Iterative Design Space Exploration for Networks Requiring Performance Guarantees

Bruno Cattelan*†, Steffen Bondorf†
*Universidade Federal do Rio Grande do Sul, Brazil
†Distributed Computer Systems (DISCO) Lab, University of Kaiserslautern, Germany

Abstract—**Networks are embedded many system nowadays. They route control messages, sensory data, etc. and thus their functionality is integral to the enclosing system. In case of safety-critical systems, additional non-functional demands need to be fulfilled. A prime example are systems from the avionics domain where worst-case message delay needs to be bounded. The Deterministic Network Calculus (DNC) analysis is able to provide worst-case bounds on message delay. Accuracy of these bounds has been steadily improved to counteract costly over-provisioning. In this paper, we first provide detailed insight on the aspect preceding the DNC analysis: the network modeling step. It already contributes pessimism resulting in inaccuracy of later results. Second, we contribute multiple algorithms for iterative design space exploration within the limits of modeling that employ the most accurate DNC analysis. In numerical experiments, we show their effectiveness to reduce an avionics network's tail latency by optimized resources usage and the required effort.**

I. Introduction

Networks are critical to fulfill communication tasks in almost every modern distributed system. Among these are routing of control messages to various actuators, reporting of sensory data, transmission of audio and video streams etc. For many of the tasks, the network's functional reliability is integral to the enclosing system. In case of safety-critical systems, additional non-functional demands need to be fulfilled as well – even if the network consists mostly of commercial off-the-shelf hardware and is shared by many tasks. In fact, without formally verified provision of performance guarantees, these systems often cannot obtain certification. As a result they are not permitted to be operated in public spaces. A prime example is the avionics domain where the above applies to networks embedded into modern aircraft.

Verification of non-functional performance guarantees can be provided by different mathematical tools. The Deterministic Network Calculus (DNC) analysis is able to provide worst-case bounds on the backlog that can build up in a server's queue and the delay a data flow experiences from its source to its destination. Accuracy of DNC bounds has been steadily improved to counteract costly over-provisioning [1], [2], [3], [4], [5], [6] and it has already been applied to certify the AFDX (Avionics Full Duplex Switched Ethernet) data network embedded into the Airbus A380 [7], [8].

The main complications for a DNC analysis arise from the network size to be analyzed and the fact that networks are a shared resource. As the network acts as a shared resource that routes all flows, their amount as well as their entanglement are decisive. Both impact the DNC bound accuracy as well the computational effort of an analysis. Scalability of the analysis w.r.t. this cost metric has been in the focus of recent improvements [9], [6] as well.

An aspect often neglected by this work is that the DNC analysis expects a complete network model to operate on. I.e., DNC is currently targeted towards analyzing a finished design. Only the derived bounds can be checked against requirements: the server backlog bounds should not cause data to be dropped and flow delay bounds should be within deadlines required to guaranteed reliable operation of the enclosing system. Employment of DNC in the design phase is thus only possible in a design space exploration. A potentially vast number of design alternatives is created, each of which is then analyzed with DNC. This procedure is obviously inefficient, yet, restrictions to the design space can be made. In this paper, we consider networks embedded into a larger system, i.e., wiring is subject to physical constraints and cannot be changed easily and data flows' sources and destinations are fixed. Thus, we focus on the exploration of the freely configurable design space. We optimize flow routes to minimize the worst-case end-to-end delay in the network, i.e., we tackle the tail latencies. We show that the complex entanglement of flows within a network results in complex interdependencies such that shortest path routing is not the optimal solution.

This design phase employment of DNC relies on a previously untested assumption: The analysis of large amounts of network is easily possible with DNC. We check this fundamental assumption by two investigations: First, we investigate the interdependency between analysis assumptions as well as restrictions and the modeling step of DNC. We provide comprehensive insight on the general demands of modeling for DNC, showcasing the involved effort and potential causes for later result inaccuracies. This is is detailed on a running example in Section II. Then, we present algorithms for (iterative) design space exploration that aim to reduce the tail latencies in an AFDX data network by a more balanced use of network resources. These algorithms are given in Section III and related work is presented in Section IV. We extended the most comprehensive open-source tool for DNC analyses, the DiscoDNC [10]. This work is the basis for our numerical experiments (Section V) that show the effectiveness and cost of the proposed algorithms that allow for usage of DNC results in the design phase of a network. Section VI concludes the paper.

This work was supported by the Carl Zeiss Foundation.

II. Deterministic Network Calculus: Modeling and Analysis of AFDX Networks

In this Section, we provide an overview over modeling and analysis assumptions imposed by DNC and illustrate it by a running example. Some of the of these assumptions result from DNC theory, others are imposed by current restrictions of DNC tool support. In later evaluations, we will use the most comprehensive open-source DNC tool available, the DiscoDNC [10]. Thus, the following exemplary modeling of a small sample AFDX data network aligns to this tool's capabilities. We will not provide a detailed treatment of DNC itself as this can be found in the literature, e.g., [11], [12], [13], [14], [15], [3], [9], [16].

Networks are usually modeled by graphs $G = (V, E)$ consisting of a set of vertices V and a set of edged E connecting pairs of vertices. Within different abstractions, vertices and edges can have different semantics. For instance, edges can be unidirectional or bidirectional. DNC analyses require a particular model, the so-called *Server Graph*. In the remainder of this Section, we illustrate how to derive an AFDX data network's server graph model for DNC analysis.

