FlowToss: Fast Wait-Free Scheduling of Deterministic Flows in Time Synchronized Networks

Randeep Bhatia*, T.V. Lakshman*, Mustafa F. Özkoç†, and Shivendra Panwar†
*Nokia Bell Labs, USA
{randeep.bhatia, tv.lakshman}@nokia-bell-labs.com
†Department of Electrical and Computer Engineering, New York University, USA
{ozkoc, panwar}@nyu.edu

***Abstract*—Motivated by important industrial automation use cases, such as closed loop motion control and autonomous mobile robots, we study wait-free scheduling of periodic flows with stringent delay and jitter requirements in time sensitive networks. The goal is to assign initial transmission time-slots to periodic flows so that network queuing delays are eliminated or are very small. We make use of Bézout's Identity to develop simple and fast scheduling algorithms for this NP-hard problem. Operating in an online mode, our algorithms can quickly allocate contention free start time-slots to new flows, without changing allocations of already scheduled flows. Our main results are greedy and random scheduling algorithms that can trade speed for solution quality. Our simulations on different network topologies show that these algorithms are computationally efficient and can easily schedule a large number of flows, thus meeting the requirements of many industrial automation use cases.**

***Index Terms*—Time Sensitive Networking; Zero Queuing; Scheduling Algorithms; Industrial Automation**

I. Introduction

With greater control of physical devices, machines, production processes, and supply chain, Industry 4.0 promises to usher in a new era of unprecedented levels of automation and productivity for physical industries and associated infrastructure. It is facilitating increased integration and deployment of connected devices and driving new use cases that require periodicity and very high determinism (see Table I), such as closed-loop industrial monitoring, control, optimization applications, and closed loop motion control.

The industrial flows we consider in this work are periodic and need to be delivered through the network by specified hard deadlines. Specifically, this means that the delays through the network should be small and the jitter should be very close to zero. Low delays are required in many factory situations, such as where commands to immobilize robots need to be sent and acted upon in real-time to ensure the safety of workers. Very low or zero jitter is also required in other common factory use cases such as keeping movements of machines and their parts precisely synchronized, achieving which requires reacting in unison to the commands received from the industrial controller. For this, the network needs to have short paths, with small propagation and processing delays as well as negligibly small queuing delays.

Specialized networks and protocols, such as Fieldbus and Profinet [1], that provide bounded network delays and jitter are used extensively in industrial settings. Recently IETF and IEEE have started to standardize enhancements for deterministic real-time networks. In particular, the IETF standard on Deterministic Networking (DetNet) [2] addresses the delivery of data flows with extremely low packet loss rates and bounded end-to-end latency over IP networks. The IEEE standards for Time-Sensitive Networking (TSN), such as IEEE 802.1Qbv [3], which defines enhancements for the so-called scheduled traffic, addresses lower layer technologies. Specifically, IEEE 802.1Qbv, in conjunction with IEEE 802.1AS [4], deals with clock synchronization of switches and end systems and the injection of frames into the network by host network interface cards (NICs) at precisely defined points in time according to a schedule. Although it defines the basic scheduling mechanisms, the problem of calculating optimized time sensitive schedules for NICs, to achieve bounded end-to-end network delay, is beyond the scope of the standard. This is the problem we address in this work.

We consider a network where the "short" paths for the periodic flows are picked in advance. Our goal is to perform a wait-free scheduling of flows using their respective paths. We assume time is slotted and the host NICs have control over the time-slots at which they can allow flows to start injecting their packets into the network. However, NICs are not allowed to subsequently delay any other packets of a flow. For instance, a host NIC can select the time-slot t, for a flow with period T, to inject its first packet into the network. But then, the flow packets must be injected into the network precisely at time-slots $t, T+t, 2T+t, \ldots$.

The goal of wait-free scheduling is to pick these start time-slots t for the flows, for which there is no "contention" at any switch in the network, thus ensuring zero network queuing delays. The scheduling has to be performed online as the flows arrive. Fig. 2, shows two simple periodic flows, f_1 with period 6ms and f_2 with period 9ms, that share switch r_4 on their common path. f_2 starts at time 2ms. Note that (as shown in

ISBN 978-3-903176-39-3 ©2021 IFIP

TABLE I
INDUSTRIAL AUTOMATION USE CASES [12].

