POLITECNICO DI TORINO
Repository ISTITUZIONALE

Design and implementation of a belief-propagation scheduler for multicast traffic in input-queued switches

Original
Design and implementation of a belief-propagation scheduler for multicast traffic in input-queued switches / Giaccone, Paolo; Pretti, Marco; Syrivelis, Dimitris; Koutsopoulos, Iordanis; Tassiulas, Leandros. - In: COMPUTER COMMUNICATIONS. - ISSN 0140-3664. - ELETTRONICO. - 103:(2017), pp. 141-152. [10.1016/j.comcom.2017.01.002]

Availability:
This version is available at: 11583/2660452 since: 2018-02-20T14:50:25Z

Publisher:
Elsevier

Published
DOI:10.1016/j.comcom.2017.01.002

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

12 October 2024

Design and Implementation of a Belief-Propagation Scheduler for Multicast Traffic in Input-Queued Switches

Paolo Giaccone[a], Marco Pretti[b], Dimitris Syrivelis[c], Iordanis Koutsopoulos[c,d], Leandros Tassiulas[e]

[a] *Dept of Electronics and Telecommunication, Politecnico di Torino, Italy*
[b] *CNR - Complex System Institute, Politecnico di Torino, Italy*
[c] *Centre for Research and Technology Hellas, Greece*
[d] *Athens University of Economics and Business*
[e] *Yale University*

Abstract

Scheduling multicast traffic in input-queued switches to maximize throughput requires solving a hard combinatorial optimization problem in a very short time. This task advocates the design of algorithms that are simple to implement and efficient in terms of performance. We propose a new scheduling algorithm, based on message passing and inspired by the *belief propagation* paradigm, meant to approximate the provably-optimal scheduling policy for multicast traffic. We design and implement both a software and a hardware version of the algorithm, the latter running on a NetFPGA. We compare the performance and the power consumption of the two versions when integrated in a software router. Our main findings are that our algorithm outperforms other centralized greedy scheduling policies, achieving a better tradeoff between complexity and performance, and it is amenable to practical high-performance implementations.

Keywords:
Multicast packet scheduling, Input queued switches, Belief propagation, NetFPGA

1. Introduction

In the last decade, input-queued (IQ) switches have been the reference switching architecture for the design of high speed routers in the Internet [3] and switches for data centers [2]. Furthermore, at a much smaller spatial scale, they are widely employed to switch data flits in Network-on-Chips [5]. The main reason is that IQ switches offer a convenient tradeoff between computational complexity and memory speed. Indeed, input buffers run at a speed equal to the line rate, so that the performance bottleneck due to the limited memory access time is minimized. Conversely, output-queued (OQ) switches always achieve optimal performance but they require very high memory speed, which is definitely unfeasible at high line rates or for large number of ports. In IQ switches, a scheduling algorithm chooses the packets to transfer from input to output ports while satisfying the switching fabric constraints, which permit at most a single packet transfer from each input port and to each output port. Finding the scheduling decision that is optimal in terms of throughput for unicast traffic requires to compute a maximum weight matching in a bipartite graph and this problem represents a reference model for a large class of resource allocation problems in computer networks. Similarly, we expect that the relevance of the multicast scheduling problem, addressed in this paper, goes beyond the scenario of IQ switches considered here.

Unicast traffic has been the predominant traffic in the Internet for a long time, but, nowadays, new applications have been arising, based on multicast traffic, in which packets are sent to a set of destinations, rather than a single one. Examples of such applications are IP video broadcasting, P2P networks and financial networks supporting high-speed trading. Moreover, in data centers multicast traffic is very relevant, due to the required data redundancy (typically, multiple copies of the same data are stored in different servers/racks) and to cooperative/parallel computations (such as MapReduce) [4]. So far, the support of multicast traffic in IQ switches is expected to have been achieved by modifying unicast scheduling algorithms in a heuristic way, without referring to the actual definition of optimal algorithms for multicast traffic [7, 15].

In this work we specifically address the problem of scheduling multicast packets in an IQ switch in order to maximize throughput, considering the optimal switching architecture in terms of queueing structure and scheduling algorithm. We propose a new distributed scheduling algorithm, inspired to the Belief Propagation (BP) paradigm, and designed to approximate a provably throughput-optimal scheduling policy. Moreover, we implement a software version of the algorithm that we integrate in a real traffic scheduler to measure performance under realistic workload. After profiling the actual resource usage under real traffic, we implement a hardware accelerator of the scheduler on a NetFPGA platform. Finally, we thoroughly evaluate the achievable performance and the power consumption of the hardware accelerator.

The paper is organized as follows. In Sec. 2 we define the multicast scheduling problem in IQ switches and we describe the optimal policy. Sec. 3 discusses some related work. In Sec. 4 we introduce the BP approach, and describe our pro-

posed scheduling algorithm. Sec. 5 compares, by simulation, the scheduler performance with other centralized greedy algorithms. The implementation of the scheduler is presented in Sec. 6 in both the software and hardware versions. The performance and power consumption of the two versions are compared in Sec. 7. Conclusions are drawn in Sec. 8.

2. Multicast traffic in input queued (IQ) switches

We consider an IQ switch of size $N \times M$ (Fig. 1), where $N = |I|$ and $M = |O|$, with I and O denoting the sets of input and output ports, respectively. Coherently with standard implementations [7, 11, 14], we assume that time is slotted and the timeslot corresponds to the duration of the internal fixed-size packets. Variable-size packets (as in Ethernet/IP packets) that are received at the input interfaces are chopped into fixed-size packets as soon as they enter the switch. These fixed-size packets are then individually enqueued and switched through a crossbar to the destination ports, where the original packets are reconstructed before being sent to the output interface. From now on, we shall always refer to the fixed-size packets transferred internally at the switch. During each timeslot, at most one packet can arrive at each input and at most one packet can depart from each output. Thus, we can define the throughput as the average number of departed packets for each timeslot, normalized by the number of output ports.

The *fanout set* of a multicast packet is defined as the set of its destination ports. Let S denote the set of all possible fanout sets, whose cardinality is $|S| = 2^M$; notably, to simplify the subsequent BP formalization, we artificially include the null fanout set in S. The adopted queueing architecture is Multicast-Virtual Output Queue (MC-VOQ), as proposed in [1], i.e., one *logical* FIFO queue is present for each possible fanout set at each input port. Let Q be the set of all possible logical queues at any input; by construction $|Q| = |S| - 1$. Thus, a maximum of $N \times |Q| = N(2^M - 1)$ queues can be defined in the whole switch. This queueing architecture, clearly poorly-scalable for large values of M if implemented using distinct physical queues, is nonetheless an interesting case, since it is *optimal*, as it avoids the well-known head-of-line blocking problem, thanks to the fact that a packet in front of a queue cannot prevent another packet behind it to be transferred. Furthermore, MC-VOQ queueing can be implemented using logical queues, by which packets in the same RAM memory are organized. In this case, the design is more scalable than using distinct physical queues, since it requires managing internal linked lists within the RAM memory and an indexing table to address specifically the first packet for each possible fanout set.

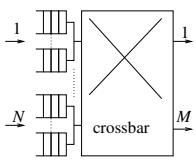

Figure 1: An $N \times M$ IQ switch with optimal MC-VOQ queueing for multicast traffic.

Combined with optimal queueing, we consider a throughput-optimal scheduling policy for multicast traffic. This policy is *fanout-splitting*, as it allows for partial packet transmissions: A packet can be sent to just a subset of its destination ports, leaving some residual destinations for future transmissions. In this case, the packet is re-enqueued to the queue corresponding to the set of residual destinations, denoted as *residual fanout set*. Such a behavior may introduce out-of-sequence packet transmissions, whose impact can be controlled and mitigated by the techniques discussed in [17].

The state and temporal evolution of the input queues can be described by a triple of time-dependent, integer-valued matrices: $Y(t)$, $A(t)$, and $D(t)$, with N rows (one for each input) and $|Q| = (2^M - 1)$ columns (one for each queue/fanout set). $Y(t) = [y_{iq}(t)]$ is the *queue length matrix* at timeslot t, whose generic entry $y_{iq}(t)$ represents the occupancy of queue $q \in Q$ at input $i \in I$ during timeslot t. Moreover, $A(t) = [a_{iq}(t)]$ is the *arrival matrix*: $a_{iq}(t) = 1$ if a new arrival occurs at input i for queue q at timeslot t, and $a_{iq}(t) = 0$ otherwise. Finally, $D(t) = [d_{iq}(t)]$ is the *service matrix* at timeslot t: $d_{iq}(t) = 1$ if queue q is served at input i, $d_{iq}(t) = -1$ if a packet is re-enqueued to q (at input i), and $d_{iq}(t) = 0$ otherwise.