A. Network Topology

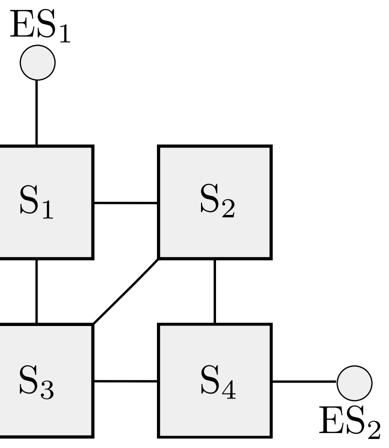
The highest level to describe a network is by its topology that provides information about connections between devices. I.e., vertices are devices and edges are bidirectional connections. The actual network whose topology is modeled gives a more refined interpretation of the this abstract model.

1) AFDX Data Networks: An AFDX data network is instantiated with end-systems that connect to a network core of switches in order to communicate to other end-systems. An AFDX data network as found in the Airbus A380 is composed of more than one hundred end-systems and a dense core of dozens of switches. The creation of network topologies representative for AFDX is depicted in detail in [17] and Figure 1a provides our running example consisting of six devices. End-systems in the periphery of the network are depicted as circles and abbreviated with ES_i whereas switches in the core are depicted as rectangles and abbreviated S_n, $i, n \in \mathbb{N}$. Each ES_i is connected to only one switch, yet each switch S_n may be connected more than one ES_i or other switch S_m, $n \neq m$. Both types of devices queue and transmit data. They are interconnected by 100Mbps full duplex Ethernet links, i.e., bidirectional edges in the graph.

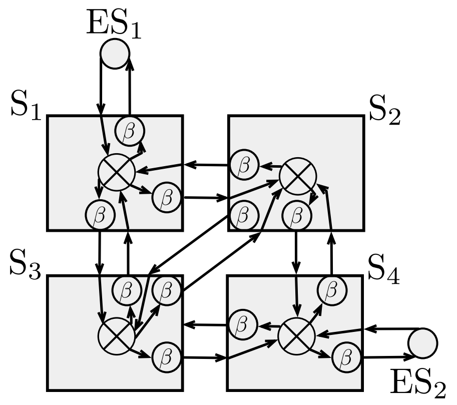
2) DNC Device Graph: In DNC, this abstract model directly derived from the network topology is known as the *Device Graph*. However, to accommodate the later analysis, bidirectional edges are stored as pairs of unidirectional ones. A DNC Device Graph does not hold any information about the devices. It is not sufficient for analysis as the DNC queueing analysis require the forwarding resource offered by devices to be precisely modeled.

B. Resource Contention and Queueing Effects

The complete DNC model required to apply an analysis demands refinement of the device graph. This refinement is



(a) Sample AFDX data network.



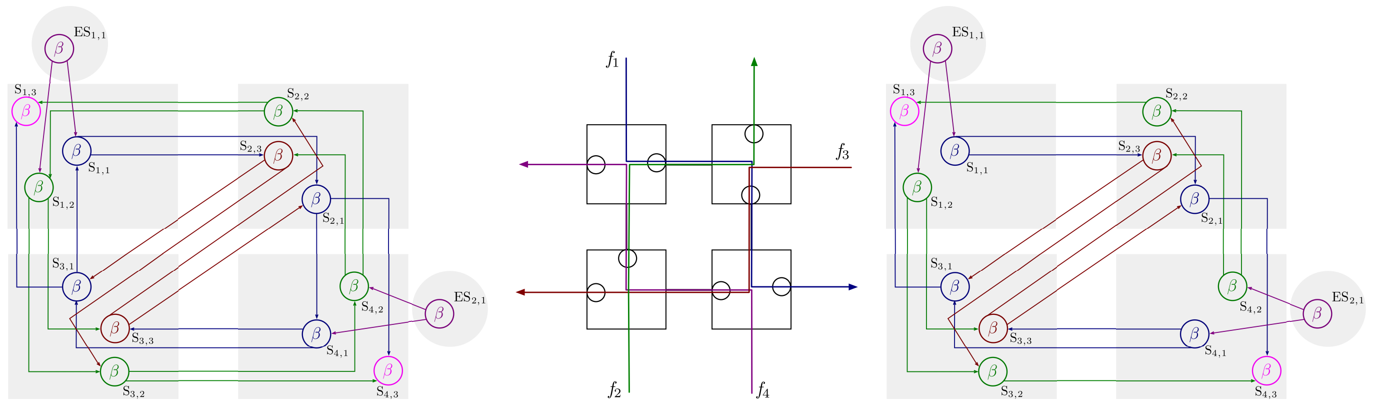
(b) Refined device graph with βs at queueing locations.

Figure 1: Running example of an AFDX data network.

based on the devices employed in the modeled network, e.g., sensor nodes in wireless sensor networks [18] or Ethernet-based devices in an AFDX data network.

1) AFDX Devices: As AFDX is based on the Ethernet standard, its switches employ the following architecture relevant to a queueing analysis. Data enters the device via an input port, then a switching fabric forwards it to an egress port. Input ports are served at line speed and switching fabrics are highly optimized such that contention over the data forwarding resource, i.e., queueing effects, manifest at the egress ports [19]. An intermediate representation depicting this additional information is shown in Figure 1b. Unidirectional links connect devices as well as these devices' sub-components: switching fabrics are depicted as circled Xs, egress servers and their queues are depicted as circled betas β, the DNC service curves. AFDX switches can assign multiple priority classes to traffic at its egress ports. In network calculus' point of view, this is captured by computing the left-over service $\beta^{\text{l.o.}}$ for a class of traffic – the priority classes are virtually separated from each other as it would be done by the scheduler. In our work, we analyze networks with the so-called arbitrary multiplexing assumption[1]. It does not explicitly assign priority classes a priori. Instead of finding a priority setting as in [20],

[1]The DiscoDNC only implements arbitrary multiplexing.