Use Case		Cycle Time (ms)	Payload Size (bytes)	Number of Devices	Service Area
Motion Control	Printing Machine	<2	20	>100	100 m
	Machine Tool	<0.5	50	∼20	3 m
	Packaging Machine	<1	40	∼50	3 m
Mobile Robots	Cooperative Motion Control	1	40-250	100	<1 km^2
Mobile Control Panels with Safety Functions	Assembly Robots or Milling Machines	4-8	40-250	4	10 m
	Mobile Cranes	12	40-250	2	50 m
Process Operation (Process Monitoring)		50	Variable	10,000 devices/km^2	

the top half of the figure), if flow f_1 were to start at time 3ms then its packets will contend with packets of flow f_2 at switch r_4. However, this contention can be avoided (as shown in the bottom half), if it were to start at time 1ms, which is a better time-slot choice for it.

Past work has shown that the wait-free scheduling problem is NP-hard and highly intractable (as hard as job shop scheduling) and hence no efficient optimal algorithm is likely [5]. Most of the past approaches have therefore been on formulating and solving the offline version of the problem via integer linear programming (ILP) [5]–[9]. Motivated by the heuristics of [10] for scheduling of no-wait manufacturing processes, [5] develops a start time assignment heuristic for the problem of minimizing "flow makespan". However, the issues with this approach are that it can only scale up to a few hundred flows, which makes it impractical, and it is also not applicable in an online setting. Nayak *et al.* [8] present algorithms for online (incremental flow) scheduling. They make use of a time division multiple access (TDMA) approach to limit the number of active flows over any path to one for the duration of a base period of the flows. Hellmanns *et al.* [9], propose optimizations to improve execution time and solution quality. The aforementioned past work differ from ours as they assume the same period length for all flows. However, when we do not place any such restriction, the problem size grows out of the reach of today's commercial ILP solvers.

We take a different approach than the ILP approach, instead we make use of the Bézout's Identity [11], a simple modular arithmetic identity, to quickly determine feasible time-slots for assignment even when the flow periods are different. Thus, we create efficient algorithms for contention checking and scheduling. Our main contributions are as follows:

- Simple and online greedy and random combinatorial algorithms that are much faster and can find wait-free schedule for many more flows than previous ILP based algorithms.
- A simple contention checking identity based on greatest common divisor (gcd) of time periods of pairs of flows.
- Extensive simulations on different size topologies that show these algorithms are computationally efficient and can easily schedule a large number of flows, thus meeting the requirements of many industrial automation use cases.

II. SYSTEM MODEL

The high level architecture of the system is illustrated in Fig. 1. It is made up of host nodes that are the sources and sinks of flow traffic. These hosts are connected through a

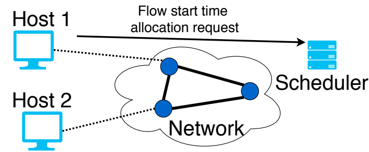

Fig. 1. System Architecture

network of switches. Although our solution is applicable to arbitrary flow paths, we assume flows are routed on shortest paths (based on a delay metric) which are also unique.

We present some system-level assumptions mainly for ease of exposition as well as to serve as a guide for designing practical solutions for industrial applications. For industrial networks, typical link speeds of 10GbE and a switch packet processing time of 100ns are assumed[1]. Note that the latter is approximately the time it takes to transmit 125 bytes at 10GbE speeds. Our choice of 125 byte packets is consistent with the use of small packets in most industrial applications (see Table I). To take into account the typical applications listed in Table I, flow periods are assumed in the range of 1 to 100ms.

We consider a standard switch model where switches forward packets by matching input ports to output ports. As a result, as long as there is at most one packet arrival per port and there is at most one packet departure per port, the packets can be forwarded as they arrive. When this happens, the only delay incurred by the packets within the switch is the forwarding (processing) time. However, anytime the matching conditions are violated, in particular when two packets arrive simultaneously destined to the same outgoing port, at least one packet cannot be forwarded and must wait for its turn in the next round of matching of input and output ports. Such packets must incur an additional queuing delay at the switch.

In our system, there is a central flow scheduler that maintains global knowledge of the network (e.g., topology, propagation, and processing delays) and the set of active flows (e.g., their periods, arrival schedule, allocated start time-slots). Scheduling requests from the hosts come to the scheduler one at a time, and each request is allocated its start time-slot upon arrival. The scheduler makes new allocations without changing the time-slot allocations of already scheduled flows. Hosts start packet transmissions for their flow into the network at precisely their allocated start time-slots. They also inform the scheduler when their flows have been completed.

