On Emulating Hardware/Software Co-designed Control Algorithms for Packet Switches

Dimitris Syrivelis
Centre for Research and Technology, Hellas

Paolo Giaccone
Dept. of Electronics and Telecommunications
Politecnico di Torino

Iordanis Koutsopoulos
Athens University of Economics and Business - CERTH

Marco Pretti
CNR - Complex System Inst.
Politecnico di Torino

Leandros Tassiulas
Dept. ECE
University of Thessaly

ABSTRACT

Hardware accelerators in networking systems for control algorithms offer a promising approach to scale performance. To that end, several research efforts have been devoted to verify a hardware version of complex control algorithms but only for small-scale hardware unit tests. In this paper we propose and evaluate an emulation framework, in which such control algorithm accelerators can be integrated to design a packet switch, able both to forward real traffic and to enable extensive experimental evaluation and demonstration scenarios. As a case study, we have integrated in the proposed framework a Belief-Propagation-driven algorithm accelerator for multicast packet scheduling.

1. INTRODUCTION

Improving the performance of computer networks has always been a major design issue, in order to support the growing traffic demand. During the last years, user-generated contents and video streaming services have stressed the available network capacity. In order to cope with relevant investments needed to upgrade communication links and switching nodes, current practices tend, instead, to optimize the performance of currently available network resources by implementing smarter control algorithms.

Hardware acceleration for networking systems has been primarily considered for line-speed processing, such as packet manipulation (e.g., encryption), routing table lookups, and forwarding operations. In particular, the switching capabilities at each network node are maximized by adopting hardware acceleration in the data plane, especially when high aggregate bandwidth must be supported. The data plane within a switching node is made up of some queueing system, which stores the packets waiting to be processed or to be sent to the output interfaces. Moreover, some non-blocking switching fabric (e.g., a crossbar or a Clos network) is responsible for physically transferring the packets to the output interfaces. Finally, a scheduling algorithm is assigned to select the packets to be transferred across the switching fabric, based on each packet destination. Newly proposed packet schedulers are often shown by simulation to significantly improve forwarding performance but, in practice, the required computation overhead might affect the maximum achievable speed.

Performance in terms of throughput, delays and fairness of the switching node depends on the scheduling decision. Maximizing performance requires the scheduler to solve some optimization problem at line speed, i.e., the scheduling decision must be taken in a very short time, not longer than the packet transmission time at the port interfaces. Just as an example, consider a 10 Gbit/s port fed by common 64 bytes Ethernet packets: in this case, the scheduler decision must be taken in less than 51 ns. It is clear that software solutions for the scheduling problem are not able to cope with complex scheduling control decisions. Thus, designers must face the following dichotomy: optimal control implementation is unfeasible, whereas implementable control policies achieve non-optimal performance.

As a practical example of such a dichotomy, consider the specific data plane architecture of core-level Cisco 12000 router [1]. This router is based on an input-queued (IQ) switch with Virtual Output Queueing (VOQ), i.e., one queue is available for each input-output pair. The VOQ structure avoids the well-known head-of-line blocking problem [7] and, combined with an optimal scheduling algorithm, allows one to maximize throughput. Unfortunately, the optimal scheduling algorithm requires to solve a maximum weight matching problem in bipartite graphs [10], whose computational complexity is $O(N^3)$, where N is the number of ports. The implementation of such an algorithm is clearly unfeasible at line rate, even for moderate N. Therefore, in commercial routers, solutions for the scheduling problem are based on heuristic algorithms that meet the following two criteria: (i) achieve reasonable performance under a wide (though not exhaustive) set of traffic scenarios; (ii) are implementable with the available processing power.

Now, the processing power needed to implement a control algorithm is a non-trivial combination of the hardware capabilities (e.g., logic frequency, parallelism, pipeline) and the software complexity; note that software is very flexible

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254797

to describe control algorithms. Hence, in order to design control scheduling algorithms that achieve the best tradeoff between performance and implementation complexity in a switching node, it is of paramount importance to emulate a detailed model of the data plane, to explore all the design dimensions of hardware accelerators.

In this work we present an emulation framework for the operational verification and performance evaluation of hardware accelerators for the control algorithms of fast packet switches. We assume that the hardware accelerators have been developed following the Hardware Description Language (HDL) development cycle. The framework includes (i) a reference design implemented on the NetFPGA [11] platform to host the hardware accelerators and (ii) a tightly integrated software datapath, which, in addition to forwarding real traffic, implements the specific queueing structure of the considered switching architecture. The software datapath is implemented on Click Modular Router platform [8]. Since our framework is a hybrid between hardware and software, we shall refer to it as a *hardware/software co-design* approach. Furthermore, we have developed a toolchain that generates a synthetic traffic pattern and maps it, with the proper injection times Note that the considered NetFPGA platform is the defacto standard hardware emulator for the development and evaluation of novel datapath accelerators, so that most researchers in the field are familiar with the related development internals. The proposed emulation framework introduces appropriate changes to the NetFPGA development approach, in order to host accelerators for control algorithms.

As a case study, we specifically consider the problem of designing a switching node built around a high-speed multicast-enabled fabric. This scenario is very challenging, since not only the optimal scheduling algorithm is computationally very complex, but even the optimal queueing structure is made up of a very large number of queues [3]. The practical relevance of such a scenario is motivated by emerging applications exploiting multicast traffic, i.e., addressed to a set of destinations nodes, rather than a single node. Note, however, that the proposed emulation framework is not applicable only to the considered scenario, but also to much broader domains.

The paper is organized as follows. In Sec. 2 we describe the components of the emulation framework architecture and present the methodology that should be followed by developers to provide the necessary hooks in their design to integrate with the simulator. In Sec. 3 we present a step-by-step example for the development and deployment on the emulation framework of a hardware accelerator. Finally, in Sec. 4 we evaluate the performance achieved by our emulation framework for the specific packet scheduler proposed in [6] and optimized to schedule multicast packets.

2. FRAMEWORK ARCHITECTURE

Our proposed emulation framework includes four subsystems: (i) the generic hardware emulator unit, which hosts the control algorithm accelerators, (ii) the software datapath, which executes forwarding operations on real traffic, (iii) the traffic generator toolchain, and (iv) the performance measurement module.

In the Sections 2.1-2.4 that follow, we shall consider each of them separately.