(a) Server graph. (b) Potential cyclic dependencies. (c) Feed-forwardized server graph.

Figure 2: Running example of an AFDX data network, continued.

the analysis dynamically computes the analyzed traffic flow's worst-case left-over service under the maximum influence of all other flows at the egress port. From the analysis' point of view, this is equivalent to a single priority class for all traffic. I.e., our results are valid for all priority assignments.

Last, note that end-systems do not provide forwarding service to other devices. They are either the source or the sink of data communicated via the AFDX network's core. In the role of a source, end-systems generate data at a higher level inside the device and directly push it into their single egress port's queue. As a sink, data will be handed to a higher level upon reception and thus no server is crossed in the end-system. Resource contention at higher levels is usually not modeled or analyzed with DNC. In other words, the focus of the model is the behavior of the system when transporting data rather than the usage of it.

2) DNC Server Graph: The DNC analysis derives the worst-case queueing effects when contending for forwarding resources. It considers these effects at consecutive servers crossed by an analyzed flow as well as this flow's cross-traffic in a compositional manner (compositional feed-forward analysis, compFFA [3], [16]). Thus, it abstracts from the sub-component modeling of the refined device graph shown in Figure 1b. Instead, egress servers and their queues identified in the refined device graph are connected with unidirectional links. This representation is called *Server Graph*. Our running example's server graph is depicted in Figure 2a – it shows that even for small networks the model can become complex as it consists of edges connecting servers. From the point of view of Network Calculus there is no real difference between the end-systems and the switches. Since only the output buffers are modeled, they are all seen as servers in the server graph. Every server's minimum forwarding capabilities are given by it respective service curve β. Modeling an AFDX data network, they correspond to 100Mbps Ethernet connections.

An assumption commonly applied to the analysis of queues is that data in them is multiplexed in a FIFO manner. As mentioned above, our analysis assumes arbitrary multiplexing. This assumption leads to worse results than FIFO multiplexing as it considers more potential interference patterns between flows. These interference patterns can be caused by crossing a switching fabric. When switching data from ingress to egress ports, FIFO multiplexing between flows might not be retained; a mismatch between modeling assumptions and actual behavior can result in invalid worst-case performance bounds. Results from an arbitrary multiplexing analysis are valid independent of the switching fabric behavior.

C. DNC Analysis Restrictions Influencing the Model

1) Cycle-free, Feed-forwardized Server Graph: In general, DNC analysis requires a network that does not create cyclic dependencies between flows. In the AFDX sample network, cyclic dependencies can, however, arise. In Figure 2b, four data flows f_1 to f_4 cross the network. Each flow enters the core from a different end-system connected to a different switch. Each flow then crosses three devices before terminating in a different end-system. With this traffic pattern, each of the egress servers connecting to a different switch queues data of two flows. Analyzing any of these flows involves the analysis of two such servers. It demands to backtrack cross-traffic recursively [21], [3]. This recursion does, however, not terminate.

Preventing cyclic dependencies can be achieved by a thorough selection of paths to be taken by flows or by modification of the server graph such that no combination of flow paths can cause cycles. The latter is known as the feed-forward property. As shown in Figure 2b, AFDX data networks do not possess this property. Algorithms exist that can convert arbitrary networks into feed-forward networks. The simplest algorithm to break cycles in a network is the spanning tree algorithm, yet, it results in large underutilization. We apply turn-prohibition instead [22].

a) Turn Prohibition: Instead of removing entire links, the algorithm instead removes *turns* [23], which are defined as a specific pair of input-output links of a device. – a model that is compatible with the conversion from device graph to server graph. A difference worth mentioning from this approach to the spanning tree is that the latter may prevent the transmission of any data through an output link of a given node, whereas the turn-prohibition may allow some of those transmissions as long as the data sent arrives from certain pre-determined set of input links. In other words, instead of fully prohibiting all transmission from device a to device b, this approach allows some of the packets to be transmitted through that link as long as the route it took would not cause a cycle. Turn Prohibition has thus a relatively low impact in the resulting network, no more than $1/3$ of the total number of turns are prohibited. Figure 2c shows the turn-prohibited server graph. Turn prohibition removed the turns $S_{3,1} \rightarrow S_{1,1}$, $S_{2,2} \rightarrow S_{1,2}$, $S_{3,2} \rightarrow S_{4,2}$, and $S_{4,1} \rightarrow S_{2,1}$, i.e., connections in the cycle shown in Figure 2b. Further note that all devices are still connected to each other, yet, not by all alternative paths that existed before.

D. Data Communication

1) AFDX Virtual Links: Virtual Links (VL) in an AFDX network are unidirectional multicast logical links from a source End-System to one or more destination End-Systems. Their resource demand when entering the network is known. It is based on two main parameters:

- Maximum frame length: The maximum frame length limits the maximum packet size that can be sent by the VL.
- Bandwidth Allocation Gap (BAG): The BAG limits the rate in which data can be sent by the VL. It is defined as the minimum time interval in which two Ethernet frames can be generated.

2) Multicast Flows in DNC: In a DNC analysis, VLs are treated like multicast flows. Recent work provides progress on this topic (mcastFFA [16]) but has not been included in tool support yet. We therefore convert multicast links to sets of unicast flows.