Let τ denote the size of a time-slot (typically 100ns). We denote the set of flows by F. Each flow $f \in F$ is characterized by a tuple $(T(f), t(f), P(f))$. Here $T(f)$, which denotes its period in time-slots, is sometimes expressed in the more natural units of milliseconds. For flow f, $t(f)$ denotes the absolute time-slot, when it becomes ready to start packet transmission. However, before flow f can start transmitting its packets, it must be granted a start time-slot for its first packet. Ideally this start time-slot $s(f)$ should lie in the interval

[1]Smaller packets can be processed without any conflict within the same time-slot size used for 125 byte packets.

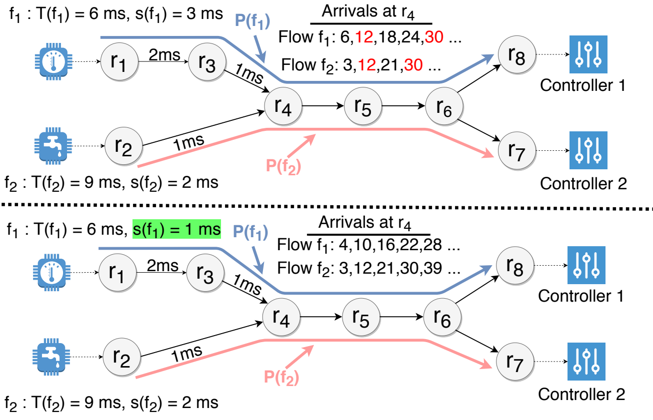

Fig. 2. Contentions at switch r_4 happen at time-slots 12 and 30 for $s(f_1) =$ 3ms (top figure) but are avoided when $s(f_1) = $ 1ms (bottom figure).

$[t(f), t(f) + T(f)]$ so as to not delay flow f by too much. $P(f)$ denotes the network path assigned to flow f. Let the set of switches and links in the network be denoted by R and L respectively with the propagation delay (in time-slot units) of link $l \in L$ denoted by $PD(l)$ and the per packet processing time (in time-slot units) of switch $r \in R$ denoted by $PP(r)$. Let $D_r(f)$ denotes the total propagation and processing delay before a switch $r \in R$ on the path $P(f)$. Then for any k,

$$D_{r_k}(f) = \sum_{i=1}^{k-1} PD(l_i) + \sum_{i=1}^{k-1} PP(r_i), \quad (1)$$

where $l_1, l_2, \ldots$ are the links and $r_1, r_2, \ldots$ are the switches on $P(f)$ and link $l_i = (r_i, r_{i+1})$ goes from switch r_i to r_{i+1}.

Note that, if there is a link $l = (r_a, r_b)$ that is shared among paths $P(f_1)$ and $P(f_2)$ of different flows f_1 and f_2, then a queue may be needed to handle packet "contentions" at switch r_a. These contentions can happen, for instance, if packets of flows f_1 and f_2 arrive in the same time-slot at ingress switch r_a of link l (as they must both depart through the same outgoing port of switch r_a corresponding to link l). This in turn may depend on the choice of start time-slots $s(f_1)$ and $s(f_2)$ for the flows f_1 and f_2. For instance, for periodic flows f_1 and f_2, contention at switch r_a is avoided if and only if start time-slots $s(f_1)$ and $s(f_2)$ are selected, for which:

$$s(f_1)+D_{r_a}(f_1)+i_1 \cdot T(f_1) \neq s(f_2)+D_{r_a}(f_2)+i_2 \cdot T(f_2), \quad (2)$$

for all choices of non-negative integers i_1 and i_2. Our goal is to assign start time-slots to flows to avoid such contentions in order to eliminate the need for queuing at the switches. Fig. 2, which shows two simple periodic flows, illustrates how by assigning the right set of start time-slots to different flows, contentions among them, and hence network queuing, can be avoided.

III. ALGORITHMS

In general, the wait-free scheduling problem is NP-hard as it can be reduced to the NP-hard no-wait job scheduling problem [5]. We, therefore, design fast heuristic algorithms focusing on generating a wait-free schedule for this NP-hard problem. Unlike [5] and [8] where the solution optimality is evaluated by the overall make-span of the flows and by link usage, respectively, our notion of an optimal solution is when the generated schedule results in a wait-free operation. Our main results are a greedy and a random scheduling algorithm. Both these algorithms can operate in an online mode, to quickly assign start time-slots to flows as they arrive. The randomized algorithm can be used in situations where somewhat faster scheduling is required, possibly even at the expense of slightly lower solution quality (meaning the solution is not 100% but is almost wait-free). For instance, it may be used to quickly re-allocate start time-slots to flows, whose start time-slot allocations no longer work, possibly after re-routing due to network failures.