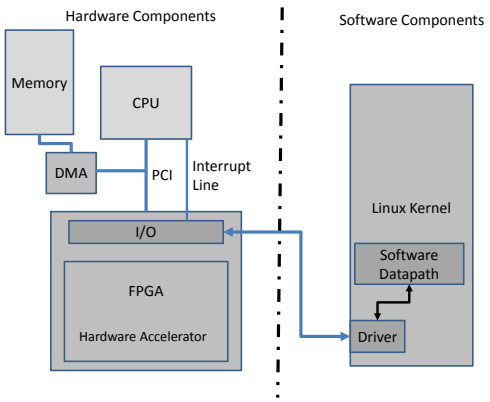

Figure 1: Overview of the emulator hardware, software and gateware components.

2.1 Hardware emulator unit

This unit is implemented on FPGA and provides the reconfigurable resources for the deployment of the accelerator for the control algorithm. Part of these reconfigurable resources is devoted to a communication subsystem, which takes care of the I/O operations with the FPGA software driver. Note that we refer to *gateware* as the code to describe the FPGA configuration.

An overview of the hardware, gateware and software components of the emulator are depicted in Fig. 1. The heart of the hardware emulator unit is the FPGA platform, where the developed gateware version of the control algorithm is deployed. The latter is tightly integrated with a hosting CPU platform via a high speed bus. In addition, Direct Memory Access (DMA) hardware is used by the communication subsystem to speed up the I/O transfer performance between FPGA and main memory. Finally, the communication subsystem gateware drives a hardware interrupt line to the hosting CPU interrupt controller, which can be appropriately wired in the hardware design to alert the software driver when results are ready to be processed in software.

2.1.1 NetFPGA platform

The hardware emulator unit is realized on the NetFPGA 1G [11] platform and, subsequently, the communication I/O subsystem is based on NetFPGA reference design gateware.

The considered NetFPGA features 4 Gigabit Ethernet ports, tightly coupled with a Xilinx Virtex-II-pro FPGA. The reference gateware NetFPGA design performs packet forwarding between the ports and the PCI-bus. The reference processing datapath is pipelined, 64-bit wide, and operates at 125 MHz, equal to the standard Gigabit Ethernet MAC clock frequency; this allows for 8 Gbit/s processing. The FPGA on-chip memory is a BRAM (block RAM); it is a scarce resource (typically a few kB) and can be directly interfaced. Other than that, as it is the case for the CPUs, an external SDRAM controller should be driven by the developed gateware to access data stored on off-chip SDRAM. The overall NetFPGA design approach considerably boosts packet processing operations and fast lookups by exploiting Content Addressable Memory (CAM). Typically, NetFPGA emulators have been developed to facilitate datapath accelerators for novel routing implementations (where many

lookups are required), heavy packet processing operations (e.g., encryption) and networking systems projects that aim at satisfying realtime constraints.

On the PCI bus interface side, apart from network packet I/O, NetFPGA design features a 32-bit register interface. These registers can be either (i) software registers that are written from software and read by hardware, or (ii) hardware registers that are updated by hardware and are read by software. The register I/O interface implements memory mapped access over the PCI bus. On the NetFPGA reference design, the registers interface has been developed to support software-based control algorithms; the registers I/O performance is poor because each register access needs explicit PCI bus address negotiation. Conversely, DMA burst transfer mode permits to move data without the negotiation delay and cannot be exploited for register I/O but only for packet transfer.

2.1.2 *Gateware communication*

The main component of the hardware emulator is the gateware communication subsystem, which abstracts the interconnection details with the software driver for the user-defined accelerator logic. An overview of the internals is depicted in Fig. 2 and it is based on two kinds of interfaces: register-based and packet-based I/O. The corresponding NetFPGA communication subsystems have been extensively altered as follows. For the register-based I/O interface, all of the original NetFPGA reference design components have been removed and the overall reconfigurable logic requirements have been reduced by 90% leaving significantly more reconfigurable resources available to the user logic. More specifically, we have removed the whole logic devoted to the Ethernet interface driver, to the input/output packet queues, and to the arbiter. On the other hand, for the packet-based I/O interface, we have kept only one input and output queue set, thus removing most of the arbiter logic. In this communication scheme, we have developed a custom gateware support to exchange Ethernet packets with the software driver. In these Ethernet packets the registers are sequentially inserted. The developed solution relies on the NetFPGA 64-bit pipelined datapath, where the packets are fragmented in 64-bit data units, which are serially delivered as registers to the user logic and vice versa.

Both interfaces exchange data between registers wired in the user logic design and the software driver. The register interface manipulates 32-bit registers, while the packet I/O interface 64-bit registers. The user also needs to appropriately wire a start signal in the design, which is set by the communication module at high, to trigger execution when all required data transfer to the user logic registers has been completed. Finally, when user logic computation is complete, an interrupt line is available to notify the software driver to collect results from the appropriate registers.

The user may choose between the available communication interfaces, or even use both of them, taking into account the tradeoffs of each approach. More specifically, register-based I/O consumes nearly 70% less reconfigurable resources than packet-based I/O, but it is a considerably slower mechanism because it does not exploit DMA transfers. The maximum number of available registers is relatively small, due to the memory mapped I/O range assigned to the NetFPGA device by Linux. On the other hand, packet I/O needs a significant amount of reconfigurable resources and may not fit

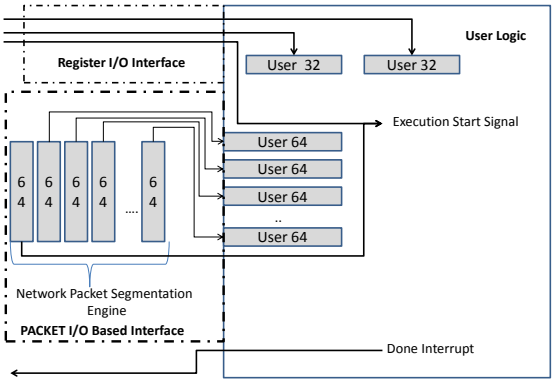

Figure 2: Hardware emulator unit design

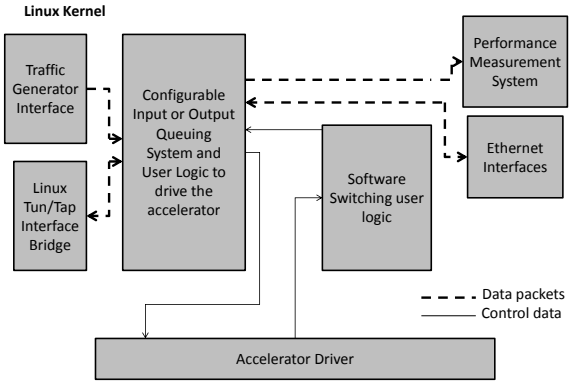

Figure 3: Software datapath design

together with an accelerator design on the NetFPGA platform. In case few bytes need to be exchanged between the hardware accelerator and the software driver for each execution cycle, this approach appears as an overkill, providing no significantly better performance, because DMA-based burst mode transfers are efficient just when larger amounts of bytes must be exchanged.