The worst-case resource demand of (unicast) flows needs to be bounded with a DNC arrival curve α. It can be derived from the two resource descriptions above: the worst-case burstiness equals one maximum frame length and the subsequent worst-case rate equals $\frac{\text{maximum frame length}}{\text{BAG}}$.

III. Route Optimization Algorithms

Defining flow routes that lead to an efficient use of network resources is a complex problem due to the exponential number of possibilities, each defining a design choice that has to be made. We provide a set of (iterative) algorithms that automate these choices by deriving knowledge about each alternative's cost in terms of the flows' delay bounds. From a network with a given topology as well as sources and sinks for the flows, we then define the best paths to take by flows. In the iterative algorithms, we base our (re)routing decision on the bottleneck flow, which is defined as the flow with the largest delay in the network. We choose the path that optimizes this flow's end-to-end delay bound. Thus, we reduce the network's tail latencies as well.

A. Non-Iterative Algorithms

The first class of algorithms offered are the Non-Iterative ones. They use some previous knowledge of the network to decide on paths for the flows that are beneficial for all flows' delay bounds. These non-iterative algorithms are fast as the DNC analysis is not applied as often as in iterative algorithms - once a flow is assigned a path, it is never modified. I.e., costly derivation of flow entanglements and delay bound computations are executed less often. However, this speed comes with a price; since the flows interfere with each other, older flows added to the network can have their delay bounds increased by the addition of new ones. For simplicity of presentation, sources and sinks are assumed to be servers already chosen from the server graph such that the intended devices in the device graph are connected.

1) Shortest Path: This is the simplest algorithm for choosing flow paths. For each pair consisting of a source and a sink this algorithm returns the shortest path from one to another. As this is a very wide-spread algorithm, it servers us as a fundamental benchmark for the remaining algorithms. The numerical evaluation will show that minimizing each flow's hops between its source and sink results in largest tail latencies. Algorithm 1 depicts our implementation in pseudo code. As we consider this a non-iterative algorithm, when the single loop iterates over the flows in the network each flow is only considered once. Note that the order of flows to add to the network does not matter as the turn-prohibited server graph does not change. Neither is the influence of any other flow considered by applying a DNC analysis.

Algorithm 1: Shortest Path

```
Data: list of flows to be added to an empty network,
      each with source & sink server, but without a
      path.
while list of flows not empty do
    currentFlow = flowList.getFirst();
    source = currentFlow.source;
    sink = currentFlow.sink;
    currentFlow.path = GetShortestPath(source,sink);
    network.addFlow(currentFlow);
    flowList.removeFirst();
end
```

2) Greedy: Secondly, we benchmark agains a greedy algorithm as often used in practice. It uses the DNC delay results to choose a flow's path. Again, flows are added to the network one-by-one. As before a flow once added is not altered afterwards.

In contrast to the shortest path algorithm, this non-iterative algorithm considers the influence of flows on each other. It computes the delay bound that the flow to be added experiences on all alternative paths between its designated source and sink servers. The algorithm then takes the path with the smallest delay bound. However, as new flows are added in the network the previously added flows might be impacted negatively and thus suffer from larger delays; the previous ordering of delay bounds on alternative paths might even change. As the greedy algorithm is non-iterative, it does not react to this potential later invalidation of previous decisions' basis.

We chose to add flows in decreasing order of their long-term arrival rate with the assumption that such flows with high long-term arrival rates would have a bigger impact on the servers. For this reason, the shortest paths should be given to them, so that they interfere with as little servers as possible. For a empty network (no flows) with all servers having the same service curve, the path with minimal delay is also the shortest path. The greedy algorithm is depicted in pseudo code in Algorithm 2.

Algorithm 2: Greedy

```
Data: list of flows to be added to an empty network,
      each with source server, sink server and arrival
      curve, but without a path.
Sort the list of flows by their arrival curves;
while list of flows not empty do
    currentFlow = flowList.getFirst();
    source = currentFlow.source;
    sink = currentFlow.sink;
    possiblePathsList = GetAllPaths(source,sink);
    temporaryFlow = currentFlow.copy();
    bestDelay = $+\infty$;
    while list of possible paths not empty do
        temporaryFlow.path =
          possiblePathsList.getFirst();
        network.addFlow(temporaryFlow);
        currentDelay = DNCanalysis(network,
          temporaryFlow);
        network.removeFlow(temporaryFlow);
        possiblePathsList.removeFirst();
        if currentDelay lesser than bestDelay then
            bestDelay = currentDelay;
            currentFlow.path = temporaryFlow.path;
        end
    end
    network.addFlow(currentFlow);
    flowList.removeFirst();
end
```

3) Load Balancer: This algorithm (see Algorithm 3) tries to evenly distribute the load across all servers by adding flows accordingly. I.e., it is a greedy algorithm that takes a different DNC result into account. Load is defined as the sum of the long-term arrival rates of the flows that cross the server. This sustained rate of a flow is not altered by a DNC analysis as long as the network can guarantee for bounded delays. Thus, the flows' long-term arrival rates that are known at the location they enter their respective first server can safely be used for this routing decision. Again, all alternative paths between the source and sink of a flow to be added are checked. The algorithm chooses the one that minimizes the maximum load on the path. This total load is a flow-local indicator of the network's overall load balance. It keeps the algorithm's complexity low but does not guarantee that the paths are optimally chosen for a balanced load. Numerical experiments show that the algorithm achieves small maximum delay bounds nonetheless and that execution times are small.