Checking for contentions among flows can be computationally quite expensive. However, we make use of flow periodicity to design a *fast* contention checking algorithm based on the greatest common divisor (gcd) of time periods of pairs of flows. We obtain further efficiencies by restricting contention checking to a single switch on only one of the links shared among flow paths. In addition, our algorithm carries out a majority of its computation in advance, thereby speeding the contention checking during flow scheduling. Finally, as we show below, contention checking can be parallelized, with almost a linear speedup.

We start with some basic notation. Recall that the goal is to assign a start time-slot to flow f from the interval $[t(f), t(f)+ T(f)]$. We denote these candidate set of time-slots by $C(f)$. Thus, there are $|C(f)| = T(f)$ candidate time-slots for f. The flows in F are arranged in their order of arrival $f_1, f_2, \ldots$. We denote by $F_i, i \geq 0$ the first i flows in F, with $F_0 = \emptyset$. In the following, lcm denotes least common multiplier and gcd denotes greatest common divisor. Let indicator variable δ_{f_i,f_j} be 1, if flows f_i and f_j share a link on their paths, and be 0 otherwise. We denote by $\delta(F_{i-1})$ the set of flows $f_j \in F_{i-1}$ for which $\delta_{f_i,f_j} = 1$. These are the only set of flows that need to be evaluated for contention when scheduling flow f_i as they have a link in common with it. Note that the sets $\delta(F_{i-1})$ may be a much smaller subset of F_{i-1}, as only a small portion of the flows in F_{i-1} may share a common link with flow f_i.

A. Greedy algorithm

The flows are presented to the greedy algorithm (Algorithm 1) in their order of arrival $f_1, f_2, \ldots$. When processing a flow f_i, the greedy algorithm computes the quantity $Q_i(s)$, for each possible start time-slot $s \in C(f_i)$. Here $Q_i(s)$ is the number of flows in $\delta(F_{i-1})$ that f_i will contend with, if it were to start at time-slot $s(f_i) = s$. Note that it is also a measure of the queue sizes required at switches on the path $P(f_i)$ for $s(f_i) = s$. The greedy algorithm sets $s(f_i)$ to a time-slot s in $C(f_i)$ for which $Q_i(s)$ is minimized. In Algorithm 1, these time-slots are in the set Q_i (the argmin operator there returns all time-slots s for which $Q_i(s)$ is minimized). In the case of ties, $s(f_i)$ is drawn randomly from the set Q_i. The computation of the $Q_i(s)$ values is the most challenging part of

the algorithm. Later (Section III-C) we present a fast algorithm for computing the $Q_i(s)$ values.

Algorithm 1 Greedy start time-slot assignment

1: **for** $i \leftarrow 1$ to $|F|$ **do**
2: $Q_i \leftarrow \arg\min_{s \in C(f_i)} Q_i(s)$
3: $s(f_i) \leftarrow$ any random time-slot in Q_i
4: **end for**
5: $A \leftarrow$ start time-slot assignments $s(f_1), s(f_2), \ldots$
6: **return** A

B. Random algorithm

Just as is the case for the greedy algorithm, the flows are presented to the random algorithm (Algorithm 2) in their order of arrival $f_1, f_2, \ldots$. However, when processing a flow f_i the random algorithm only evaluates $Q_i(s)$ for a randomly sampled subset of time-slots $RC_i \subseteq C(f_i)$. It then assigns $s(f_i)$ to be that time-slot s in RC_i for which $Q_i(s)$ is minimized (ties broken randomly). The time it takes to pick the best time-slot s in RC_i is proportional to the size n of the set RC_i, which is passed in as an argument to Algorithm 2.

Note that, in the random algorithm, there can be a tradeoff between speed and quality. That is, for smaller values of n, even though the random algorithm may run faster, the solution it finds, may not always be wait-free. However we find that even for $n = 25$, the random algorithm almost always delivers a wait-free solution for up to 100000 flows, thus making 25 a good choice for n in practice.