2.2 Software datapath emulator unit

This unit provides a software platform, which integrates the hardware accelerated control function with a full-fledged packet forwarding datapath. The latter can be configured to emulate specifically the data plane architecture of the switching device in terms of queueing structure and data transfer operations. The internals are depicted in Fig. 3.

The packet entry path can start from three different sources: (i) the traffic generator subsystem that we have developed, (ii) the hardware Ethernet interfaces connected to the network, and (iii) local applications which generate traffic via the hosting Linux kernel network stack. The exit path delivers packets to three different sinks as well: (i) to a user configurable performance measurement subsystem, (ii) to the network via hardware Ethernet interfaces, and (iii) to the local Linux kernel network stack. In Fig. 3 the datapath interconnections are depicted with bold dotted lines, the other

packet sources and sinks are depicted on the right and left sides of the scheme, and the arrow directions indicate entry and exit paths.

The two main components of the datapath are the configurable input and output queueing system and the software switching system. The first component accepts packets at the input and enqueues them depending on the emulated switching behavior. The second component is invoked when forwarding operations are executed. In the case of an input-queued switch, packets are immediately enqueued on arrival, whereas, in the case of an output-queued switch, all user-defined forwarding actions take place on packet arrivals, and the packets are enqueued at the output interface queues for immediate delivery. The hardware-accelerated control algorithm can be invoked at any point in time, according to the user logic in the queuing component. This process can be triggered either by a timer, or on packet arrivals, or by a specific signal from the traffic generator. The user logic is responsible for interpreting the data exchanged with the hardware accelerator. The driver only provides a generic API to push data to the accelerator, and facilitates a callback mechanism where the switching component handler is registered. The interaction of these components is depicted in Fig. 3 with continuous lines.

Typically, the designer will need to add some configuration options to the queueing component, in order to create automatically the input and output queues, according to the number of current inputs and outputs, and according to the desired queueing structure (e.g., VOQ). Moreover, additional code is required for collecting the data to be sent to the accelerator. On the other hand, the switching component requires custom code to receive the accelerator response and perform the required actions.

As mentioned in the Introduction, the software datapath emulation framework has been developed using the Click Modular Router packet processor [8]. In such a system, the developer builds packet processing components, which are integrated into a single binary file. Execution sequence of the components, along with custom configuration options, are all defined in a configuration file loaded during Click bootstrap. The packet processor may also run in the Linux kernel as a module with certain limitations.

In our case, Click-based software datapath communicates with the hardware accelerator via an appropriately modified NetFPGA driver. Indeed, the original version of the latter provides standard network I/O support for the four Ethernet interfaces that are exported by the NetFPGA reference design. Moreover, it provides an interface for register I/O access. We have removed both supports from the driver to bypass the network stack operations and we have just kept the low level functions that implement packet transfers via the PCI bus to and from the NetFPGA. For the packet I/O communication subsystem, we have built appropriate wrapper functions to support the 64-bit Ethernet packet segmentation logic for register exchange with the hardware.

2.3 Traffic generator

The traffic generator subsystem generates traffic arrival patterns on a user-defined number of emulated inputs. These patterns are devised to test the performance of the control algorithm, and are typically determined during a preliminary simulation study. The traffic generator subsystem feeds the emulator datapath and it is comprised of an offline synthetic traffic generator and an online Ethernet packet generator.

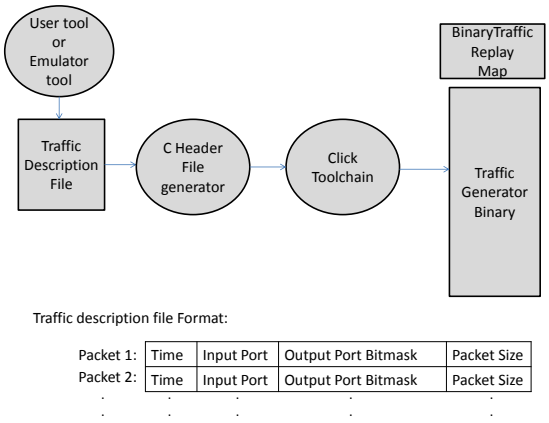

Figure 4: Traffic generator toolchain design

The original traffic pattern files can be generated either by the user or by a synthetic traffic tool, according to the following two processes:

- the time instants at which the packets are generated; this process is either deterministic or stochastic (e.g., Poisson), and is controlled by a single parameter, as the average offered load.
- the set of destinations for each packet (in case of unicast packets, the destination is unique), which is described by a traffic matrix providing the probability that a particular set is chosen.

The traffic generator engine delivers the packets at the inputs of the software datapath component, as depicted in Fig. 3. Moreover, it supports timeslot emulation, where the packet arrivals per timeslot are defined in the traffic pattern file as well. Finally, the traffic generator engine accepts configuration to instruct the datapath to invoke the control algorithm after a specific number of emulated timeslots have elapsed; thus, it is possible to investigate scenarios in which the scheduling decisions are taken with a frequency lower than the packet arrival rate.

The traffic pattern is described through a text file, as shown in Fig. 4. Each packet is described in a single line with tab-separated fields. The first field is the generation time (or timeslot) for the packet; the file is sorted in increasing order according to this field. The second field is the input port where the packet is generated. The third field describes the corresponding set of destinations, represented by a bitmask (of size equal to the number of outputs) with the i-th bit equal to 1 iff the packet is destined to the i-th output port. The fourth field is the packet size (in bytes).

The traffic description file is preprocessed by a custom tool that produces a binary "traffic replay map" as a C header file, as depicted in Fig. 4. This binary file contains an encoded version of each packet arrival, in a few bytes. After the compilation is completed, the traffic generator engine binary is ready to inject the packets.