Algorithm 3: Load Balancer

```
Data: list of flows to be added to an empty network,
      each with source server, sink server and arrival
      curve, but without a path.
Sort the list of flows by their arrival curves;
while list of flows not empty do
    currentFlow = flowList.getFirst();
    source = currentFlow.source;
    sink = currentFlow.sink;
    possiblePathsList = GetAllPaths(source,sink);
    temporaryFlow = currentFlow.copy();
    bestLoad = $+\infty$;
    loadPerServer = GetLoadPerServer();
    while list of possible paths not empty do
        temporaryFlow.path =
          possiblePathsList.getFirst();
        currentLoad = 0;
        serverList = temporaryFlow.path;
        while list of servers in path not empty do
            server = serverList.getFirst();
            currentLoad += loadPerServer.get(server);
            serverList.removeFirst();
        end
        possiblePathsList.removeFirst();
        if currentLoad lesser than bestLoad then
            bestLoad = currentLoad;
            currentFlow.path = temporaryFlow.path;
        end
    end
    network.addFlow(currentFlow);
    flowList.removeFirst();
end
```

B. Iterative Algorithms

In contrast to the above algorithms, the iterative algorithms we propose next make use of a iterative search to compensate the lack of analysis of older flows in the algorithms. As mentioned in Section III-A2, adding a flow to the network can actually invalidate the indicators used to decide on a previously added flow's path. Iterative algorithms try to mitigate this potential problem and its negative impact on the delay bound distribution in a network by reducing maximum latencies in successive iterations of rerouting flows as new flows are added to the network. The basic idea is that after a number n of flows have been added, a iterative search is initiated looking for the network's bottlenecks. Each iteration rearranges flows in a way that their delay bound becomes smaller. We chose this flow-local DNC result for performance reasons (see also Section III-A3). This partially solves the problem mentioned before. In very few cases however, this can in fact lead to an iteration's result being worse than the one it attempted to improve, generating a different bottleneck with higher delay. The n chosen for this paper is 100 flows.

Such iterative algorithms need a well-defined termination condition. The obvious condition is to stop as soon as no improvement can be found. Yet, this does not guarantee termination. We decided to additionally limit the amount of iterations to a total of 10 to guarantee termination. This constitutes another tradeoff between computational cost and optimality of the final result that can be adjusted in future work. In the remainder of this section, we present our iterative algorithms. As mentioned above, they make use of the non-iterative algorithms presented in Section III-A for the creation of partial networks on which the iterative search is executed. The considerations regarding accuracy and cost are evaluated in Section V.

Algorithm 4: Iterations

```
Data: a network with flows
counter = 0;
optimized = false;
while not optimized and counter is lesser than maximum iterations do
    flowList = network.getFlows();
    worstDelay = 0;
    bottleneck = flowList.getFirst();
    while list of flows not empty do
        currentFlow = flowList.getFirst();
        currentDelay = DNCanalysis(network, currentFlow);
        if currentDelay is higher than worstDelay then
            worstDelay = currentDelay;
            bottleneck = currentFlow;
        end
        flowList.removeFirst();
    end
    network.remove(bottleneck);
    source = bottleneck.source;
    sink = bottleneck.sink;
    possiblePathsList = GetAllPaths(source,sink);
    temporaryFlow = bottleneck.copy();
    bestDelay = $+\infty$;
    while list of possible paths not empty do
        temporaryFlow.path = possiblePathsList.getFirst();
        network.addFlow(temporaryFlow);
        currentDelay = DNCanalysis(network, temporaryFlow);
        network.removeFlow(temporaryFlow);
        possiblePathsList.removeFirst();
        if currentDelay lesser than bestDelay then
            bestDelay = currentDelay;
            currentFlow = temporaryFlow.copy();
        end
    end
    network.add(currentFlow);
    if bestDelay is equal to worstDelay then
        optmized = true;
    end
    counter = counter + 1;
end
```

1) Iterative Search: The algorithm, depicted in Algorithm 4, finds the bottleneck of the network, defined as the flow with the highest delay bound. Then, it optimize the network design by generating all possible paths for this flow and rerouting it via the one with the least delay bound. It stops after a given number of iterations set by the user or if it can not find a better path to the bottleneck. Algorithm 4 comprehensively depicts the proceedings in pseudo code. The non-iterative algorithms that make use of this algorithm are explained below with the nomenclature "+ Iterations" .

2) Shortest Path + Iterations: In order to evaluate the Iterative Search algorithm complexity and performance alone, a network created by the Shortest Path was used. The Iterative Search is executed in this complete network, and the behavior can be used to give an insight in the behavior of the next algorithms. This shows the positive impact that one execution of the Iterative Search algorithm can have in a network, and the cost attached to it. This algorithm is depicted in Algorithm 5.

