximum throughput; however, again, they do not consider systems with choice.

Venkataramani *et al.* [35], [36] allow choice but resort to testing; they collect simulation traces to estimate performance. Like most testing-based approaches, theirs is efficient but is only as good as the user-supplied test cases.

Burns and Martin [37] consider systems with choice but rely on an explicit representation of the state space of the (concurrent) system, making it scale poorly. While we have not directly compared their technique with ours, it seems doubtful that their explicit approach would scale as well as VIS.

IX. Conclusion

Establishing a strict formalism for describing packet editing operations (our PEG) enables a hardware synthesis procedure that can be used to create high-performance packet processors. Our procedure synthesizes circuits whose performance is comparable with those in state-of-the-art switches while dramatically raising the level of abstraction above that of standard RTL. Our main contribution is a procedure for scheduling a purely functional specification into a sequential one performed across multiple clock cycles. The direct benefit of our technique is improved designer productivity and code maintainability. Experimental results on modules from actual product-quality designs suggest that our approach is viable.

We also addressed worst-case performance analysis and FIFO sizing for linear pipelines of modules with data-dependent throughput. The performance of such a pipeline depends on both its elements and their interaction. Interaction makes the analysis of a real pipeline, with finite FIFOs, difficult. To answer the problem, we proposed two algorithms, one exact and one heuristic, which use a model-checking algorithm to evaluate the feasibility of candidate solutions.

As presented, our FIFO sizing technique uses a simple cost function, which is the sum of all FIFO sizes. This is a good metric for ASICs, where each FIFO can be custom sized. However, the algorithm is easily modified to use a different cost function; for example, FIFO implementations on FPGAs take advantage of the FPGA built-in primitives, which have fixed sizes; thus, the resource usage does not increase linearly with the FIFO size.

Although the analysis algorithms require substantial running time, we consider them practical since they can be run in parallel with the synthesis of the individual modules.

REFERENCES

[1] *Matlab for Synthesis—Style Guide*, Xilinx Inc., San Jose, CA, Mar. 2008. release 10.1.

[2] D. E. Comer, *Network Systems Design Using Network Processors*. Englewood Cliffs, NJ: Prentice-Hall, 2004, ch. 20/21, pp. 293–330. Agere Version.

[3] B. Klein and J. Garza, "Agere systems-communications optimized PayloadPlus network processor architecture," in *Network Processor Design: Issues and Practices*, vol. 1. San Mateo, CA: Morgan Kaufmann, 2002, pp. 219–233.

[4] *Gigabit-Capable Passive Optical Networks (G-PON): Transmission Convergence Layer Specification*, ITU-T Rec. G.984.3, Feb. 2004.

[5] *Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specification*, IEEE Std. 802.3-2005, May 2005.

[6] *Virtual Bridged Local Area Networks*, IEEE Std. 802.1Q-2005, May 2006.

[7] C. Soviani, "High level synthesis for packet processing pipelines," Ph.D. dissertation, Columbia Univ., New York, Oct. 2007. cUCS–041–07.

[8] L. Liu, X.-F. Li, M. Chen, and R. D. C. Ju, "A throughput-driven task creation and mapping for network processors," in *Proc. HiPEAC*. Ghent, Belgium: Springer-Verlag, Jan. 2007, vol. 4367, pp. 227–241.

[9] E. D. Willink, J. Eker, and J. W. Janneck, "Programming specifications in CAL," in *Proc. OOPSLA Workshop Generative Tech. Context Model-Driven Archit.*, Seattle, WA, Nov. 2002.

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

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[12] T. Li, "MPLS and the evolving Internet architecture," *IEEE Commun. Mag.*, vol. 37, no. 12, pp. 38–41, Dec. 1999.

[13] R. M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, no. 3, pp. 309–311, Sep. 1978.

[14] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 4, pp. 385–418, Oct. 2004.

[15] R. K. Brayton *et al.*, "VIS: A system for verification and synthesis," in *Proc. CAV*. New Brunswick, NJ: Springer-Verlag, Jul. 1996, vol. 1102, pp. 428–432.

[16] N. Weng and T. Wolf, "Pipelining vs. multiprocessors—Choosing the right network processor system topology," in *Proc. Adv. Netw. Commun. Hardw.*, Munich, Germany, Jun. 2004.

[17] G. J. Brebner, P. James-Roxby, E. Keller, and C. Kulkarni, "Hyper-programmable architectures for adaptable networked systems," in *Proc. Int. Conf. ASAP*, Galveston, TX, Sep. 2004, pp. 328–338.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[19] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a domain specific language to a platform FPGA," in *Proc. 41th ACM/IEEE Des. Autom. Conf.*, San Diego, CA, 2004, pp. 924–927.