At runtime, the traffic generator efficiently reproduces each arrival event using bitwise operations. Before being sent to the datapath input, though, each packet gets annotated with its fanout set bitmask and the current time. The

packet annotation space is provided by Click framework and does not affect the packet size and contents.

2.4 Performance measurement

Packets can be sent either to the physical interfaces or to the performance measurement subsystem, in order to evaluate the performance of the control algorithm. This subsystem is a packet sink able to measure throughput and delay, per input and output port. The delay is evaluated as the difference between the current time and the generation time of the packet, which is carried by the packet across the data plane. This subsystem uses the Click Modular Router kernel memory log system to dump the performance results. This design choice is important, since the traffic measurement subsystem does not interact with disk files, and therefore it does not affect performance.

3. EMULATION METHODOLOGY

Let us now describe the development methodology to be followed for the proposed emulation framework. In a few steps, the accelerated control algorithm can be quickly integrated into a real-life networking system. The final deployment can be used for both performance assessment of a final system configuration, as well as for proper operation verification by large scale experiments and direct comparison with simulated results. Note that, for the former case, the final system can be either an all-hardware switching fabric (ASIC) or a hardware/software co-designed solution, in which a part of the control algorithm is designed in software.

3.1 Accelerator integration

The gateware accelerator core may be developed outside the context of the hardware emulator unit framework, following the standard hardware description development cycle, using either the structural approach of Verilog or VHDL languages. Moreover, the developer needs to design a flexible pipeline and/or a state machine that may be revisited to meet the timing constraints on the FPGA deployment target. One important design decision is the encoding and the volume of data to be exchanged between the accelerator and external components, in order to minimize the communication overhead. According to the implemented communication scheme, the accelerator cannot communicate via shared memory with external components. The provided integration signals of the emulator communication I/O subsystem are depicted in Fig. 5.

Initially, the developer defines the 32-bit or 64-bit registers to receive the data over the register I/O interface or the packet I/O interface, respectively. In the same spirit, different register instances need to be defined for data output. The clock and reset signals are supplied by the communication subsystem; thus, the developer needs to remove any other clock sources from the design. The communication subsystem also supplies a start signal line, which changes state whenever a data transfer cycle is completed, so that the accelerator can begin execution. Finally, the accelerator is equipped with an interrupt line, which alerts the communication subsystem when results are ready. In the case of memory-mapped register I/O, the interrupt is delivered to the software driver, which initiates the register read process, while, in case of packet I/O, the interrupt is delivered to the emulator hardware that prepares and sends the outgoing packet. User logic may ignore both the start signal and the interrupt signal (in case of memory mapped registers, for polling-based access), but it cannot ignore the latter when packet I/O based transfer is used. The user edits the configuration options of the communication subsystem, to choose between register and packet I/O, or both. When both options are selected, the interrupt line is connected to the packet I/O module and the registers can be read only via polling. The communication module accepts further configuration to generate the number of bit signals required to accommodate all the user logic input and output registers.

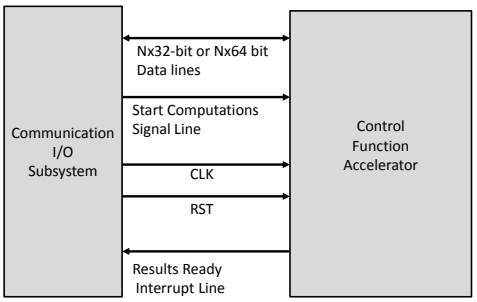

Figure 5: Integrating the gateware accelerator with the communication subsystem

The integrated design gets processed by the Xilinx tools [2] to produce a configuration bitstream for the NetFPGA. As far as the reconfigurable area availability is concerned, the first design approach is to use only the register I/O communication, which requires considerably less area than packet I/O, before the accelerator design is revisited. On the other hand, timing constraints violation definitely requires a development effort. Note that such, possibly required, design improvements are a step forward to the final deployment and are not valid only for the system emulation. After this step is successfully completed, the hardware integration is finished and the developer may proceed with the software-side design.

3.2 Software datapath integration

After the accelerator is installed successfully on the NetFPGA using the default deployment tools, the developer has to configure and add logic to the software datapath framework. At the architectural level, the datapath can be configured as an $N \times M$ switch, where N is the number of input ports and M the number of output ports. Moreover, the inputs and outputs can be configured to be connected to either virtual or physical Ethernet interfaces; alternatively, inputs can be connected to the traffic generator and outputs to the performance measurement subsystem, as depicted in Fig. 3.

The framework module hosting the queues supports the configuration of general-purpose queues, in addition to the typical input and output queueing structures. These configuration options are passed as arguments to an emulator script, which generates the appropriate Click Modular Router configuration files.

3.2.1 Adding user logic

Depending on the overall design approach, the developer needs to add functionality to the software datapath core as follows. If the control algorithm is totally implemented in hardware, the developer needs to add code in the queueing module, to transfer the required queue states in the appropriate format to the hardware accelerator. In most cases, these states are updated online by enqueue and dequeue operations. The developer uses a library interface to update such state variables and operate the enqueue and dequeue functions.

The modified NetFPGA driver API exports generic functions to build and send register values grouped in Ethernet packet(s) for packet I/O, and a slightly different version for the register I/O. Moreover, in the software module that performs switching, we provide a callback, already attached to the accelerator interrupt, which can be used to receive the data from the accelerator using the respective driver API calls.

The developer has to code how to perform the switching operations based on the accelerator feedback. Note that Click Modular Router library features many approaches to switching, which can be properly modified by the developer to achieve the required functionality. The general purpose queues are allocated during the configuration, and the developer is equipped with an array of pointers to the queue objects, which can be properly organized, according to the required queueing architecture (e.g., VOQ).In case the control algorithm is hardware/software co-designed, part of the computation is implemented in software. It is straightforward to add new processing modules to the Click Modular Router configurations, by editing the configuration files produced by the emulator tool before deployment. Finally, the developer has the option of using the supplied traffic generator, described in Sec. 2.3.

4. DEMONSTRATION AND EVALUATION

In this section we evaluate the proposed emulator framework by a design example. We specifically consider the problem of scheduling multicast packets in IQ switches. This problem is very challenging, since it is computationally very complex and must be solved in very short time, as briefly discussed in Sec. 1. Performance is strongly dependent on the scheduling algorithm.