Algorithm 5: Shortest Path + Iterations

```
Data: list of flows to be added to an empty network,
      each with source & sink server, but without a
      path.
while list of flows not empty do
    currentFlow = flowList.getFirst();
    source = currentFlow.source;
    sink = currentFlow.sink;
    currentFlow.path = GetShortestPath(source,sink);
    network.addFlow(currentFlow);
    flowList.removeFirst();
end
Run Iterative Search algorithm;
```

3) Greedy + Iterations: The greedy algorithm revealed the problem of flows added in the beginning of the execution never being optimized again. Thus, we combine it with an iterative search. However, it already uses DNC delay bound results for the flow to be added. We extend this by the iterative search that covers all the bottleneck flows' delay bounds. Thus, paths of already added flows are optimized for delay bounds but at increased cost of the algorithm. Indeed, execution times of this exhaustive combination of the greedy algorithm and iterative improvements became computationally infeasible. As before, we implemented a flexible tradeoff: the iterations that optimize bottleneck flow paths are only started after another n flows were added to the network, and we have set $n = 100$. With this parameter setting we observe a vast reduction of computational effort. Only few flows were actually rerouted per start of iterative search as most already underwent a path optimization before. Many times, the iterative search did not change any flow, making the algorithm behave just as the greedy algorithm up to this point, yet, with a larger execution time. We present the pseudo code for this in Algorithm 6. Is also important to note that this algorithm is very similar to the Greedy algorithm, making use of the Iterative Search algorithm to solve the short comes of the first.

Algorithm 6: Greedy + Iterations

```
Data: list of flows to be added to an empty network,
      each with source server, sink server and arrival
      curve, but without a path.
Sort the list of flows by their arrival curves;
counter = 1;
while list of flows not empty do
    currentFlow = flowList.getFirst();
    source = currentFlow.source;
    sink = currentFlow.sink;
    possiblePathsList = GetAllPaths(source,sink);
    temporaryFlow = currentFlow.copy();
    bestDelay = $+\infty$;
    while list of possible paths not empty do
        temporaryFlow.path =
          possiblePathsList.getFirst();
        network.addFlow(temporaryFlow);
        currentDelay = DNCanalysis(network,
          temporaryFlow);
        network.removeFlow(temporaryFlow);
        possiblePathsList.removeFirst();
        if currentDelay lesser than bestDelay then
            bestDelay = currentDelay;
            currentFlow.path = temporaryFlow.path;
        end
    end
    network.addFlow(currentFlow);
    flowList.removeFirst();
    if counter equals to 100 then
        Run Iterative Search algorithm;
        counter = 1;
    else
        counter = counter + 1;
    end
end
```

4) Load Balancer + Iterations: The Load Balancer algorithm uses an heuristic to select the best paths. For this reason, the selected paths do not guarantee for optimality and an succeeding iterative search is still worthwhile. Therefore, we combined it with an iterative search in the same way as with the greedy algorithm. Moreover, it combines two metrics of bottlenecks: the load-based one and the delay bound-based one. We skip a pseudo code depiction for brevity.

IV. Related Work

In this section, we present some background on iterative algorithms focused on network optimization. Also, we comment about some other works which showed potential application of network calculus results for optimization of designs, rather than just network analysis. In the end, we comment about other available Network Calculus tools.

Optimizing networks using iterative search is an established research topic. The literature focuses on construction and design of network topologies together with flow definition, such as [24] stating that shortest paths are optimal in their context. This work and others tries to minimize cost while respecting flow delays constraints.

It is also known the potential that Network Calculus has for optimization of networks delays, and more specifically of AFDX networks, as shown by [8]. This paper makes use of priority assignment to optimize allocation of the flows on the network. It also shows that is possible to use DNC with iterative searches like genetic algorithms. Priority optimization was also applied in [20] to reduce tail latencies in data center networks. Other works such as [25] also used iterative algorithms with heuristics, but focused on routing protocols.

Further DNC tools are available as well, some of these have also been extended for the analysis of AFDX data networks. For instance, the commercial PEGASE tool [26] for DNC and the open-source Net2Plan-AFDX tool [27], an extension of Net2Plan to apply DNC to AFDX. Another DNC tool, WOPANets, also offers optimization features using a simplex algorithm and genetic algorithms. Yet, these tools do not provide the advanced PMOO analysis of the DiscoDNC.

V. Numerical Evaluation

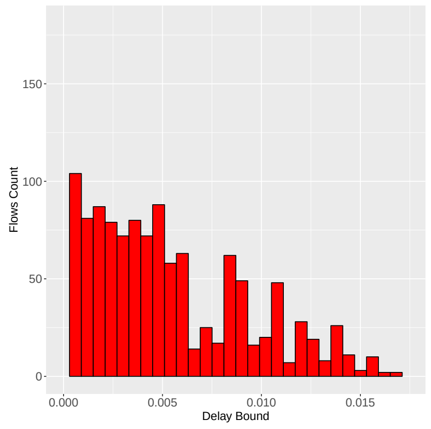
We implemented the above algorithms in the DiscoDNC [10] to evaluate their performance. Section V-A presents our methodology for reproducible numerical results, Section V-B provides results for delay bound distributions and tail latencies, and Section V-D gives the computational cost of the applied algorithms.

A. Methodology and Parameters for Network Instantiation

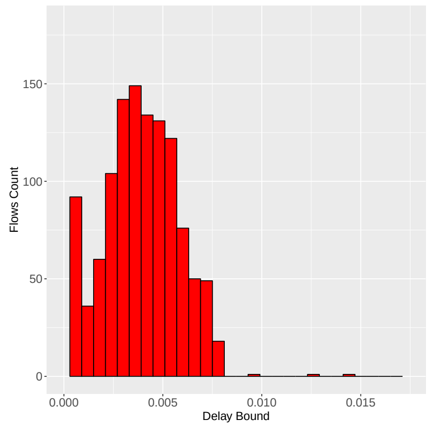
To compare the algorithms' performance, all of them were applied in the same AFDX network. The network generation is described in [17]. It offers the various variables that we set as follows:

- Topology creation setting:
  - Number of Virtual links - varied across experiments
  - Minimum and Maximum Number of End-Systems - uniformly distributed in $[90, 110]$
  - Minimum and Maximum Connections per Switch - uniformly distributed in $[2, 4]$
  - Minimum and Maximum End-Systems per Switch - uniformly distributed in $[8, 16]$
  - Number of Switches - fixed to 8
  - Maximum Number of VLs per End-System - fixed to 25
  - Maximum Number of End-Systems Receiving the Same VL - fixed to 15
  - Minimum and Maximum Destinations per VL - fixed to 3 (low number due to the required unicast flow transformation)
- Virtual Link (Flow) creation setting:
  - Max frame size - uniformly distributed into 3 groups, which are given in bytes and uniformly distributed in $[100, 400]$, $[100, 800]$, or $[100, 1400]$
  - BAG - given in milliseconds randomly selected from this possibilities: $[2, 4, 8, 16, 32, 64, 128]$

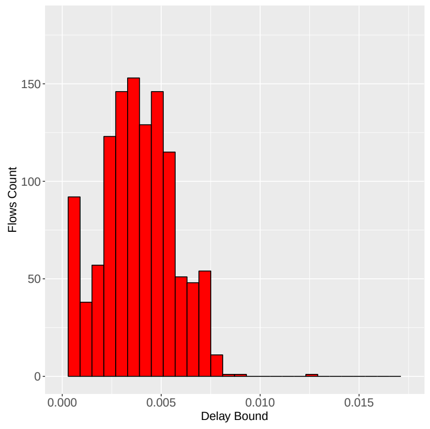
We define the size of a network by the amount of VLs as this is the variable we freely scale. We tested 20 randomly



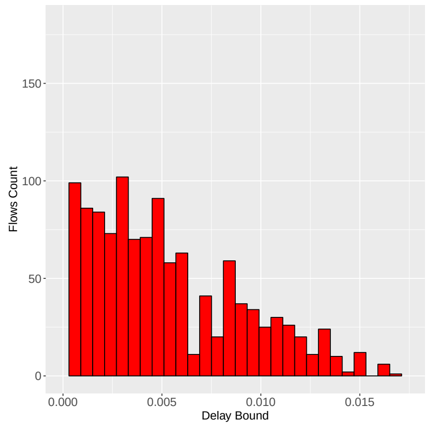
(a) Shortest Path.



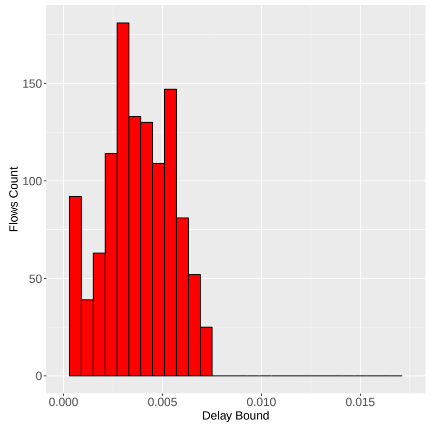
(b) Greedy.



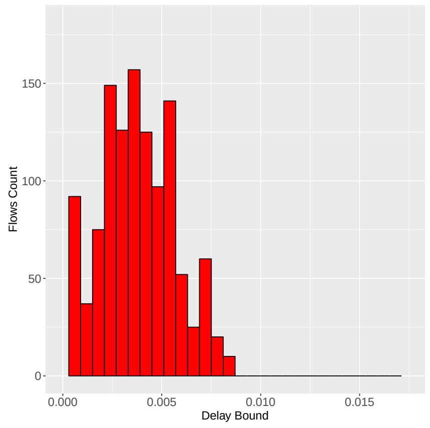
(c) Load Balancer.



(d) Shortest Path + Iterations.



(e) Greedy + Iterations.



(f) Load Balancer + Iterations.

Figure 3: Delay bound distribution results for an AFDX data network with 1200 flows, derived with the DNC PMOO analysis.

created networks of every size. As some variables are set to constant values, the generated networks are not too distinct from each other; this similarity increases comparability between networks. Yet, their switches (devices) have different connections and flows have different combinations of source and sinks.

Another factor tested was the type of analysis. All experiments were executed using TFA, SFA and PMOO analysis implementations offered by the DiscoDNC to investigate they impact of this choice.

All tests were executed on a Supermicro X7DVL server equipped with two Intel Xeon E5420 CPUs and 12GB RAM. In order to achieve expressive results on execution times, only one algorithm was executed at a time.

B. Delay Bound Distributions

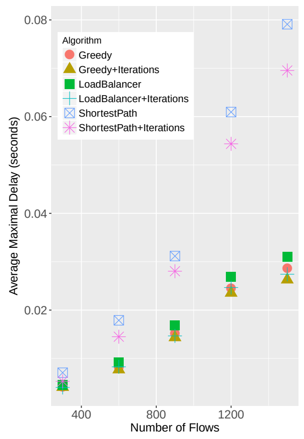
We show the distribution of delay bounds in a single sample AFDX data network with 1200 flows in Figure 3. The Shortest Path algorithms creates flow paths resulting in largest delay bounds. Moreover, the distribution of flows is more spread over the histogram's bins, i.e., the tail is not defined by a small number of outliers. The Shortest Path + Iterations is not able to improve on this situation.

The other non-iterative algorithms on the other hand have a much clearer tendency to the small delay bounds for most of the flows. Both bottleneck metrics, end-to-end delay bound as used in Greedy as well as the load used in Load Balancer, achieve high density of small flow delays. The average and the maximum observed delay bounds are reduced. However, the algorithms also suffer from outliers defining large tail latencies (see Figures 3b and 3c).