$D(t)$ is computed by the scheduling algorithm and must satisfy *feasibility conditions* imposed by the switching device. The latter allows at most one packet to be sent from each input and one copy of the packet to arrive at each output, and it supports fanout splitting. These conditions can be formally defined as detailed in [1]. To understand the notation, consider a toy example of a 2×2 switch, with outputs labeled by 1 and 2. The set of all possible fanout sets is $S = \{\{1\}, \{2\}, \{1, 2\}, \emptyset\}$. The set of all possible queues is $Q = \{q_{\{1\}}, q_{\{2\}}, q_{\{1,2\}}\}$. Let us now assume that the scheduler chooses to transfer one packet from $q_{\{1\}}$ at input 1 and one packet from $q_{\{1,2\}}$ at input 2, while re-enqueueing the latter packet on $q_{\{1\}}$ at input 2, due to the conflict on output 1. The corresponding service matrix will be:

$$D = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \tag{1}$$

The queue length evolution can be described by the usual relation: $Y(t + 1) = Y(t) + A(t) - D(t)$. A traffic scenario, described by a stochastic (matrix) process $A(t)$, is said to be *admissible* if $\mathbb{E}\{A(t)\}$ does not overload any input nor any output port. In formulae:

$$\sum_{q \in Q} \mathbb{E}[a_{iq}] < 1 \qquad \forall i \in I \tag{2}$$

$$\sum_{i \in I} \sum_{q \in Q(j)} \mathbb{E}[a_{iq}] < 1 \qquad \forall j \in O \tag{3}$$

where $Q(j) \subset Q$ is the set of all queues associated to a fanout set including output j.

As shown in [1], while by construction an OQ switch yields 100% throughput under any admissible multicast traffic, an IQ switch does not, even if a throughput-optimal scheduling algorithm is adopted. An interested reader can refer to the simple

counterexample for a 2×4 switch reported in Fig. 1 of Ref. [1]. Nevertheless, it is possible to define a scheduling algorithm that maximizes the throughput (according to the formal definition reported in [1]). The throughput-optimal scheduling decision $\hat{D}(t)$ for the service matrix at timeslot t, capable of maximizing the throughput under any admissible Bernoulli i.i.d. traffic pattern, was proposed in [1] as well, and can be formalized as an ILP optimization problem maximizing the following cost function:

$$w(D, Y) = \sum_{i \in I} \sum_{q \in Q} D_{iq} Y_{iq} \tag{4}$$

defined for any service matrix D and queue length matrix Y. The optimal scheduling algorithm selects $\hat{D}(t)$ by solving:

$$\hat{D}(t) = \arg\max_{D \in \mathcal{D}} w(D, Y(t)) \tag{5}$$

where $\mathcal{D}$ denotes the set of all feasible service matrices. At each timeslot t, the algorithm chooses a feasible service matrix maximizing the total cost function w, computed for the current value of the queue length matrix $Y(t)$. Notice that this policy does not maximize the number of packets transferred at each timeslot. The resulting combinatorial optimization problem is NP-hard [1], so that in practice only approximate algorithms are viable to solve this problem. Let us remark that the combination of MC-VOQ with the optimal scheduler solving (5) is the only solution known so far in the literature to provably maximize the throughput under multicast traffic in an IQ switch.

3. Related work

Packet networks operate on resources that are shared among the nodes and can often be modeled as *constrained networks of queues*, in which packets are served according to the decisions of a scheduling algorithm. The scheduling decisions must satisfy some constraints, specific of the considered scenario.

For example, in the wireless scenario, different radio nodes can transfer packets simultaneously, if a proper diversity scheme for the communication is adopted to avoid/reduce interference among simultaneously transmitting nodes. In particular, for a FDM (or TDM, or CDMA) diversity scheme, each receiver is associated with one frequency (or, respectively, one timeslot within the frame, or one code) and the scheduler computes the packets to transmit. Packets are chosen such that at most one packet can be transferred at the same time using the same resource (frequency, temporal position, or code). Queues are placed to solve contentions for any shared resource, by storing packets waiting for their opportunity to be transmitted. Note that, in a generic packet network, the switching constraints among the queues may be more complex than the ones presented above.

Pioneering work in [21] has devised the optimal scheduling policy in generic constrained queueing networks. Such a policy, called "max-pressure", is provably optimal and it achieves the maximum throughput under any admissible i.i.d. Bernoulli traffic. Results of [21] are universal and have stimulated a huge interest in the research community, which has devoted many efforts to apply/extend these results in many contexts regarding wireless and wired networks. One, extensively studied, scenario is the scheduling problem in an IQ switch considered in our work. This scenario can be seen as a one-hop constrained queueing network, in which the optimal max-pressure policy degenerates into the Maximum Weight Matching (MWM) policy. The latter, proved to be optimal also in [12], is not practically implementable, but it has inspired the design of a huge number of scheduling algorithms.

However, results in [12, 21] only refer to unicast traffic. So far, few results have been obtained regarding the optimal scheduling policy for multicast traffic in the context of constrained queueing systems. Notably, in the generic context of multihop networks of queues, [17] describes how to schedule optimally the multicast traffic generated by a set of multicast sessions across a given set of multicast trees. At odds with our scenario, a packet can only be transferred from each node to a new one, but it cannot be re-enqueued within the same node. The latter feature does not allow one to achieve maximum throughput in the specific single-hop queueing system represented by an IQ switch. The work [17] defines a weight for each tree, corresponding to the state of "congestion" associated to it. The scheduler chooses the tree to be served based on its weight, computed similarly to the max-pressure policy.

In the context of IQ switches, many papers have addressed the problem of scheduling multicast traffic, but without any flavor of optimality. Most of the previous work has focused on architectures with just one queue per input, which is obviously non-optimal (even in the unicast case), because of the heavy head-of-line blocking experienced by the traffic. For example, [15] has investigated the tradeoff achievable among concentration of residual fanout set, fairness and implementation complexity for scheduling algorithms based on one single queue per input. These results were extended to variable size multicast packets in [22].

Adopting a possibly large number of queues per input (i.e., one for each possible fanout set) [1] has proposed the optimal policy maximizing the throughput under multicast traffic; furthermore, it has highlighted the intrinsic performance limitations of IQ switches under multicast traffic. However, the proposed algorithm for optimal multicast scheduling requires to solve a very complex combinatorial optimization problem, which cannot be solved in practice. Our contribution is to show how to approximate efficiently the optimal scheduling algorithm of [1].

Notably, [19] considers a completely different approach, based on the standard VOQ architecture designed for unicast traffic and on a classical scheduler for unicast traffic. The scheme works as follows. Whenever a multicast packet arrives, it is enqueued in the VOQ corresponding to one destination in its fanout set. The scheduler chooses the VOQs to serve as if the traffic was unicast. When the packet is served, just one copy is sent to the output corresponding to the VOQ. If some residual fanout is left, the packet is re-enqueued in one VOQ corresponding to any of its residual fanout. Thanks to the induced load-balancing across all the VOQs, the proposed scheme is able to achieve maximum throughput, at the expense

of possible out-of-sequence problems. Note that the proposed approach, even though very promising and practically relevant, does not exploit the multicast capabilities of the switching fabric, which is instead considered in our work.

A preliminary version of our work appeared in [6] and in [20], where we proposed our novel approach based on Belief Propagation (BP), investigated its performance and proposed an efficient implementation. BP is a well-established methodology to solve combinatorial optimization problems. As shown in [10], by constructing a proper *factor graph* (like the one tailored to our scheduling problem, described in Sec. 4), it is in principle possible to compute the solution of the problem by a distributed message-passing algorithm. The nodes in the factor graph exchange real-valued messages (intuitively, representing the local "belief" of the optimal solution), based on "propagation equations" that are specific to the problem considered. The construction of BP equations is conceptually a well-established issue, even though non-trivial manipulations are often required to put them in a conveniently simple form. Note that, even though we generically speak of BP, our proposed algorithm is of the *min-sum* type [10], which can be regarded as a special case, specifically suited for computing MAP (maximum a posteriori probability) estimates. Quite recently, [16] highlighted the relevance of methods borrowed from statistical physics to solve complex combinatorial optimization problems in the field of networking. Our BP-inspired approach is an example of such methods.