Algorithm 2 Random start time-slot assignment

1: **procedure** RANDOM(n)
2: **for** $i \leftarrow 1$ to $|F|$ **do**
3: $RC_i \leftarrow n$ random slots from $C(f_i)$
4: $Q_i \leftarrow \arg\min_{s \in RC_i} Q_i(s)$
5: $s(f_i) \leftarrow$ any random time-slot in Q_i
6: **end for**
7: $A \leftarrow$ start time-slot assignments $s(f_1), s(f_2), \ldots$
8: **return** A
9: **end procedure**

C. Flow pair contention checking algorithm

Algorithms 1 and 2 require $Q_i(s)$ values for flow f_i, for all start time-slots $s \in C(f_i)$. When flow f_i is assigned start time-slot s, we can determine whether the packets from flows f_i and any flow $f_j \in \delta(F_{i-1})$ can arrive simultaneously at any of the ingress switches r on the common links between paths $P(f_i)$ and $P(f_j)$, by checking for it in all the time-slots in one lcm of their periods. However, computationally this can be very expensive. For instance, consider checking 10000 flows $f_j \in \delta(F_{i-1})$ against 100000 time-slots s in $C(f_i)$ (assuming $T(f_i) = 10$ms flow period and time-slots of size $\tau = 100$ns). This checking has to be done for each of the $lcm(T(f_i), T(f_j)) \geq 100000$ time-slots t (as that is the number of time-slots in the common period of the two flows). Thus, even if flow paths share only one link on the average, the number of computations required makes this "brute force" approach highly intractable.

We design an efficient algorithm by leveraging some properties of the solution which are established before. First, for flows f_i and flow $f_j \in \delta(F_{i-1})$, we only check for contention at the ingress switch r on the first common link among their paths. This follows from Theorem 1 whose proof is omitted.

Theorem 1. *Let $e_1, e_2, \ldots$ be the set of common links on the paths $P(f_1)$ and $P(f_2)$ for the flows f_1 and f_2. These links $e_1, e_2, \ldots$ form a single connected path SP (merge only once) [13]. Furthermore, flows f_1 and f_2 contend if and only if they contend on the ingress switch of the first link on their shared sub-path, where their paths merge.*

Second, we use the fact, derived using Bézout's Identity [11], (proof omitted) that there is at least one contention at switch r, among packets of flow f_i with transmission time-slots $t_1, t_1 + T(f_i), t_1 + 2T(f_i) \ldots$ and packets of flow f_j with transmission time-slots $t_2, t_2 + T(f_j), t_2 + 2T(f_j) \ldots$, if and only if the following delay differential equality is satisfied:

$$t_1 + D_r(f_i) - (t_2 + D_r(f_j)) \equiv 0 \mod gcd(T(f_i), T(f_j)). \tag{3}$$

In the rest of this section we use the terms GCD and $gcd(T(f_i), T(f_j))$ interchangeably. The algorithm works by keeping an array for the $Q_i(s)$ values, corresponding to the time-slots $C(f_i)$, initialized to all zeros. It then picks $t = GCD - rem$, where rem is the non-negative remainder:

$$rem = (D_r(f_i) - (s(f_j) + D_r(f_j))) \mod GCD. \tag{4}$$

Note that, (3) is satisfied for $t_1 = t$ and $t_2 = s(f_j)$. Thus, with start time-slot t for flow f_i, at least one of its packets that arrive at time $t + k_1 T(f_i)$, for integers k_1, will contend with at least one of the packets for flow f_j at switch r (the ones that arrive at time $s(f_j) + k_2 T(f_j)$ for integers k_2). In particular, this contention will happen with any of the starting time-slots $t, t+GCD, t+2GCD, \ldots$ for flow f_i. The algorithm therefore increments the positions in the array for the $Q_i(s)$ values corresponding to these time-slots. By doing this, the algorithm is able to identify all starting time-slots for flow f_i, for which it will contend with flow f_j. It repeats this process for all flows $f_j \in \delta(F_{i-1})$.

Algorithm 3 Contention detection

1: **procedure** COMPUTECONTENTION(i)
2: $Q_i \leftarrow$ array of size $T(f_i)$ initialized to zeros
3: **for** $f_j \in \delta(F_{i-1})$ **do**
4: $r \leftarrow$ first_ingress_switch ($P(f_i), P(f_j)$)
5: $rem \leftarrow D_r(f_i) - (s(f_j) + D_r(f_j)) \mod GCD$
6: $t \leftarrow GCD - rem$
7: $K \leftarrow \left\lfloor \frac{T(f_i) - t}{GCD} \right\rfloor$
8: **for** $k \in 0, 1, \ldots K$ **do**
9: $Q_i(t + kGCD) \leftarrow Q_i(t + kGCD) + 1$
10: **end for**
11: **end for**
12: **return** Q_i
13: **end procedure**