Some previous work has investigated how to solve packet scheduling problems using an FPGA-based accelerator. For example, in [4] the authors investigated the performance improvement achievable by a packet scheduler for unicast traffic, but they did not evaluate the accelerator in an integrated forwarding system. Furthermore, the problem addressed in [4] is much simpler than the current one; indeed, the computational complexity of the optimal scheduling algorithm for unicast traffic is polynomial, whereas our problem for multicast traffic turns out to be NP-hard [3].

In the following, we show how to design an accelerator for the multicast packet scheduler presented in [6], to demonstrate the integration with the emulator framework and provide an evaluation of the presented subsystems. The reason for this choice is that the algorithm in [6] is amenable to efficient FPGA implementation, due to its intrinsic parallel nature.

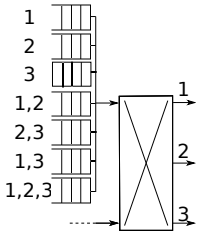

Figure 6: IQ switch with $N = 2$ inputs and $M = 3$ outputs. Each logical queue of MC-VOQ is tagged with the corresponding fanout set.

4.1 Belief propagation for multicast traffic

We consider an IQ switch of size $N \times M$ (see Fig. 6). The switching fabric is non-blocking and bufferless (e.g., a crossbar), able to transfer any non-conflicting set of packets, i.e., at most one packet is transferred from each input and to each output at any time. We assume that the switching fabric natively supports the transfer of multicast packets, i.e., an input can send a copy of the same packet to different outputs, at the same time.

The scheduler, based on the state of occupancy of the queues, has to select a set of non-conflicting packets, in order to maximize throughput. Its task is usually very complex from the computational point of view and the optimal scheduling policy requires to solve an ILP (Integer Linear Programming) problem, which, as previously mentioned, is solvable in polynomial time only under unicast traffic [10]. Here we consider the more challenging problem of scheduling multicast traffic. A *fanout set* is defined as a subset of output ports, so that a multicast packet can be described by the fanout set of its output destinations. For example, a packet destined to outputs 1,2,4 has fanout set $\{1, 2, 4\}$.

We assume that time is slotted and the switching fabric runs synchronously. At each timeslot, packets arrive and are stored in the input queues. Based on the queue occupancy, the scheduler selects the packets to be transferred across the switching fabric.

The adopted queueing architecture is MC-VOQ [3], i.e., one *logical* queue is present for each possible (non-empty) fanout set and each input port. For example, the packets with fanout sets $\{1, 2, 4\}$, $\{1, 4\}$, $\{3, 5\}$, $\{1\}$, $\{1, 2, 5\}$ are stored in 5 different queues. Clearly, each input has to support at most $2^M - 1$ logical queues. As an example, Fig. 6 lists all the possible queues for $M = 3$.

Note that MC-VOQ is the only *optimal queueing* architecture for an IQ switch, because it avoids the head-of-line blocking problem, i.e., a front packet will never prevent another packet in the same queue from accessing a free output.

4.1.1 The optimal scheduling policy

Combined with optimal queueing (MC-VOQ), we consider the throughput-optimal scheduling policy for multicast traffic [3]. Such a policy allows "fanout-splitting", i.e., a packet can be sent to just a subset of its destination ports, leaving some residual destinations for future transmissions. In case of residual destinations, the packet is re-enqueued into the queue corresponding to the residual fanout set. For example, a copy of the packet destined to $\{1, 2, 4\}$ might be sent just to output 1 (if, for instance, outputs 2 and 4 are busy with

packets transmitted from other inputs), while the packet is moved to the queue corresponding to the fanout set $\{2,4\}$.

To be more general, let I and O denote the sets of input and output ports, respectively, whose cardinalities are $|I| = N$ and $|O| = M$. The "power set" $\mathcal{P}(O)$ is the set of all possible fanout sets. As one logical queue is present for each non-empty fanout set, the set of all possible logical queues can be represented by $\mathcal{P}(O) \setminus \{\varnothing\}$, where $\varnothing$ denotes the null fanout set. For example, if $M = 3$, then

$$\mathcal{P}(O) = \{\{1\},\{2\},\{3\},\{1,2\},\{2,3\},\{1,3\},\{1,2,3\}\}$$

A scheduling decision can be represented by N pairs of fanout sets $\sigma_i, \tau_i \in \mathcal{P}(O)$, one for each input $i \in I$. In particular, we define for input i:

1. The *service* fanout set σ_i, representing the served queue;
2. The *transmission* fanout set τ_i, representing the subset of outputs to which the packet is actually transmitted;
3. The *residual* fanout set $\sigma_i \setminus \tau_i$, representing the queue in which the packet is, in case, re-enqueued.

In the above example, the packet at input i, destined to $\sigma_i = \{1,2,4\}$, is sent to outputs $\tau_i = \{1\}$ and then re-enqueued to outputs $\sigma_i \setminus \tau_i = \{2,4\}$.

The decision variables must satisfy the following constraints. First, by construction, each transmission fanout set must be a subset of the corresponding service fanout set:

$$\sigma_i \supseteq \tau_i \qquad \forall i \in I \tag{1}$$

Second, the transmission fanout sets must not generate conflicting packets, i.e, their intersection must be empty:

$$\bigcap_{i \in I} \tau_i = \varnothing \tag{2}$$

According to [3], the objective ("weight") function, to be maximized at each timeslot, is

$$w\left([\sigma_i, \tau_i]_{i \in I}\right) = \sum_{i \in I} [\ell_i(\sigma_i) - \ell_i(\sigma_i \setminus \tau_i)] \tag{3}$$

where $\ell_i(\sigma)$ is the current length of the queue associated to the (non-empty) fanout set σ at input i (whereas $\ell_i(\varnothing) \equiv 0$). The main idea behind this function is *to serve at higher priority packets that are stored in large queues and that are possibly re-enqueued into small ones.* As previously mentioned, this policy was shown in [3] to be optimal in terms of throughput.