4. Belief Propagation (BP) approach

Our BP-based scheduling algorithm runs at each timeslot and solves (5) based on the current state of the queues. For the sake of simplicity, we will omit the time index t from the following notation. We can observe that a service matrix $D \in \mathcal{D}$ can be equivalently represented by N pairs of fanout sets $\sigma_i, \tau_i \in S$, one for each input $i \in I$, as follows:

$$D \;\Leftrightarrow\; [\sigma_i, \tau_i]_{i \in I} \tag{6}$$

In particular, σ_i (if nonempty) represents the served queue, τ_i the subset of outputs to which the packet is actually transmitted (*transmission* fanout set), and $\sigma_i \setminus \tau_i$ the queue in which the packet is possibly re-enqueued (*residual* fanout set). By construction,

$$\sigma_i \supseteq \tau_i \qquad \forall i \in I \tag{7}$$

Note that $\sigma_i = \emptyset$ (empty fanout set), whence $\tau_i = \emptyset$, means that no queue is served and the input port i does not transmit anything. We can attribute the same meaning of no transmission at input i even to degenerate configurations with $\sigma_i \neq \emptyset$ and $\tau_i = \emptyset$, so that $\sigma_i \setminus \tau_i = \sigma_i$ (i.e., the packet is re-enqueued in the served queue). Apart from the latter degenerate case, which is avoided by the scheduler, we can reconstruct the service matrix from $[\sigma_i, \tau_i]$ as

$$d_{iq} = \begin{cases} 1 & \text{if } q = \sigma_i \\ -1 & \text{if } q = \sigma_i \setminus \tau_i \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Moreover, to complete the feasibility constraints, we must avoid conflicting packets at each output, namely, we have to impose the following service constraints:

$$\sum_{i \in I} \chi\{\tau_i \ni j\} \leq 1 \qquad \forall j \in O \tag{9}$$

where $\chi\{\cdot\}$ denotes a characteristic function, equal to 1 if the condition denoted by the argument is verified (i.e., input i transmits to output j), and 0 otherwise.

To clarify this alternative notation, the service matrix (1) of the toy example introduced above admits the fanout variable representation $[\sigma_1, \tau_1, \sigma_2, \tau_2]$, defined as follows: $\sigma_1 = \tau_1 = \{1\}$ (i.e., one packet from input 1 to output 1, without re-enqueueing), $\sigma_2 = \{1, 2\}$ (i.e., one packet from input 2 to outputs 1 and 2) and $\tau_2 = \{2\}$ (i.e. the packet in σ_2 is actually transferred only to output 2, due to the conflict with σ_1, and a copy is re-enqueued in the queue towards output $\sigma_1 \setminus \tau_1 = \{1\}$).

We can conveniently adapt the queue length matrix definition to the new notation. Let y_{is} be the length of the queue associated to the fanout set $s \in S$ at input $i \in I$. We assume $y_{i,\emptyset} = 0$. Now, thanks to (8), it is possible to rewrite the cost function (4) as a function of the fanout set variables σ_i, τ_i and claim the following:

Lemma 1. *In an IQ switch with MC-VOQ, the throughput-optimal scheduling policy computes the service matrix at time t as*

$$\hat{D} = \arg\max_{D \in \mathcal{D}} \sum_{i \in I} \left(y_{i\sigma_i} - y_{i(\sigma_i \setminus \tau_i)} \right) \tag{10}$$

Lemma 1 provides an important insight in the optimal scheduling policy: the adopted cost function represents *the difference between the length of the served queue and the length of the queue where the residual fanout set is eventually re-enqueued*. This difference will be denoted as *max-pressure weight of a queue*, because it clearly turns out to be an extension of the aforementioned universal max-pressure policy [21], in which the weight of serving one queue is computed as the local queue length minus the (downstream) queue length where the packet is sent. In our specific case, the downstream queue corresponds to the queue where the residual fanout set is re-enqueued.

When considering the cost function in (10), it is worth noting that, for some given service matrix $D \in \mathcal{D}$, the elementary contribution $y_{i\sigma_i} - y_{i(\sigma_i \setminus \tau_i)}$ to the cost function may be negative for some input i, if the queue in which the packet would be re-enqueued is longer than the served queue. In this case, it is possible to improve the overall cost function $w(D, Y)$ by *not* serving the packet from input i in D. Thus, we can argue that the throughput optimal scheduling policy avoids serving a queue whose occupancy is smaller than that of the queue where the packet would be re-enqueued. This behavior may imply some delay impairment at low loads, due to the missed opportunity of transmitting a packet.

Now, it is important to observe that the service constraints (9) involve variables τ_i associated to different inputs, whereas, for a given input i, the variable σ_i is only coupled to the corresponding τ_i, by the condition (7). As a consequence of Lemma 1, the

optimal σ_i for a given choice $\tau_i = \tau$, which we shall denote as $\hat{\sigma}_{i\tau}$, can be determined by a *local* maximization at each input i,

$$\hat{\sigma}_{i\tau} = \arg \max_{\sigma \in S \,|\, \sigma \supseteq \tau} \{y_{i\sigma} - y_{i(\sigma \setminus \tau)}\} \tag{11}$$

We define also the optimized "local" weights

$$w_{i\tau} \triangleq \max_{\sigma \in S \,|\, \sigma \supseteq \tau} \{y_{i\sigma} - y_{i(\sigma \setminus \tau)}\} \tag{12}$$

Note that (11) identifies, for each input, the best candidate queue ($\hat{\sigma}_{i\tau}$) to transmit towards each set of destinations τ, and (12) evaluates the corresponding weight. The original optimization problem in Lemma 1 is then reduced to a (constrained) optimization over the sole τ_i variables (transmission fanout sets). The optimal solution can be written as

$$[\hat{\tau}_i]_{i \in I} = \arg \overset{\checkmark}{\max_{[\tau_i]_{i \in I}}} \sum_{i \in I} w_{i\tau_i} \tag{13}$$

where the check mark recalls that the optimization is constrained by (9).

4.1. The construction of the factor graph

Thanks to (13), the combinatorial optimization problem can be solved using a *factor graph* [10]. The latter is a bipartite graph, whose two species of nodes (called *variable nodes* and *function nodes*) are associated respectively to the decision variables and to the couplings among them. An arc between a function node and a variable node means that the corresponding variable is involved in the corresponding coupling.

In our problem, a convenient set of decision variables for the factor graph is defining $x_{ij} = 1$ if $\tau_i \ni j$ (i.e. the fanout set τ_i comprises output j) and 0 otherwise. This allows us to write $x_{ij} \triangleq \chi\{\tau_i \ni j\}$ in the service constraints (9), which completely specify any transmission fanout set as $\tau_i = \{j \in O \,|\, x_{ij} = 1\}$. In terms of these variables, we can identify two different kinds of couplings, namely, the local weights $w_{i\tau_i}$, appearing in (13), and the constraints (9) themselves. Each local weight is associated to an input i and involves variables $x_{i1}, \ldots, x_{iM}$, whereas each service constraint is associated to an output j and involves variables $x_{1j}, \ldots, x_{Nj}$.

The factor graph associated to our problem can be obtained by constructing a fully connected $N \times M$ bipartite graph, whose N left-most nodes correspond to the inputs and M right-most node correspond to the outputs, and each left node is connected to all the output nodes and vice versa. Then, we "cut" each arc (ij) and connect each pair of "dangling bonds" to a new (variable) node, while the original nodes ($i \in I$ and $j \in O$) become the function nodes of the factor graph. As an example, Fig. 2 shows the factor graph for a 2×3 switch. The right-most nodes represent the coupling due the service constraints. The middle nodes represent the decision variables x_{ij}. Finally, the left-most nodes represent the local weight associated to the chosen transmission fanout τ_i at input i, computed based on the incident decision variables x_{ij}, $\forall j \in O$. For example, let us assume $\sigma_1 = \tau_1 = \{1, 2\}$, $\sigma_2 = \{1, 2, 3\}$ and $\tau_2 = \{3\}$. In the factor graph, we would have $x_{11} = x_{12} = x_{23} = 1$ and $x_{13} = x_{21} = x_{22} = 0$. In

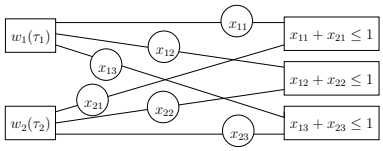

Figure 2: Factor graph for a 2×3 multicast switch. Circles and rectangles denote variable and function nodes, respectively.

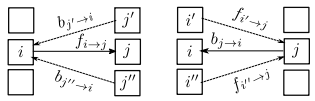

Figure 3: Messages exchange in BP algorithm among input and output nodes.

conclusion, the factor graph associated to our problem is of the type sketched in Fig. 2 with $N+M$ function nodes, NM variable nodes and $2NM$ edges. This guarantees the scalability of the proposed approach, since the factor graph size does not scale with the number of the queues (growing as $N2^M$).