Running time of Algorithm 3: The time it takes to compute contending starting time-slots $t, t + GCD, t + 2GCD, \ldots$ for

a given pair (t_a, t_b) is $T(f_i)/GCD$. This is done for every flow $f_j \in \delta(F_{i-1})$. The total time for this is bounded by:

$$\sum_{f_j \in \delta(F_{i-1})} \frac{T(f_i)}{gcd(T(f_i), T(f_j))}. \quad (5)$$

Note that, as the computation for different flows $f_j \in \delta(F_{i-1})$ can be carried out independently, linear speedup is possible with the use of multiple processors. Thus the overall time to compute $Q_i(s)$ for Algorithms 1 and 2, with P processors, is upper bounded by:

$$O\left(\frac{1}{P} \sum_{f_j \in \delta(F_{i-1})} \frac{T(f_i)}{gcd(T(f_i), T(f_j))}\right). \quad (6)$$

Furthermore, note the quantities $\frac{T(f_i)}{gcd(T(f_i),T(f_j))}$ are typically small in practice. For instance, for $T(f_i) = 5\text{ms}$ or 50000 time-slots and $T(f_j) = 3\text{ms}$ or 30000 time-slots $\frac{T(f_i)}{gcd(T(f_i),T(f_j))} = 5$. Thus the running time of the Algorithm 3 is mainly $O(\frac{1}{P}|\delta(F_{i-1})|)$.

Running time of Algorithm 1: For scheduling flow f_i, in addition to computing the $Q_i(s)$ values, the greedy algorithm needs to find the time-slots s for which $Q_i(s)$ is minimum. This takes an additional $T(f_i)$ time serially, as that is the number of potential start time-slots for flow f_i. With P processors a speedup of almost P is possible for computing the minimum. Thus the running time of the greedy algorithm, for the i-th flow, with P processors, is upper bounded by:

$$O\left(\frac{T(f_i)}{P} + \frac{1}{P} \sum_{f_j \in \delta(F_{i-1})} \frac{T(f_i)}{gcd(T(f_i), T(f_j))}\right). \quad (7)$$

Running time of Algorithm 2: Just like the greedy algorithm, the random algorithm needs to find the time-slots s for which $Q_i(s)$ is minimum. However for the random algorithm this step takes only n/P time with P processors, which is a small constant (as n may only be 25). Thus, the running time of the random algorithm for the i-th flow, with P processors, is upper bounded by:

$$O\left(\frac{n}{P} + \frac{1}{P} \sum_{f_j \in \delta(F_{i-1})} \frac{T(f_i)}{gcd(T(f_i), T(f_j))}\right). \quad (8)$$

D. Other Considerations

Incremental updates: Note that both our algorithms naturally support incremental addition or deletion of flows. For both the algorithms, only the sets F_i need to be updated, either to add a new flow or to delete an existing flow. These operations can be supported in constant time with a data structure for fast insertion and deletion.

Handling Contention: In some cases, especially when the number of flows is very large (for high flow loads), the algorithms may fail to find a wait-free solution. In this case, one option is to increase the switch capacity reserved for TSN flows. The other option is to use an admission control policy. One possibility is to outright reject any flows that cannot be admitted without a contention. The other option is to reject flows, only if admitting them would make the number of flow contentions exceed a threshold. Although the schedule may not be wait-free, the worst case jitter may still be kept in check because the maximum contention experienced by admitted flows is limited. Also, in many industrial applications, there is tolerance against packet losses, particularly if the losses are not bursty [14]. For such applications, packets may be selectively dropped, which may help eliminate or limit contention. For instance, for two contending flows that have the same period, by alternately dropping their packets, their contentions can be completely eliminated, while also avoiding contiguous packet losses. In general though, much more sophisticated packet dropping schedules may be needed to avoid any unnecessary packet drops. Finally, other options such as sending some flow packets on alternate paths, may also be employed for avoiding contention [14].

IV. Performance Evaluation

We evaluate our algorithms in three different networks *A*, *B*, and *C*, where the former one is undirected and the latter two are directed. There are 59 nodes connected by 96 links in *A*, whereas there are 81 nodes and 296 links in *B* and finally, 33 nodes and 100 links in *C*. The average path-lengths of the networks *A*, *B*, and *C* are 4.46, 5.01, and 4.75, respectively.