The scheduling problem can be simplified, observing that constraint (1) involves *separately* the decision variables σ_i, τ_i of each single input i. As a consequence, one can perform a preliminary optimization procedure, that runs *in parallel* for each input $i \in I$, to determine the maximum value of each single element of the sum in (3), for each possible choice $\tau_i = \tau \in \mathcal{P}(O)$, which we shall call "local weight"

$$w_i(\tau) = \max_{\sigma \in \mathcal{P}(O) | \sigma \supseteq \tau} \{\ell_i(\sigma) - \ell_i(\sigma \setminus \tau)\} \tag{4}$$

The same procedure computes the optimal σ_i, for each possible choice $\tau_i = \tau \in \mathcal{P}(O)$, which we shall denote as $\hat{\sigma}_i(\tau)$. After this preliminary procedure, the original problem is reduced to an optimization over the sole τ_i variables, constrained by (2). The weight function to be maximized is

$$\hat{w}\left([\tau_i]_{i \in I}\right) = \sum_{i \in I} w_i(\tau_i) \tag{5}$$

4.1.2 The belief propagation approach

In order to solve the resulting problem, one can resort to a Belief Propagation (BP) algorithm, as proposed in [6]. BP is a rather general class of algorithms [9], based on message passing among the elements of a given system (input and output ports in our case). Such a distributed nature makes it specially suitable for hardware implementations. As shown in [9], given a proper formulation of a constrained optimization problem, the construction of a BP algorithm is a rather well-established issue, even though some analytical manipulations are usually required to obtain conveniently simple equations. Here, due to limited space, we shall basically give only the pieces of information that are needed for the implementation. More details are reported in [6]. Let us only mention the fact that, even though we generically speak of BP, our algorithm is of the *min-sum* type [9], which can be regarded as a special case of BP, specifically suited for computing MAP (maximum a posteriori probability) estimates.

At a high level description, the algorithm can be divided in the following phases:

1. Message Initialization (MI). Messages are initialized at some "null" value.
2. Message Update (MU). Messages are exchanged between each input and output, and updated iteratively by each input and output, concurrently. At each iteration, each message is computed as a function of the received messages and the state of the queues. This phase ends when either a maximum number of iterations is reached or the message values converge to a fixed point.
3. Scheduling Decision (SD). The scheduler chooses the packets to be transferred, based on the final values of quantities called *beliefs*, that again depend on the messages and on the state of the queues.

The beliefs, associated to each transmission fanout set τ at each input i, are defined as

$$m_i(\tau) = w_i(\tau) - \sum_{j \in \tau} b_{j \to i} \tag{6}$$

where the information about the state of the queues is incorporated in the local weights $w_i(\tau)$, defined by equation (4), whereas $b_{j \to i}$ are "backward" messages (from output j to input i), defined below. The belief $m_i(\tau)$ is an estimate of the weight (objective function value) that can be obtained by choosing a specific transmission fanout set $\tau_i = \tau$. Moreover, the backward message $b_{j \to i}$ is an estimate of the weight degradation due to possible conflicts generated by a transmission from i to j (i.e., by a choice τ_i such that $j \in \tau_i$). Backward messages are defined by self-consistency equations, that depend also on intermediate quantities $f_{i \to j}$, called "forward" messages (from input i to output j):

$$b_{j \to i} = \max_{i' \in I \setminus i} f_{i' \to j} \tag{7}$$

$$f_{i \to j} = \max\left\{0, \max_{\tau \in \mathcal{P}(O) | \tau \ni j} m_i(\tau) + b_{j \to i} - \max_{\tau \in \mathcal{P}(O) | \tau \not\ni j} m_i(\tau)\right\} \tag{8}$$

These equations define one elementary iteration of the MU phase of the algorithm. The beliefs are updated according

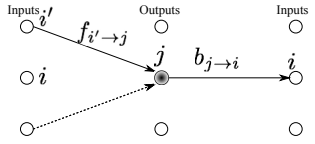

Figure 7: Update of backward message $b_{j\to i}$ during one iteration of MU

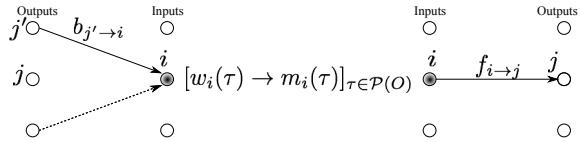

Figure 8: Update of forward message $f_{i\to j}$ during one iteration of MU.

to equation (6). Figs. 7 and 8 provide a graphical representation of the various dependencies expressed by equations (7) and (8), respectively. The backward message $b_{j\to i}$ (from output j to input i) is computed as the maximum among all the forward messages $f_{i'\to j}$, coming from any input i', except i, to the same output j. The forward message $f_{i\to j}$ (from input i to output j) is computed by a more complex expression (including also the beliefs), which however similarly depends on all the backward messages $b_{j'\to i}$, coming from any output j', except j, to the same input i.

A major difficulty of this approach is that, due to the "densely connected" constraints (basically, we have a single constraint (2) involving all variables), there are several cases in which the MU phase does not converge [6]. Because of this problem, a fully distributed SD phase, in which every input i takes its decision τ_i on the basis of its own belief $m_i(\tau)$, is not feasible, since it might lead to conflicting decisions. Such a difficulty can be overcome, using BP with a fixed number of iterations, in conjunction with a centralized decimation algorithm, which at each step fixes a given variable $\tau_i = \tau$ with the maximum belief $m_i(\tau)$ (over all inputs), and then re-runs BP, keeping compatibility with previously fixed variables. The resulting algorithm, which we denote as DEC-BPn (meaning exactly decimation with n BP iterations), is described by the pseudocode reported in Fig. 9.

The algorithm takes as input the array of queue lengths $\ell_i(\sigma)$, for each input i and each fanout set σ, and returns the scheduling decision, in terms of the fanout set variables

```
DEC-BPn (input: [ℓ_i(σ)]_{i∈I,σ∈P(O)}; output: [σ_i, τ_i]_{i∈I})
0.  for i ∈ I and τ ∈ P(O), compute w_i(τ) and σ̂_i(τ) by (4)
1.  set Ĩ = I and Õ = O
2.  while Ĩ ≠ ∅
3.   for i ∈ Ĩ and j ∈ Õ, set b_{j→i} = 0
4.   repeat n times
       for i ∈ Ĩ and j ∈ Õ, compute f_{i→j} by (8) and (6)
       for i ∈ Ĩ and j ∈ Õ, compute b_{j→i} by (7)
5.   for i ∈ Ĩ and τ ∈ P(O) | τ ⊆ Õ, compute m_i(τ) by (6)
6.   choose i ∈ Ĩ and τ ∈ P(O) | τ ⊆ Õ that maximize m_i(τ)
7.   if m_i(τ) = 0, set τ = ∅
8.   set τ_i = τ and σ_i = σ̂_i(τ)
9.   set Ĩ = Ĩ\i and Õ = Õ\τ_i
```

Figure 9: DEC-BPn algorithm.