Using the BP algorithm, the solution is obtained throughout a distributed message-passing algorithm running among the input function nodes and the output function nodes of the factor graph. "Forward" messages ($f_{i \to j}$) are sent from the inputs to the outputs and "backward" messages ($b_{j \to i}$) from the outputs to the inputs, as depicted in Fig. 3. Notably, the forward messages are computed based on the backward messages, and vice versa, in an iterative and distributed way. When the values of the messages converge, the final service configuration is computed locally at the nodes.

In the following we report the final BP equations, whose derivation is rather technical [6]. We can define the *beliefs*, associated to each transmission fanout set variable τ_i, as

$$m_{i\tau} = w_{i\tau} - \sum_{j \in \tau} b_{j \to i} \tag{14}$$

These quantities represent, apart from an irrelevant additive constant, an estimate of the weight that can be obtained by choosing a specific value $\tau_i = \tau$. Moreover, the backward messages $b_{j \to i}$ are an estimate of the weight degradation due to possible conflicts generated at output j by the choice $x_{ij} = 1$, i.e., $j \in \tau_i$ (transmission from i to j). These messages are defined by suitable self-consistency equations, namely

$$b_{j \to i} = \max_{i' \in I \setminus i} f_{i' \to j} \tag{15}$$

$$f_{i \to j} = \max \left\{0\,,\, \max_{\tau \in S \,|\, \tau \ni j} m_{i\tau} + b_{j \to i} - \max_{\tau \in S \,|\, \tau \not\ni j} m_{i\tau}\right\} \tag{16}$$

where the "forward" messages $f_{i \to j}$ can be finally regarded as an estimate of the weight gain that can be obtained by the single choice $x_{ij} = 1$ (rather than 0). The solution of these self-consistency equations by iterative refinement involves message passing from input to output ports (forward messages) and vice versa (backward messages). It is a well known fact that BP likely converges, if the underlying factor graph is treelike (notably, equations are exact if the graph is rigorously a tree). In

```
DEC-BPn (input: [y_iσ]_{i∈I,σ∈S} ; output: [σ_i, τ_i]_{i∈I})
0.  for i ∈ I and τ ∈ S , compute w_iτ and σ̂_iτ by (12) and (11)
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 (16) and (14)
          for i ∈ Ĩ and j ∈ Õ, compute b_{j→i} by (15)
5.    for i ∈ Ĩ and τ ∈ S | τ ⊆ Õ, compute m_iτ by (14)
6.    choose i ∈ Ĩ and τ ∈ S | τ ⊆ Õ that maximize m_iτ
7.    if m_iτ = 0, set τ = ∅
8.    set τ_i = τ and σ_i = σ̂_iτ
9.    set Ĩ = Ĩ\i and Õ = Õ\τ_i
10. return [σ_i, τ_i]_{i∈I}
```

Figure 4: DEC-BPn algorithm (decimation with n BP iterations).

```
GR-LQF (input: [y_iσ]_{i∈I,σ∈S} ; output: [σ_i, τ_i]_{i∈I})
1.  set Ĩ = I, Õ = O, and σ_i = τ_i = ∅ for i ∈ I
2.  while Ĩ ≠ ∅
6.    choose i ∈ Ĩ and σ ∈ S | σ ∩ Õ ≠ ∅ that maximize y_iσ
7.    if not found or y_iσ = 0, break
8.    set σ_i = σ and τ_i = σ ∩ Õ
9.    set Ĩ = Ĩ\i and Õ = Õ\τ_i
```

```
GR-RND (input: [y_iσ]_{i∈I,σ∈S} ; output: [σ_i, τ_i]_{i∈I})
    (...)
6.    choose random i ∈ Ĩ and σ ∈ S | σ ∩ Õ ≠ ∅ such that y_iσ > 0
7.    if not found, break
    (...)
```

Figure 5: Greedy algorithms GR-LQF (longest queue first) and GR-RND (randomly chosen queue). Dots denote that some steps of the latter algorithm are fully equivalent to those of the former.

our case, the factor graph is densely connected, and, consistently, we find several instances of the problem in which BP does not converge. Because of this problem, it is not possible to use directly the beliefs (14) to fix the decision variables, since this may lead to unsatisfied service constraints. This is why we have resorted to use BP with a fixed number of iterations, in conjunction with a simple decimation algorithm, which at each iteration fixes a given variable $\tau_i = \tau$ with the maximum belief $m_{i\tau}$, simplifies the equations to be compatible with the choice taken, and then reruns BP. The resulting algorithm is described by the pseudocode reported in Fig. 4.

The proposed algorithm, denoted as DEC-BPn, takes as input the queue length matrix Y at timeslot t and returns the scheduling decision, in terms of the fanout set variables σ_i, τ_i for each input i. Referring to the pseudocode in Fig. 4, step 0 performs the local optimization procedure, defined by (11) and (12), obliviously of the feasibility constraints of the service matrix. These constraints are considered instead in the following steps. The "sets" $\tilde{I}$ 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 3–5 represent three different phases of BP, namely, initialization of backward messages, computation of forward messages as a function of backward messages and vice versa (with n iterations), and computation of beliefs (as a function of backward messages). 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 $m_{i\tau}$ (when the maximum is not unique, we randomly solve the tie, also to improve the scheduling fairness). 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, when the decimation loop is over, the current values of the fanout set variables define univocally a feasible service matrix D, computed as (8), which is used to configure the switching fabric.

5. DEC-BPn performance by simulation

In this section, we evaluate the performance of DEC-BPn by means of simulations obtained by an ad-hoc event-driven simulator written in C++. We compare the results obtained for our algorithm against other centralized scheduling algorithms, designed to support multicast traffic, under different traffic conditions. The latter algorithms are greedy approaches, operating two slightly different strategies, described by the pseudo-codes in Fig. 5. Note that the overall structure of both algorithms is similar to that of DEC-BPn, even though the steps typical of BP (0 and 3–5) are missing. The characterizing step is in fact only 6: GR-LQF chooses the longest queue, whereas GR-RND chooses a random queue, provided, in both cases, that the corresponding fanout set includes some available outputs.

The input traffic is generated according to a Bernoulli i.i.d. arrival process, in which ρ is the average input load, or equivalently the probability that a packet arrives at an input port during a timeslot. The corresponding fanout set is chosen at random in a possible set of candidate ones, as described below. The traffic admissibility conditions in (2) and (3) imply $\rho \leq \rho_{\max}$, where $\rho_{\max} = M/(Nf)$ is the maximum admissible input load and f is the average *fanout* (i.e., the average cardinality of the fanout set).

We consider two different families of candidate fanout sets. The first one is referred to as *uniform traffic* and derived from [15]: The fanout set of each packet is chosen at random among all possible $2^M - 1$ ones. For this case, it can be shown that $f = \frac{M 2^{M-1}}{2^M - 1}$ and $\rho_{\max} = \frac{2^M - 1}{N 2^{M-1}}$. Note that, since $\rho_{\max} \approx \frac{2}{N}$, the input load is quite small for large switches, and this fact prevents the arrival of "critical" traffic patterns. This observation motivates the other traffic family, which has been devised in such a way to keep $\rho_{\max}$ independent of the switch size. The latter family is referred to as *concentrated traffic* and corresponds to the worst-case traffic model presented in [1]. Such a model was thoroughly designed to create extensive contention among inputs and was crucial in [1] to show the intrinsic throughput limitations of IQ switches under multicast traffic. Without going into the details of their construction, in Table 1 we describe three different concentrated traffic scenarios (denoted as "Conc-1", "Conc-2", and "Conc-3"), reporting the corresponding $\rho_{\max}$

Table 1: Fanout sets for each concentrated traffic scenario.

Traffic	$\rho_{\max}$	Input 1	Input 2	Input 3
Conc-1 2×4	1.00	$\{1,2\}$ $\{3,4\}$	$\{1,3\}$ $\{2,4\}$	not used
Conc-2 2×4	0.67	$\{1,2,3\}$ $\{2,3,4\}$	$\{1,2,4\}$ $\{1,3,4\}$	not used
Conc-3 3×12	1.00	$\{1,2,3,4\}$ $\{5,6,7,8\}$	$\{1,5,9,10\}$ $\{2,6,11,12\}$	$\{3,7,9,11\}$ $\{4,8,10,12\}$

Table 2: Maximum throughput under uniform traffic.

Traffic	Uniform 10×10	Uniform 4×10	Uniform 2×10
$\rho_{\max}$	0.20	0.50	1.00
Algorithm	Throughput		
GR-LQF	1.00	0.86	0.64
GR-RND	1.00	0.92	0.75
DEC-BP0	1.00	0.98	0.95
DEC-BP1	1.00	0.98	0.95
DEC-BP2	1.00	0.97	0.95
DEC-BP4	1.00	0.97	0.95

and the list of all fanout sets for each input.