We evaluate and compare the performances of the random and the greedy algorithms. We use three different sampling sizes, $n = \{5, 25, 100\}$, for the random algorithm in order to illustrate the trade-off between the solution run-time and the solution quality. We focus on three different performance metrics, namely *scheduling duration* of an arriving flow, *expected number of conflicts* that will be caused by an arriving flow, *expected number of maximum conflict* that will be experienced by any flow in the system. We run 100 randomly generated experiments. For every experiment, we generate 100000 flows each with a random period between 1 ms to 100 ms, a random source node, and a random destination node that is different from the source. Our choices are guided by the number and type of flows expected in an industrial network.

We assume the network controller completed the pre-computation (shortest path routing, link propagation delays, and processing delays) before the arrival of the first flow. The total pre-computation times for the networks A, B, and C are 35.6, 137.3, and 3.6 seconds, respectively. We schedule the incoming flows based on their order of arrival and record the run-time of the scheduling algorithm. The reported scheduling duration in Fig. 3(a) is the smoothed plot of the average scheduling duration of the i-th flow over the 100 experiments. Fig. 3(b) illustrates that the average number of contentions experienced by the i-th flow over the 100 experiments. For a fair comparison, we evaluate all our algorithms on the same set of ordered flows, within an experiment. As seen from Fig. 3(a) and Fig. 3(b), the greedy algorithm has the longest run-time, but always finds a contention free solution. The run-time of the random algorithm has a proportional relationship with the size n of the random set RC_i. However, the improved speed

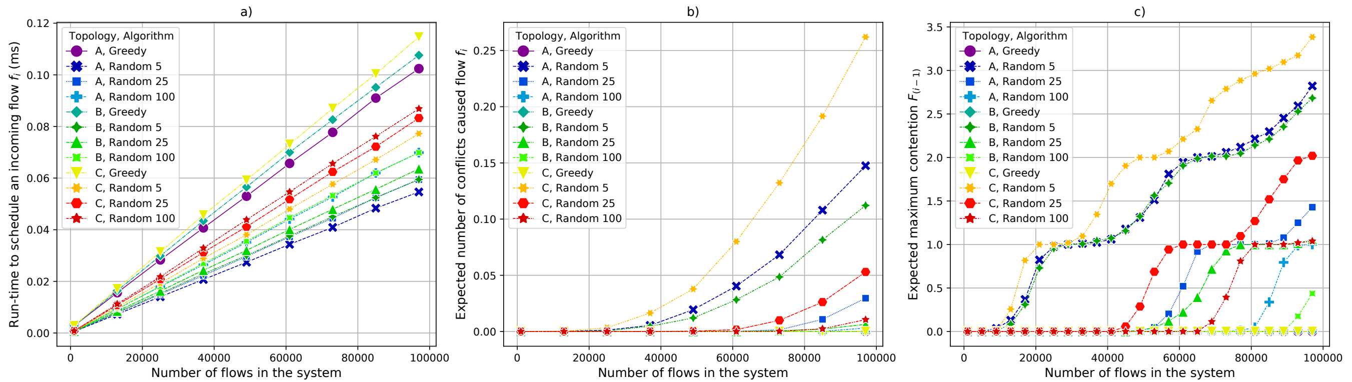

Fig. 3. a) The time to schedule an incoming flow increases linearly with the number of flows in the system. The greedy algorithm has the longest run-time. As the sampling size of the random algorithm decreased the run-time is reduced. b) The expected number of conflicts caused by an incoming flow is the lowest for the greedy algorithm and the highest for the random 5 algorithm. As the sampling size of the random algorithm decreased, the run-time is reduced. This in turn increases the average number of conflicts caused by the incoming flows. c) The expected number of maximum contentions experienced by the flows in the system. The random 5 algorithm can schedule 100000 flows with $3-4$ maximum conflicts per flow.

of the random algorithm comes at the expense of marginally higher contentions. Thus, when small amounts of jitter can be tolerated or handled with previously mentioned methods i.e., admission control, additional switch capacity, or packet drops (see Section III-D), random algorithms may be used for faster scheduling.

Random algorithms limit the search space significantly and may miss out on a wait-free schedule opportunity, but in practice, they can find wait-free schedules even for a large number of flows: up to 25000 flows for $n = 5$ and 80000 flows for $n = 100$. To further illustrate the solution quality of the algorithms, we present the expected number of maximum conflicts experienced by a flow in the system in Fig. 3(c). The greedy algorithm can schedule all 100000 flows without any conflicts. Random 5 algorithm can schedule the 100000 flows with only a maximum of 3 or 4 conflicts per flow. Note that the reported number of conflicts is only an upper bound on the actual conflicts within individual switches (the number of flow arrivals on any common time-slot at any switch). The actual contention within the switches may be much lower than this upper bound [2].