σ_i, τ_i for each input i. Step 0 performs the preliminary optimization procedure defined by equation (4). The "lists" $\tilde{I}$ and $\tilde{O}$ of "unreserved" inputs and outputs, respectively, are initialized at step 1, assuming that all the ports are initially available. Step 2 begins the decimation loop, which continues until every input has taken a decision, i.e., as far as $\tilde{I}$ is not empty. Steps 3 and 4 represent the MI and MU phases, respectively; step 5 computes the final belief values. Step 6 chooses an input i and a transmission fanout set τ, such that i is available and τ contains only available outputs, maximizing the belief $m_i(\tau)$. Step 7 states that, if the maximum belief found is zero, the algorithm assigns a null transmission fanout set (which corresponds to a vanishing belief as well). The transmission fanout set at input i and the corresponding optimal queue to be served are fixed at step 8. Step 9 updates the lists of available inputs and outputs. Finally, it is understood that, when the decimation loop is over, the current values of the fanout set variables are returned.

4.2 Implementing DEC-BPn

Initially, DEC-BPn has been implemented in Click Modular Router as an integrated system, capable of forwarding real traffic in a 4×4 switch. For simplicity, we have assumed fixed-size packets and slotted time, with the timeslot duration equal to the packet transmission time. A simple profiling on such a software version has revealed that the scheduler execution occupies 94% of each timeslot, while the actual packet switching process and the queue manipulation operations account for 6%. This result was expected, because all the packet enqueue/dequeue operations rely only on pointer arithmetic operations, which take place very efficiently on instruction-set processor architectures. Since the scheduler needs to be invoked as often as possible, a hardware accelerated version has been considered to improve its execution performance. Therefore, we have developed a gateware version of the scheduler, making use of the Verilog hardware description language. The gateware DEC-BPn design has been realized as a state machine, still for a 4×4 switch.

4.2.1 The scheduler communication interface

The DEC-BPn scheduler exports 61 16-bit registers at its input. The first 60 registers are used for passing all the MC-VOQ queue lengths $[\ell_i(\sigma)]$, for any input $i \in I$ (with $|I| = 4$ inputs) and any queue $\sigma \in \mathcal{P}(O)$, (with $|\mathcal{P}(O)| = 15$ queues per input). The last register acts as a control register and it is used to initiate calculation and to indicate when the decision is ready. The scheduler outputs 8 4-bit registers with all the decision variables (service and transmission fanout sets): σ_i and τ_i, for $i = 1, 2, 3, 4$.

The decisions are encoded in each register with a bitmask, designed to index queues (and their corresponding fanout set), and to perform easily queue set operations. Each bit position is reserved for a given port; the value 1 indicates that the respective port belongs to the set represented by the current bitmask. For example, fanout set $\{1, 2, 4\}$ is represented by 1101. As a result, a few bitwise operations can determine whether a given port belongs to a fanout set or not, and queue head pointers are directly indexed by the respective bitmask values, which allow for instant retrieval.

The external logic is responsible for placing incoming packets on the appropriate MC-VOQ structures and log all queue backlogs. The backlog values are placed on the input regis-

ters of the scheduler accordingly. Then the control register is set at 0x1 to initiate execution. As soon as the result is ready, the control register value changes to 0x2. It is expected that the external logic hooks an interrupt line to the respective register bit to get notified or just poll for the result. The result can be read from the output registers and appropriate forwarding operations as well as MC-VOQ reenqueuing operations have to be performed by the datapath logic.

4.2.2 The gateware scheduler state machine

The C software library of the scheduling algorithm, available from the simulator adopted in [6], has been properly restructured to be easily mapped to gateware. During this process, all the for-loops have been transformed as follows. (i) The for-loops that performed independent operations on different regions of data have been "unrolled", so that hardware may execute all operations on a single cycle. (ii) The for-loops that used the feedback from the previous cycle for the calculations during the next one have been converted to state machines.

In this way, the gateware scheduler design has required 81 states to compute all the local weights according to equation (4), in parallel among the 4 input ports. Subsequently, it has required 68 states to compute the forward messages and 53 states to compute the backward messages, during each DEC-BPn iteration. Finally, 71 states are needed to perform all the necessary matching and comparison operations to reach the final decision. In total, for 3 hard-coded iterations, the gateware version of DEC-BP3 needs 515 cycles to produce the final decision. The combinatorial logic within each state has been carefully placed to minimize the critical path delay, so that the overall design can operate at high clock rates. For example, this design can be clocked at 72.9 MHz on a mid-range reconfigurable hardware platform, which enables the DEC-BP2 scheduler to complete execution in 7.06 μs, almost 3 times faster than the software version running on a Intel Core i7 CPU running at 3.06 GHz.

4.2.3 Integration with the emulator framework

The behavior of DEC-BPn gateware accelerator has been verified with ModelSim system for proper operation using some test traffic pattern. In order to integrate with the emulator, we firstly interfaced DEC-BPn gateware with the emulator communication subsystem signals. Initially we aimed to use the packet I/O interface. We appropriately grouped the input and output registers in 64-bit groups to interconnect with the packet datapath signals. Next we have appropriately connected the control register bits with the "start" and "done" interrupt signals, as well as the clock and reset signals. The Xilinx tools were used to compile the design for the NetFPGA target. Unfortunately the final design had reconfigurable resource requirements that exceeded the NetFPGA capacity. We revisited the DEC-BPn design to save some space but it was not adequate, so we decided to use the register I/O interface instead to decrease considerably the required reconfigurable area. Towards that end, we had simply to re-group the I/O registers in 32-bit groups and rewire them with the reconfigured communication module. Another change was to wire the control register to a communication bus register to get the start signal via the software driver, while the "done" signal was removed from the control register and connected to the provided interrupt line. This time the DEC-BPn configuration bitstream was successfully built for the NetFPGA.