In order to compare the algorithms, we have evaluated both the throughput and the average delay. Throughput is evaluated in terms of maximum sustainable load at the outputs; this is a value between 0 and 1, representing the maximum fraction of timeslots exploited to transmit a packet at the outputs. Even though the traffic is admissible, the throughput may be less than one, even for the optimal scheduling algorithm, because of the aforementioned intrinsic throughput limitations [1]. The delay is evaluated as the average time interval between the timeslot when a packet enters the switch and the one when all its copies leave the switch. All the reported results have been obtained with a minimum 5% of accuracy computed on a 95% confidence interval.

5.1. Simulation results

Let us start by comparing the performances under uniform traffic. Table 2 shows the maximum achievable throughput under three uniform scenarios. When considering the symmetric traffic scenario (second column), all the algorithms behave exactly in the same way and achieve maximum throughput. This is due to the low input load (always less than $\rho_{\max}$, to be admissible), which does not generate "critical" loading conditions, as already observed in the previous section. Conversely, when concentrating the traffic on few inputs (4 and 2), performances are different, and, in both scenarios, DEC-BP outperforms the other two centralized greedy approaches, independently of the number of iterations, whose actual value does not really affect the final performance. Thus, in the following we will consider DEC-BP0 as the best candidate algorithm for multicast scheduling.

Fig. 6 shows the average delays under the three uniform scenarios. Here, we do not report the curves for DEC-BPn for $n \geq 1$, as they turn out to be fully overlapped with that of DEC-BP0. Fig. 6(a) shows that, under symmetric traffic, all the algorithms achieve maximum throughput (for $\rho = \rho_{\max}$) but the delay in the low load regime is worst for DEC-BP0. This effect is to be ascribed to the specific form of the cost function

Table 3: Maximum throughput under concentrated traffic.

Traffic	Conc-1 2×4	Conc-2 2×4	Conc-3 3×12
$\rho_{\max}$	1.0	0.67	1.0
Algorithm	Throughput		
OPTIMAL	0.75	0.99	-
GR-LQF	0.70	0.91	0.63
GR-RND	0.69	0.89	0.61
DEC-BP0	0.75	0.97	0.66
DEC-BP1	0.75	0.97	0.66
DEC-BP2	0.75	0.97	0.66
DEC-BP4	0.75	0.98	0.66

in Lemma 1, which does not directly minimize the queue sizes (at odds with the maximum weight matching for unicast traffic), and trades higher delays at low load with higher throughput at high loads. Note however that the delays for low load experienced by DEC-BP0 are negligible in absolute terms. The other two uniform scenarios point out some relevant differences among the three algorithms, especially in terms of throughput. Table 2 shows that DEC-BP0 always outperforms both greedy approaches, with a gain between 6% and up to 48%. In terms of delays, Figs. 6(b)-(c) display a behavior similar to Fig. 6(a) for low load, but different when the load is higher, due to the different maximum throughput. Let us finally note that, when the traffic is no longer sustainable, the delays appear still finite because of the finite queues; otherwise, they would have grown to infinity.

We now consider the concentrated traffic scenarios. Table 3 displays the achievable throughput for all the policies considered so far, with the addition of OPTIMAL algorithm, which simply finds the optimal solution of (10) by an exhaustive search over the whole solution space $\mathcal{D}$. We could not simulate OPTIMAL for $M > 4$, due to the large computational effort required. Recall now that OPTIMAL is the only provably-optimal scheduling policy that maximizes throughput for multicast traffic. Our results show that, even in this case, the effect of the number of iterations in DEC-BPn is negligible, which again promotes DEC-BP0 as the best scheduling algorithm. Notably, in both Conc-1 and Conc-2 traffic scenarios, DEC-BP0 achieves the same performance as OPTIMAL. As in the previous scenarios, DEC-BP0 outperforms the other greedy approaches, with throughput gains between 5% and 10%.

Fig. 7 shows the delays under concentrated traffic. All the curves exhibit a similar behavior, coherent with the achieved maximum throughput. Furthermore, we can observe an interesting property of OPTMAL policy: In the low load regime, the delay is larger than for the other policies, as observed for DEC-BP0 for uniform traffic. Indeed, OPTIMAL maximizes the throughput, but does not always minimize delays. As already observed when discussing uniform traffic, this is due to the cost function in (10) which does not minimize the queue lengths. The latter observation corroborates our previous argument, namely, that the higher delays experienced by DEC-BP0 are mainly due to the fact that this algorithm approximates the optimal policy.

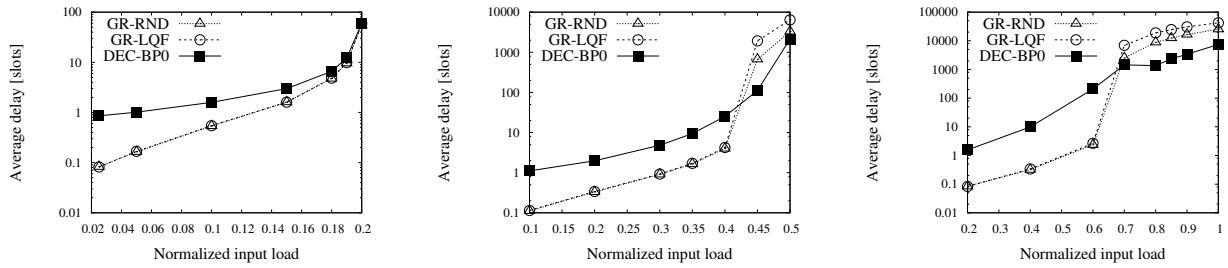

Figure 6: Average delay under uniform traffic: (a) 10×10, (b) 4×10, (c) 2×10.

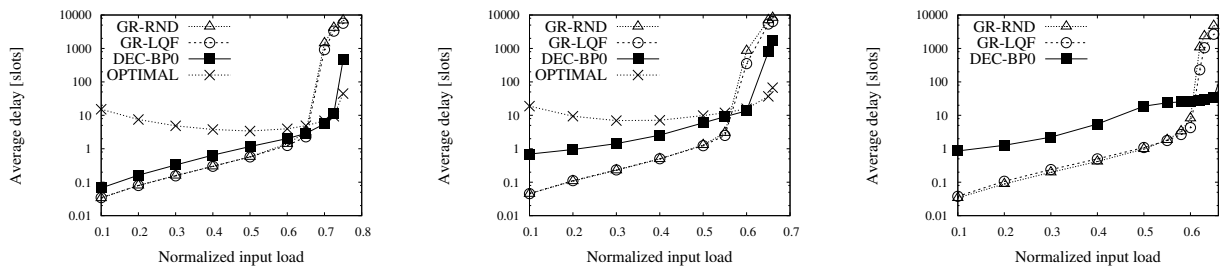

Figure 7: Average delay under concentrated traffic: (a) Conc-1, (b) Conc-2, (c) Conc-3.

5.2. Results under random queue-length matrices

From the simulation results reported so far, increasing the number of iterations n in DEC-BPn does not appear to provide a meaningful performance improvement. For this reason, we also investigate the effect of n on the efficiency of DEC-BPn in a slightly different setting, namely, on uncorrelated instances of the optimization problem defined in (10). As a term of comparison, we consider GR-LQF, which was previously shown to be the best competing algorithm.

We take a random queue-length matrix $Y = [y_{iq}]$, where y_{iq} is generated according to a geometric distribution with average 100 (i.e., the queues are loaded with 100 packets on average). We run both DEC-BPn and GR-LQF on Y. Let D_{BP} be the service matrix computed by DEC-BPn and let D_{LQF} be the one computed by GR-LQF, defined as:

$$D_{BP} \;\Leftrightarrow\; [\sigma_i^{BP}, \tau_i^{BP}]_{i \in I} \qquad D_{LQF} \;\Leftrightarrow\; [\sigma_i^{LQF}, \tau_i^{LQF}]_{i \in I}$$

We define the cost-gain factor g as the ratio between the corresponding cost functions in (10):

$$g = \frac{\sum_{i \in I} \left(y_{i\sigma_i^{BP}} - y_{i(\sigma_i^{BP} \setminus \tau_i^{BP})} \right)}{\sum_{i \in I} \left(y_{i\sigma_i^{LQF}} - y_{i(\sigma_i^{LQF} \setminus \tau_i^{LQF})} \right)}$$

We expect on average $g > 1$, since, from the results in Sec. 5.1, DEC-BPn always achieves higher throughput than GR-LQF, which means that it finds on average a better solution to (10) with respect to GR-LQF.