V. Conclusion

We designed and evaluated fast greedy and random algorithms for wait-free scheduling of periodic flows in TSN. For this, we introduced a contention checking algorithm that makes novel use of Bézout's identity combined with fast processing of network delays and already scheduled flows to quickly eliminate possible conflicting start time-slots. Compared to prior ILP based algorithms, that are only able to handle a few hundred flows, our greedy algorithm easily scales to hundreds of thousands of flows. Our random algorithm offers additional tuning for the solution quality vs. run-time trade-off. In our simulations for up to 100000 flows, the greedy algorithm always finds a wait-free solution and the faster random algorithm delivers almost wait-free schedules.

[2]For example, even if a flow conflicts with two other flows, that does not necessarily mean all three flows will arrive at a switch at a common time-slot. Consider flows f_k, f_l, f_m with periods (in time-slot units) $T(f_k) = 15$, $T(f_l) = 6$, $T(f_m) = 10$ and delays $D_r(f_k) = 10$, $D_r(f_l) = 1$, $D_r(f_m) = 0$, respectively. Flows f_k and f_l conflict at $t = 25$, and flows f_k and f_m conflict at $t = 40$ but flows f_l and f_m do not conflict.

Acknowledgments

This work was supported in part by NYU Wireless, an Ernst Weber Fellowship, and by the NY State Center for Advanced Technology in Telecommunications (CATT).

References

[1] E. Tovar and F. Vasques, "Real-time Fieldbus communications using Profibus networks," *IEEE Trans. Ind. Electron.*, vol. 46, no. 6, pp. 1241–1251, 1999.

[2] N. Finn, P. Thubert, B. Varga, and J. Farkas, "Deterministic networking architecture," Internet Requests for Comments, RFC Editor, RFC 8655, 2019. [Online]. Available: https://datatracker.ietf.org/doc/rfc8655

[3] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," *IEEE Std 802.1Qbv-2015*, pp. 1–57, 2016.

[4] "IEEE standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications," *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, pp. 1–421, 2020.

[5] F. Dürr and N. G. Nayak, "No-wait packet scheduling for IEEE time-sensitive networks (TSN)," in *Proc. ACM Int. Conf. Real Time Netw. Syst.*, Brest, France, 2016, p. 203–212.

[6] J. Falk, F. Dürr, and K. Rothermel, "Exploring practical limitations of joint routing and scheduling for TSN with ILP," in *Proc. IEEE 24th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2018, pp. 136–146.

[7] N. G. Nayak, F. Dürr, and K. Rothermel, "Time-sensitive software-defined network (TSSDN) for real-time applications," in *Proc. ACM Int. Conf. Real Time Netw. Syst.* Brest, France: ACM, 2016, p. 193–202.

[8] N. G. Nayak, F. Dürr, and K. Rothermel, "Incremental flow scheduling and routing in time-sensitive software-defined networks," *IEEE Trans. Ind. Informat.*, vol. 14, no. 5, pp. 2066–2075, 2018.

[9] D. Hellmanns, L. Haug, M. Hildebrand, F. Dürr, S. Kehrer, and R. Hummen, "How to optimize joint routing and scheduling models for TSN using integer linear programming," in *Proc. ACM Int. Conf. Real Time Netw. Syst.*, Nantes, France, 2021.

[10] R. Macchiaroli, S. Mole, and S. Riemma, "Modelling and optimization of industrial manufacturing processes subject to no-wait constraints," *Int. J. Prod. Res.*, vol. 37, pp. 2585–2607, 11 2010.

[11] G. A. Jones and J. M. Jones, *Elementary number theory*. New York: Springer-Verlag, 1998.

[12] 5G ACIA, "5G for connected industries and automation," 5G-ACIA, Tech. Rep., February 2019. [Online]. Available: https://bit.ly/3vfaBKp

[13] G. Bodwin, "On the structure of unique shortest paths in graphs," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2019, pp. 2071–2089.

[14] J. Gebert and A. Wich, "Alternating transmission of packets in dual connectivity for periodic deterministic communication utilising survival time," in *29th European Conf. on Netw. and Commun. (ECNC)*, 2020.