Table 1: Communication I/O performance analysis and area requirements on NetFPGA 1G and i7 platform

Communication I/O	Average delay per byte	FPGA area
Packet I/O	9 ns	6265 slices
Register I/O	180 ns	2580 slices

Regarding the software side of the emulator, we started by configuring the Click-based datapath. In our case we have a 4×4 IQ switch, with the inputs connected to a traffic generator instance and the outputs connected to the performance measurement module. Moreover, the queueing module is configured to build the required number of queues to implement the MC-VOQ structure at the inputs. We passed all these arguments to the emulator configuration script that generated appropriate Click configuration files for the datapath. In the sequel, we added logic to the queueing module, the switching module and the traffic generator.

In the queueing module we implemented the MC-VOQ structure and we mapped pointers to the backlog counters which were used for the transfer to the accelerator via the driver API. Each time a transfer cycle is complete, we use the driver API once more to write a register that initiates the scheduler execution. The scheduler response calls a handler in the switching module which has been developed to read the decisions via the driver API, to re-enqueue the packets within the MC-VOQ structure and to forward them to the output performance module sink. The traffic patterns used in the original simulator have been converted to an emulator traffic description file. In our case, each packet is timestamped with its generation timeslot and tagged with its fanout set. The emulator toolchain is used to build the binary traffic map and integrate it with the traffic generator code. After this step, the software datapath is complete and the integration with the emulator has finished.

4.3 Emulator framework evaluation

The proposed emulator framework has been deployed as follows. The gateware communication subsystem has been compiled with Xilinx tools version 10.1 and deployed on NetFPGA 1G PCI platform. The host platform features a 3.06 GHz i7 processor running Fedora 14 with Linux kernel version 2.6.35 32-bit. We used the latest version (2.0.1) of Click Modular Router framework for the datapath implementation. Any co-designed networking system, that is integrated in the emulator framework and uses the provided communication subsystem, shows the fixed I/O performance and reconfigurable area requirements of Table 1.

As expected, the register I/O interface is significantly slower (20 times) because of the memory mapped I/O access that does not allow burst mode memory transfers. Moreover, register I/O interface occupies CPU cycles for each register transfer, while packet I/O uses DMA and the CPU is releaved to perform other tasks during a transfer. Note that the averages presented in Table 1 have been calculated on 256-byte data transfers and the time was measured using Linux "gettimeofday" virtual system call implementation that uses the i7 CPU cycle (TSC) counter (introduced in

Table 2: Gateware 4×4 DEC-BPn execution performance analysis on NetFPGA 1G hosted on Intel i7 CPU

	$n = 0$	$n = 1$	$n = 2$
Push data to input registers	$23\mu s$	$23\mu s$	$23\mu s$
DECBPn execution	$3.77\mu s$	$5.44\mu s$	$7.12\mu s$
Get data from output registers	$13\mu s$	$13\mu s$	$13\mu s$
Datapath execution	$1\mu s$	$1\mu s$	$1\mu s$
Timeslot duration	$40.77\mu s$	$42.44\mu s$	$44.12\mu s$

Linux 2.6.35). There is a fixed setup delay for each packet or register transfer that makes few bytes I/O operations very expensive. The communication subsystem performance is expected to be the bottleneck in most emulated networking systems with accelerated control algorithms, mainly because PCI bus interconnection is slow compared to CPU and gateware processing speeds. Note that this is not inherent to the emulator design rather than the deployment platform and is a common issue of hardware software codesigned systems of this type [5]. For example, running the emulation framework on NetFPGA 10G over PCI-Express would significantly improve performance of both communication subsystem approaches. Nevertheless, measurements on the emulated subsystems can provide the relative differences in performance which should be taken into account in the selection of the final system interconnection components.

On the other hand, register I/O needs almost 3 times less reconfigurable area, which leaves a significant amount of resources available to the user logic. This may be crucial for some designs, as it was for the presented DEC-BPn scheduler.

4.3.1 Evaluation of DEC-BPn emulation

DEC-BPn algorithm was proposed and its performance investigated by simulation in [6]. We have run the emulator traffic generator engines for uniform and concentrated traffic matrices, defined in [6]. More specifically, we have used 4 different uniform traffic patterns for different input loads. Table 2 shows the performance of the emulated DEC-BPn forwarding system. As expected, register I/O is the bottleneck and defines the top operating speed of the forwarding system. On the other hand, several experimental scenarios may be implemented on the emulated DEC-BPn, including real traffic forwarding, and the relative performance results can be used to assess the performance in a final system configuration.

5. CONCLUSION

In this paper we have presented a hardware/software codesigned emulator framework for the operational verification and evaluation of hardware accelerators that implement datapath control algorithms in fast packet switches. The proposed approach is implemented on a combination of two popular frameworks, the Click Modular Router for the software datapath and the NetFPGA hardware for the control algorithm accelerators. Following the proposed accelerator integration methodology, the developer can quickly realize a full-fledged packet switch that exploits the hardware accelerator, while focusing only on devising the control algorithm.

Acknowledgements

The research leading to these results has been funded by the European Union Seventh Framework Programme under the FET project "STAMINA".

6. REFERENCES

[1] Cisco XR 12000 series and Cisco 12000 series routers. `http://www.cisco.com/`. Accessed: 2014-01.

[2] The Xilinx ISE design suite. `http://www.xilinx.com/products/design-tools/ise-design-suite`. Accessed: 2014-01.

[3] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri. Multicast traffic in input-queued switches: optimal scheduling and maximum throughput. *IEEE/ACM Trans. on Networking*, 11(3):465–477, 2003.

[4] S. Atalla, D. Cuda, P. Giaccone, and M. Pretti. Belief-propagation-assisted scheduling in input-queued switches. *IEEE Trans. on Computers*, 62(10):2101–2107, 2013.

[5] K. Chuang, S. Yalamanchili, A. Gavrilovska, and K. Schwan. ShareStreams-V: A virtualized QoS packet scheduling accelerator. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 265–268, 2008.

[6] P. Giaccone and M. Pretti. A belief-propagation approach for multicast scheduling in input-queued switches. In *Workshop on networking accross disciplines: Communication networks, complex systems and statistical physics (NETSTAT)*, pages 1403–1408, 2013.

[7] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Trans. on Communications*, 35(12):1347–1356, 1987.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. on Computer Systems*, 18(3), 2000.

[9] F. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. on Information Theory*, 47(2):498–519, 2001.

[10] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. *IEEE Trans. on Communications*, 47(8):1260–1267, 1999.

[11] G. Watson, N. McKeown, and M. Casado. NetFPGA: A tool for network research and education. In *Workshop on Architecture Research using FPGA Platforms (WARFP)*, 2006.