Fig. 8 shows the average value of g obtained for DEC-BPn when the number of iterations n grows from 0 to 32, for different switch sizes. These results have been obtained by averaging g on a number of random matrices varying between 100 and 10,000, in order to guarantee that a minimum 5% of accuracy was achieved with a 95% confidence interval. The considerable performance improvement of DEC-BPn upon increasing n is now evident, as g reaches 1.5 (for small switches) up to 3.5 (for large switches) when the number of iterations is large enough. These results also imply that, at least theoretically, the stability region of DEC-BPn, even though not optimal, may be larger than GR-LQF by a factor g. In other words, for some (unknown) worst-case scenario, the expected throughput of DEC-BPn might be 50% (for small switches) or 300% (for large switches) larger than that achieved by GR-LQF. Furthermore, only few iterations (up to 5) are sufficient to achieve almost the maximum cost gain in DEC-BPn. Notably, with no iteration, DEC-BP0 achieves an average cost gain between 1.2 and 1.7 for switches strictly larger than 4×4, thus we expect a possible throughput increase between 20% and 70% with respect to GR-LQF.

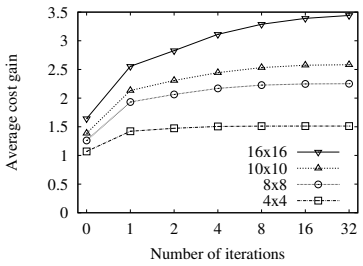

Figure 8: Average cost gain of DEC-BPn with respect to GR-LQF

In conclusion, DEC-BPn appears to be more robust with a number of iterations slightly larger than 0. This shows that the

BP messages updates in step 4 of DEC-BP*n* (Fig. 4) play a relevant role to optimize the performance of the scheduling algorithm, at the price of a negligible increase of complexity with respect to DEC-BP0.

6. A DEC-BP*n* scheduler system design

The simulation results, reported in the previous section, demonstrate that DEC-BP*n* scheduler achieves better throughput than standard greedy approaches that are expected to be commonly used in practical implementations [7, 11, 14]. On the other hand, such simulations do not provide any insight about the actual execution performance, the integration, and the resource utilization of an implementation that must operate at linespeed. In order to further assess the algorithm performance, we have designed and implemented (i) a software library version of the scheduler, that can be integrated in software routers or software packet processors, and (ii) a hardware description language library version, referred to as *gateware* version, which can be directly integrated in the hardware implementation of high performance switches. We have evaluated the performance for both versions. For the gateware version, we have evaluated also the required hardware resources.

In Sec. 6.1 we describe the general design of the scheduler, and in Secs. 6.2 and 6.3 we discuss the implementation in software and in gateware, respectively.

6.1. DEC-BPn processing components

The scheduler processing components are depicted in Fig. 9. The scheduler takes as input the length of all MC-VOQ queues and computes: i) the best candidate queue to serve at each input and the corresponding max-pressure weight (step 0 in the pseudocode of Fig. 4) and ii) the final service matrix produced from the BP iterations (steps 1 - 9 in Fig. 4). The processing has been therefore separated in two sequential steps presented with separated boxes in Fig. 9. Initially, the max-pressure weight calculations (i.e. computing all the differences $y_{i\sigma} - y_{i(\sigma\setminus\tau)}$ in (11)) and comparisons take place, and in the sequel the BP forward and backward message exchange iterations are executed. The max-pressure calculations can be performed independently for each input port and thus the available level of parallelism can be fully exploited at this step. On the other hand, the BP iterations need the generalized weight results at the beginning (step 2 in Fig. 4) so they have to start after the first step has finished. Forward and backward message exchange is performed in loops, where, in addition, the result of each iteration is used as feedback for the next one. Thanks to the high level of parallelism of the message exchange, also this step can be parallelized.

The scheduler needs to be tightly integrated with the datapath of the switch because it has to be invoked at every new packet arrival, to support the linespeed forwarding. Also the MC-VOQ queueing architecture must be managed at linespeed. This implies that each queue must be able to support at each timeslot two writes (one arrival and possibly one re-enqueueing) and one read (one packet departure due the scheduler decision). We have implemented a software version of the overall datapath of

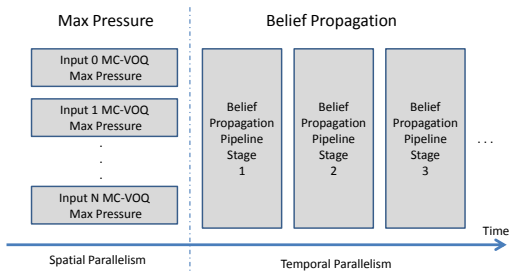

Figure 9: DEC-BP*n* processing components, exploiting both temporal and spatial parallelism

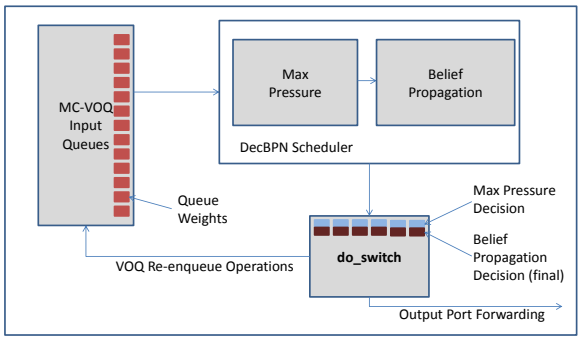

Figure 10: A high-level schematic of DEC-BP*n* scheduler integrated with the forwarding datapath of the software router

a 4×4 switch in the Linux kernel of a server using the "click modular router" packet processor framework [9]. In our scenario, a maximum of $4 \times (2^4 - 1) = 60$ queues are required to implement MC-VOQ for all the inputs. The scheduler has been implemented in two versions: a software version running on the same Linux server and a gateware version running on a hardware accelerator integrated in the same server. A whole schematic of the integrated datapath is depicted in Fig. 10, described in details in the following sections.

In order to validate the two implemented versions, we have developed a traffic generator residing in the Linux kernel to avoid system calls and packet memory copy overheads. This software module generates 1500-byte Ethernet packets for each input and each packet gets annotated with the fanout set bitmask. The arrival process is generated using the same sequence of packets used to simulate DEC-BP*n* in Sec. 5.

6.2. DEC-BPn software version

We have devised a special encoding scheme, based on bitwise-operations (typically used in hardware designs), to describe the fanout set of packet and to index queues; this allowed us to perform very efficiently operations on fanout sets and queues, enabling linespeed performance. More specifically, bitmasks have been used to represent the fanout set of a packet and the corresponding queue. In each bitmask, each bit position is reserved for a respective port (e.g. the most significant

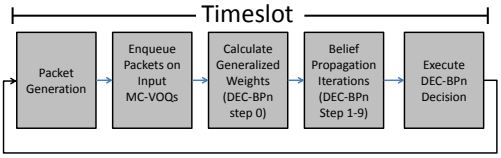

Figure 11: Temporal sequence of tasks in the datapath of the software router. The third and fourth block corresponds to DEC-BP*n* scheduler.

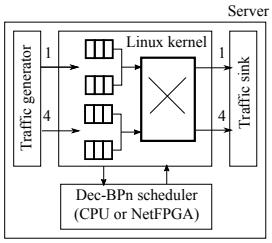

Figure 12: Implementation of the datapath for 4×4 software router on a Linux server

bit is reserved for port 0). A bit value equal to 1 indicates that the respective port belongs to the set represented by the current bitmask. As a result, a few bitwise operations can determine whereas a port belongs to a set or not, and queue-head pointers are directly indexed by the respective bitmask values, thus enabling instant retrieval. The same encoding scheme is used for the scheduler decision, so that the switching fabric can exploit bitwise operations to identify the path for the desired output destinations of a transmitted packet. Moreover, the DEC-BP*n* software version spawns a separate thread for each input port that calculates the max-pressure (step 0 in Fig. 4). All these threads need to join at the beginning of the BP calculations (steps 1 - 9 in Fig. 4).

The sequence of the operations is depicted in Fig. 11, showing also the time interval corresponding to a timeslot. During each timeslot, the traffic generator sends the packet to the designated inputs. The arriving packets are enqueued into the proper queue in the MC-VOQ system and the scheduler is triggered. Thus, the max-pressure weight is computed for each queue, and this process runs in parallel for each input port, coherently with Fig. 10. During the final execution of the scheduler decision, the packets are forwarded to their destination and, in the case of fanout splitting, the packet is re-enqueued in the correct queue.

A simple profiling on the software scheduler, running on the hardware system described in Sec. 7.1, revealed that the scheduler execution occupies 94% of the timeslot, while actual packet switching operation and queue manipulation operations account for 6%. This was expected because all the packet enqueue/dequeue operations rely only on pointer arithmetic operations which take place very efficiently on instruction set processors.