[20] A. Mihal, S. Weber, and K. Keutzer, "Sub-RISC processors," in *Customizable Embedded Processors*, P. Ienne and R. Leupers, Eds. Amsterdam, The Netherlands: Elsevier, 2006, ch. 13, pp. 303–338.

[21] G. Schelle and D. Grunwald, "CUSP: A modular framework for high speed network applications on FPGAs," in *Proc. FPGA*, Monterey, CA, 2005, pp. 246–257.

[22] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proc. MCP*. Marstrand, Sweden: Springer-Verlag, Jun. 1998, vol. 1422, pp. 272–285.

[23] A. Xie, S. Kim, and P. A. Beerel, "Bounding average time separations of events in stochastic timed Petri nets with choice," in *Proc. Int. Symp. ASYNC*, Barcelona, Spain, Apr. 1999, pp. 94–107.

[24] P. K. Murthy and S. S. Bhattacharyya, "Buffer merging—A powerful technique for reducing memory requirements of synchronous dataflow specifications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 2, pp. 212–237, Apr. 2004.

[25] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, vol. 2050. New York: Springer-Verlag, 2001.

[26] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[27] R. Lu and C.-K. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *Proc. ICCAD*, 2003, pp. 227–231.

[28] R. Lu and C.-K. Koh, "Performance analysis of latency-insensitive systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 3, pp. 469–483, Mar. 2006.

[29] M. R. Casu and L. Macchiarulo, "Throughput-driven floorplanning with wire pipelining," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 5, pp. 663–675, May 2005.

[30] C. D. Nielsen and M. Kishinevsky, "Performance analysis based on timing simulation," in *Proc. 31st ACM/IEEE Des. Autom. Conf.*, San Diego, CA, Jun. 1994, pp. 70–76.

[31] H. Holgaard and T. Amon, "Symbolic timing analysis of asynchronous systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 10, pp. 1093–1104, Oct. 2000.

[32] S. Kim and P. A. Beerel, "Pipeline optimization for asynchronous circuits: Complexity analysis and an efficient optimal algorithm," in *Proc. ICCAD*, San Jose, CA, Nov. 2000, pp. 296–302.

[33] P. Prakash and A. J. Martin, "Slack matching quasi delay-insensitive circuits," in *Proc. Int. Symp. ASYNC*, Grenoble, France, Mar. 2006, p. 195.

[34] P. A. Beerel, A. Lines, M. Davies, and N.-H. Kim, "Slack matching asynchronous designs," in *Proc. Int. Symp. ASYNC*, Grenoble, France, Mar. 2006, p. 184. pp. 11.

[35] G. Venkataramani, M. Budiu, C. Tiberiu, and S. C. Goldstein, "Global critical path: A tool for system-level timing analysis," in *Proc. ACM/IEEE Des. Autom. Conf.*, San Diego, CA, Jun. 2007, pp. 783–786.

[36] G. Venkataramani and S. C. Goldstein, "Leveraging protocol knowledge in slack matching," in *Proc. ICCAD*, San Jose, CA, Nov. 2006, pp. 724–729.

[37] S. M. Burns and A. J. Martin, "Performance analysis and optimization of asynchronous circuits," in *Proc. UC Santa Cruz Conf. Adv. Res. VLSI*, 1991, pp. 71–86.

Cristian Soviani received the B.S. degree in computer science and electrical engineering from Bucharest Polytechnic University, Bucharest, Romania, in 1998 and the Ph.D. degree in computer science from Columbia University, New York, NY, in 2007.

Since 2007, he has been with Synopsys, Inc., Mountain View, CA, where he currently works with a team developing a static timing analysis tool. His research interests include high-level synthesis, programmable logic devices, and optimization algorithms.

Ilija Hadžić (S'96–M'99) received the B.S. degree in electrical engineering from the University of Novi Sad, Novi Sad, Serbia (formerly Yugoslavia), in 1995 and the M.S. and Ph.D. degrees in electrical engineering from the University of Pennsylvania, Philadelphia, in 1996 and 1999, respectively.

While at Penn, his research intersected the fields of dynamically reconfigurable hardware and programmable (active) networks. Since 1999, he has been with Bell Laboratories, Lucent Technologies (now Alcatel-Lucent), Murray Hill, NJ, where he currently works as a Distinguished Member of the Technical Staff. His research and professional interests include data, optical, and converged broadband access networks, all with an emphasis on hardware and system-level software.

Stephen A. Edwards (S'93–M'97–SM'06) received the B.S. degree in electrical engineering from the California Institute of Technology, Pasadena, in 1992, and the M.S. and Ph.D. degrees in electrical engineering from the University of California, Berkeley, in 1994 and 1997, respectively.

He is currently an Associate Professor with the Department of Computer Science, Columbia University, New York, NY, where he has been with since 2001 after a three-year stint with Synopsys, Inc., Mountain View, CA. His research interests include embedded system design, domain-specific languages, and compilers.