6.3. DEC-BPn gateware version

Motivated by the large execution time of the software version of the scheduler, highlighted above, we decided to explore the potential of a hardware design of the scheduler, that could be integrated in the datapath of a real switch. Therefore, we have designed and implemented a gateware version of the scheduler, using the Verilog hardware description language. The gateware DEC-BP*n* is a state machine tailored to a 4×4 switch.

6.3.1. Scheduler communication interface

The gateware DEC-BP*n* scheduler exports 61 16-bit registers at the input. The first 60 registers are used for passing all the MC-VOQ queue lengths. The last register acts as the control register and is used to initiate calculation and indicate when the decision is ready. At the output, 8 4-bit registers are used to represent the scheduler decision $D = [\sigma_i, \tau_i]_{i=1}^{4}$ (i.e. the queue to serve σ_i and the corresponding destinations τ_i, for any input i), using the same representation as (6). Note that 4 bits are required to represent each σ_i and τ_i, thanks to the bit-wise encoding scheme described in Sec. 6.2.

The sequence of operations in the gateware scheduler is as follows. The input registers of the scheduler are updated with the lengths of all 60 MC-VOQ queues. 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 the appropriate forwarding operations as well as MC-VOQ re-enqueueing have to be performed by the datapath logic.

6.3.2. Scheduler state machine

The software version of the scheduler has been heavily restructured to be mapped to gateware. All the iterative loops appearing in the pseudocode of Fig. 4 have been transformed as follows: i) the loops performing independent operations on distinct data have been "unrolled", so that hardware may execute all operations on a single cycle; ii) the loops that use the feedback from the previous cycle for the calculations during the current cycle have been converted to state machines. As a result, the gateware scheduler design features 81 states that compute the max-pressure weights for all 4 inputs, with each state performing in parallel the required operations for all 4 input ports. Additional 68 states compute the forward messages and additional 53 states compute the backward messages, during each BP iteration. Finally, additional 71 states perform all the necessary matching and comparison operations to reach the final decision. In total, for 3 hardcoded BP iterations (i.e. $n = 3$) the gateware scheduler needs 515 cycles to produce the final decision. The combinatorial logic within each state has been carefully placed to minimize critical path delay, so that the overall design can operate at high clock rates.

7. DEC-BP*n* experimental evaluation

In order to test the implementation in a real system setup, we have integrated both the software and gateware versions of the DEC-BP*n* scheduler with a software switching datapath developed in the Linux kernel, according to the scheme in

Fig. 12. The software version runs on the same computation resources (CPU) of the server, whereas the gateware version runs on an external FPGA card, which acts as a hardware accelerator for the scheduling algorithm. The latter configuration allows a hardware/software co-designed approach for demonstration, where the forwarding datapath runs in software and the DEC-BP*n* runs in hardware. This deployment decision was motivated by the lack of enough resources in the NetFPGA 1G card to fully accommodate the datapath in hardware. Indeed, the number of MC-VOQ queues grows very fast with the number of input and output ports, thus it is convenient to manage the queues directly in the server.

In the following we describe the experimental deployment for the software and gateware versions of a 4×4 DEC-BP*n* scheduler. Our goal is to assess the performance of a full-fledged forwarding system controlled by DEC-BP*n* in terms of resource usage and power consumption.

7.1. Experimental deployment of 4×4 *DEC-BPn*

We have used a server with a 3.06 GHz Intel Core i7 processor and 12 GB of RAM in order to compare the gateware and the software versions of the 4×4 DEC-BP*n* scheduler. The operating system was Fedora 14 32-bit version, with Linux kernel 2.6.36 for an x86 architecture.

To evaluate the gateware version of DEC-BP*n*, we installed a NetFPGA 1G [23] card on the PCI bus of the server. This card features 4 Gigabit-Ethernet ports, tightly coupled with a Xilinx Virtex-II-pro FPGA. Note that the choice of the operating system was dictated by the full compatibility with the NetFPGA card.

The off-the-shelf reference gateware NetFPGA design performs packet forwarding between the 4 Ethernet ports and the PCI-bus. The reference processing datapath is pipelined, 64-bit wide and operates at the Ethernet MAC clock frequency of 125 MHz which allows for 8 Gbit/s processing. The FPGA on-chip memory is a BRAM (block RAM); it is a very scarce resource (few kbytes) and can be directly interfaced in the design. Other than that, as 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 implementations). Typically NetFPGA is used to accelerate novel routing implementations (where many lookups are required), heavy packet processing operations (e.g., encryption) and projects that aim at satisfying real time constraints. The most well-known application is the reference architecture of an OpenFlow switch [13].

NetFPGA features two different communication mechanisms to exchange data with the host computer:

- The network packet I/O interface. It is used to exchange network packets with the host network stack via an appropriate Linux driver. This is a high performance interface that exploits DMA burst transfers and achieves low latency and high bandwidth communication. Its only drawback is that it consumes significant FPGA resources and, as a result, user logic needs to be implemented in the space left by the gateware managing packet I/O. Nevertheless, this interface is the most appropriate for accelerating datapath operations.
- The memory-mapped register interface. It is a higher latency interface that occupies the CPU for the data transfers with significantly less FPGA resource requirements. In typical NetFPGA designs this interface is used to implement control plane operations.

In our case we have heavily modified the NetFPGA framework to implement the scheduler. Due to the required scheduler logic size, we were forced to use memory-mapped register interface for communication. In Sec. 7.3 we will evaluate the latency introduced by this communication approach.

7.2. Mapping gateware DEC-BPn to NetFPGA

We have developed the scheduler Verilog gateware library state machine, which has been integrated in the NetFPGA reference design as a logic block. The scheduler input registers have been connected to software register infrastructure of the NetFPGA. A total of 31 NetFPGA 32-bit software registers were needed. The first 30 registers are used as input for 60 16-bit MC-VOQ weights and 1 register is used as the control register. The output of the scheduler is connected to 1 NetFPGA 32-bit hardware register to accommodate all the 8 4-bit registers described in Sec. 6.3.1

The Xilinx synthesis tools utilized 16,265 reconfigurable logic units (also known as slices) to map the gateware scheduler to reconfigurable resources of the particular Virtex-II Pro FPGA. This equals to 68% of the total available reconfigurable resources on this particular platform (23,616 slices) and, as a result, the standard Ethernet forwarding datapath had to be removed from the NetFPGA. This eliminated the possibility to use Ethernet packets for I/O communication between hardware scheduler and software datapath, which would provide much faster I/O performance than the register interface, as we explained earlier in this section. Nevertheless, a slightly larger FPGA can facilitate both the datapath and the DEC-BP*n* gateware. The deployment of the design on this platform is clocked at 72.9 MHz, which is an acceptable result if we consider the available reconfigurable resources and the design size requirements. Notably, the Xilinx synthesis tools required 3 hours on a high-end server to place and route the design. They were configured to optimize for space rather than clock speed.

In Fig. 13 the deployment of the gateware DEC-BP*n* is depicted. In order to drive the gateware DEC-BP*n* in this experimental deployment, we coupled it with the software datapath which was also used for the evaluation of the software version of the scheduler. The gateware runs on the NetFPGA in the figure and all the software modules run on Intel Core i7 processor. The integration of software and gateware requires to add NetFPGA driver support to Click modular router software and use the NetFPGA register API to push MC-VOQ lengths into the input registers and to read back the service matrix from the output registers.

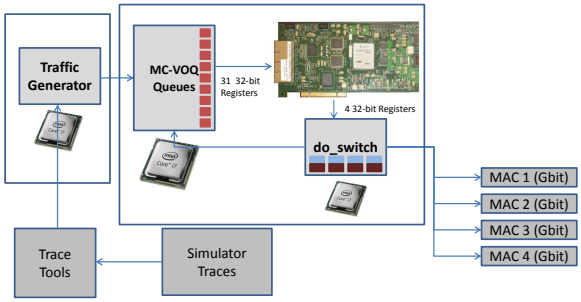

Figure 13: A hardware/software co-design schematic of DEC-BPn implemented on NetFPGA attached to the PCI bus of an Intel Core i7 motherboard

Table 4: Execution time of the software version on Intel Core i7 platform.

Task	BP iterations (n) 0	1	2
DEC-BPn execution	10μs	13.5μs	17μs
Datapath execution	1μs	1μs	1μs
Timeslot duration	11μs	14.5μs	18μs

7.3. Experimental performance evaluation

In these experiments we used the same uniform and concentrated traffic scenarios defined in Sec. 5. To validate both software and hardware implementations, we have compared the achieved throughput with the one obtained with the simulator and we have verified the exact functional equivalence among the different versions (software, gateware and simulator) of the scheduler. The validation has been obtained by feeding exactly the same sequence of packets generated during the simulation.

Referring to Fig. 11, we group the implementation steps into two main tasks: datapath execution (comprising the traffic generation and the execution of the scheduling decision) and scheduler execution (comprising the max-pressure weight computation and the BP iterations). All measurements of the execution time in software have been acquired using the Linux `gettimeofday` time system calls. These calls are implemented using the latest Linux kernel support available and use the x86 architecture TSC (Timestamp Counter) obtained from the same processing core to achieve μs accuracy. Conversely, the execution times in hardware have been measured by the number of clock cycles in the NetFPGA and are therefore very accurate (around tens of ns).

Table 4 shows the experimental execution times for the software version of the scheduler. As expected, the timeslot duration is affected by the number n of BP iterations. The results prove that the scheduler execution is by far the most resource demanding task, occupying almost all of the timeslot duration. The datapath execution is almost negligible, and this is achieved thanks to the fact that the movements of the packets across the queues have been implemented by moving pointers, instead of the actual data.

Table 5 reports the execution time for the gateware version of the scheduler. This version includes the additional step of exchanging data via the register interface: recall that the datapath execution runs in software on Intel Core i7 processor, which at each timeslot pushes queue lengths to the NetFPGA over the PCI bus and retrieves the results. Register I/O is slow compared to the rest of the tasks and has a significant impact on timeslot duration. Notably, NetFPGA computes DEC-BPn around 2.4 times faster than the software version. Nevertheless, the overall duration of the timeslot for the gateware version is much worse due to the delay introduced by the input and output registers. Note that NetFPGA data exchange could be significantly improved if one uses the packet I/O interface instead of the registers. As we have explained in Sec. 6, such an interface has been removed to leave enough resources for the scheduler. Furthermore, improving the hardware interface speed is out of the scope of our proof-of-concept hardware implementation.

Table 5: Execution time of the gateware version of the scheduler running on NetFPGA hosted on Intel Core i7 platform.

Task	BP iterations (n) 0	1	2
Push data to input registers	23μs	23μs	23μs
DEC-BPn execution	3.77μs	5.44μs	7.12μs
Get data from output registers	13μs	13μs	13μs
Datapath execution	1μs	1μs	1μs
Timeslot duration	40.77μs	42.44μs	44.12μs

When comparing the software and the gateware version,it should be noted that the actual number of clock cycles is completely different, due to the different clock (3.06 GHz for Intel Core i7 processor and 72.9 MHz for NetFPGA). Actually, DEC-BP2 scheduler execution requires around 52,000 cycles on the $x86$ processor, whereas the gateware version state machine needs 515 cycles to carry out the same computations. This implies a huge potential performance gain due to an implementation of the same gateware logic in a dedicated ASIC.

7.4. Power consumption

When comparing the software and gateware version, it is worth investigating the power consumption. The adopted Intel Core i7 processor has a Thermal Design Power (TDP) of 130 W, which is the theoretical maximum that a cooling system is required to dissipate with all internal cores operating at full speed. Instead, under full load, the NetFPGA platform requires around 10 W with all Ethernet ports connected [18]. It was not possible to measure the specific power contribution due to the execution of the two versions of the scheduler, because the server platform, common to both versions, is equipped with 2 mechanical disks and many other peripherals that cause a wide range of fluctuation (up to 5 W at idle). Thus, we used a different, low-power CPU platform to run both versions.

We adopted the power measurement system presented in [8], which monitors, at 63 kHz sampling rate, the power consumption on all the subsystems of an Intel Atom D525 ultra-low power platform. Note that Intel Atom is targeted at embedded computers and is typically expected to run on batteries. The evaluation platform comprises now the Intel Atom D525 platform (with a TDP of 13 W), an ultra low power SSD disk and the NetFPGA. The power supply was modified to use the Nitos EMF unit [8]. We measured the power of the platform for hours

Table 6: Power consumption on D525 Intel Atom Platform and NetFPGA 1G

	Scheduler contribution	
Idle State	NetFPGA	Atom processor
23.8 W	2.3 W	3.6 W

when idling and then we repeated many times the execution of the DEC-BP*n* scheduler. The results are presented in Table 6. In order to be fair we also measured execution time of both versions on this platform to check if they have the same forwarding performance. Intel Atom achieves a timeslot duration of 38μs executing the software version, while 52μs executing the gateware version of DEC-BP2 on NetFPGA. Furthermore, the software version is 36% faster at forwarding than NetFPGA but the latter is also 36% less power hungry. Thus, the two versions offer a different performance power tradeoff. We remark that the performance of the considered NetFPGA version is a worst case, since its execution has been severely delayed by the register I/O interface.

8. Conclusion

We have proposed a new scheduling algorithm, denoted as DEC-BP*n*, aimed at approximating the optimal policy for the transmission of multicast packets in IQ switches. Our algorithm has two main advantages. First, the proposed message-passing approach is amenable to an efficient parallel hardware implementation. Second, we have shown that it outperforms other greedy approaches, even when the number of iterations is very small. These encouraging findings allow us to conclude that our approach provides a very convenient tradeoff between implementation complexity and performance. We have also presented both hardware and software implementations of DEC-BP*n*, and integrated them in the standard datapath of a software switch. This allowed us to evaluate the required area logic, the execution time and the power consumption.

References

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

[2] Avramov, L., Portolani, M., 2014. The Policy Driven Data Center with ACI: Architecture, Concepts, and Methodology. Cisco Press.

[3] Chao, H., Liu, B., 2007. High performance switches and routers. Wiley-IEEE Press.

[4] Chen, K., Hu, C., Zhang, X., Zheng, K., Chen, Y., Vasilakos, A., July 2011. Survey on routing in data centers: insights and future directions. IEEE Network 25 (4), 6–10.

[5] Dally, W., Towles, B., 2001. Route packets, not wires: on-chip interconnection networks. In: Design Automation Conference, 2001. Proceedings. pp. 684–689.

[6] Giaccone, P., Pretti, M., June 2013. A belief-propagation approach for multicast scheduling in input-queued switches. In: IEEE ICC. pp. 1403–1408.

[7] Gupta, P., McKeown, N., Jan 1999. Designing and implementing a fast crossbar scheduler. IEEE Micro 19 (1), 20–28.

[8] Keranidis, S., Kazdaridis, G., Passas, V., Korakis, T., Koutsopoulos, I., Tassiulas, L., 2013. Online energy consumption monitoring of wireless testbed infrastructure through the NITOS EMF framework. In: ACM WiNTECH. pp. 89–90.

[9] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M. F., Aug. 2000. The Click modular router. ACM Transations of Computing Systems 18 (3).

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

[11] McKeown, N., Apr 1999. The iSLIP scheduling algorithm for input-queued switches. IEEE/ACM Transactions on Networking 7 (2), 188–201.

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

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

[14] Passas, G., Katevenis, M., Pnevmatikatos, D., Nov 2015. The combined input-output queued crossbar architecture for high-radix on-chip switches. IEEE Micro 35 (6), 38–47.

[15] Prabhakar, B., McKeown, N., Ahuja, R., 1997. Multicast scheduling for input-queued switches. IEEE JSAC 15 (5), 855–866.

[16] Saad, D., Yeung, C., Rodolakis, G., Syrivelis, D., Koutsopoulos, I., Tassiulas, L., Urbanke, R., Giaccone, P., Leonardi, E., Nov 2014. Physics-inspired methods for networking and communications. IEEE Communications Magazine 52 (11), 144–151.

[17] Sarkar, S., Tassiulas, L., 2002. A framework for routing and congestion control for multicast information flows. IEEE Transactions on Information Theory 48 (10), 2690–2708.

[18] Sivaraman, V., Vishwanath, A., Zhao, Z., Russell, C., 2011. Profiling per-packet and per-byte energy consumption in the NetFPGA Gigabit router. In: IEEE INFOCOM Workshop on Computer Communications. pp. 331–336.

[19] Smiljanic, A., Nov. 2002. Scheduling of multicast traffic in high-capacity packet switches. Communications Magazine, IEEE 40 (11), 72 – 77.

[20] Syrivelis, D., Giaccone, P., Koutsopoulos, I., Pretti, M., Tassiulas, L., March 2014. On emulating hardware/software co-designed control algorithms for packet switches. In: Emulation Tools, Methodology and Techniques (EMUTools).

[21] Tassiulas, L., Ephremides, A., 1992. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. IEEE Transactions on Automatic Control 37 (12), 1936–1948.

[22] Veltri, L., 2001. Maximum throughput in multicast input queued packet switches. In: IEEE ICC. Vol. 7. pp. 2033 –2037 vol.7.

[23] Watson, G., McKeown, N., Casado, M., 2006. NetFPGA: A tool for network research and education. In: 2nd Workshop on Architecture Research using FPGA Platforms (WARFP).