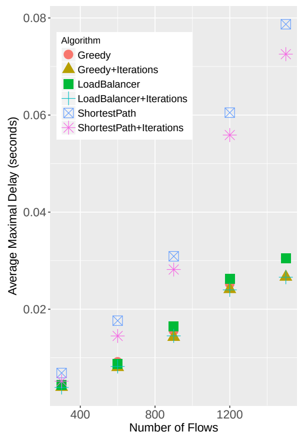
This problem is naturally addressed by the "+ Iterations" approaches shown in Figures 3e and 3f. They work on the bottlenecks found in the networks, considering the interference of the flows on each other. Therefore, Greedy + Iterations and Load Balancer + Iterations have their maximum delay bound reduced by rerouting their respective outlier flows. Overall, Greedy + Iterations is the one with the most flows in the lesser delay bounds region. I.e., the mix of the bottleneck metrics load and end-to-end delay bound was not beneficial.

C. Delay Bounds across Network Sizes

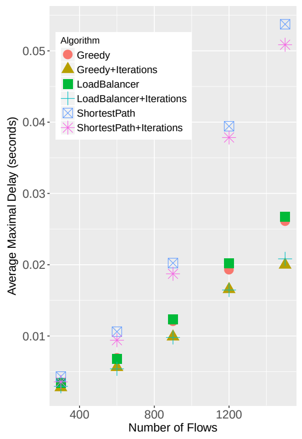
Another visualization is the maximum delay bound calculated by all three DNC analyses for a given AFDX network with increasing amount of VLs (flows). This is shown in Figures 4a to 4c. It depicts the average maximum delay of the



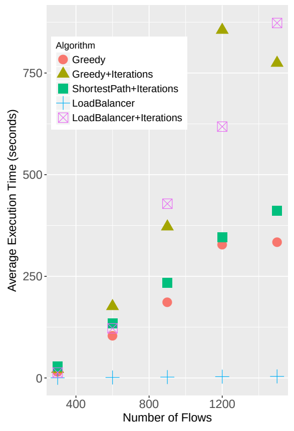
(a) TFA maximum delay bounds.



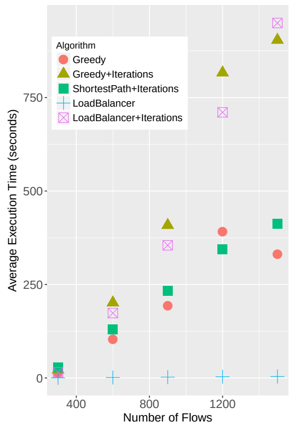
(b) SFA maximum delay bounds.



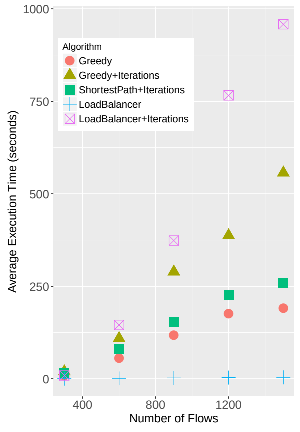
(c) PMOO maximum delay bounds.



(d) TFA execution times.



(e) SFA execution times.



(f) PMOO execution times.

Figure 4

network flows, giving an indicator of the average performance of each algorithm. The figures show that the algorithm with the least maximum delay is the Greedy + Iterations. In most cases, Load Balancer + Iterations follows closely. Both seem to behave very similarly, even regarding execution times shown in Figures 4d to 4f. Again, the overall network is slightly more optimized by using the Greedy + Iterations.

The best algorithms implemented were the Load Balancer + Iterations and the Greedy + Iterations. They both show performances far superior to the Shortest Path. Again, they are behaving quite similar. The relative order between them depends on the actual network and the DNC analysis. None performs constantly better than the other, as shown in Figure 4. Greedy + Iterations also yields the most optimized networks when considering only to the maximum delay bounds (tail latencies) instead of the distribution of all flows' delay bounds.

D. *Execution times*

All of the execution times are depicted in the point graphs in Figures 4, lower row, to give insight on the cost of the algorithms. The Load Balancer + Iterations and the Greedy + Iterations both have similar execution times. These are the largest among the experiments and scale the worst due to the many iterations they both run. Their complexity can be computational explained by comparison to the Shortest Path + Iteration execution time showing the Iterative Search algorithm's complexity for a complete network (all flows added). Since the other "+ Iteration" algorithms execute the iterative search many times, their execution times is negatively effected to a large extent. Another observation is that execution times for the algorithms changes vastly between the applied DNC analysis, TFA, SFA, or PMOO analysis. Since many of the algorithms heavily use these DNC analyses, slightly faster ones show a considerably smaller overall cost. The usually faster being the PMOO analysis is a positive observation, as it also gives the least pessimistic delay bounds (see 4, upper row).

Note that the Shortest Path algorithm was not included in the execution time presentation. This is because it is executed in the creation of the network, as is the default algorithm used by the AFDX network generator, and not as a separated experiment. However, it's execution time is naturally lower than the Load Balancer's one since it consists of less computations. Is also worth noting that the LoadBalancer algorithm has a performance similar to the Greedy, but with the best scalability from all the algorithms.

VI. Conclusion

In this paper we addressed the problem of exploring a network design space in order to optimize flow delay bounds. We started our presentation with the interdependency between deterministic network calculus (DNC) modeling and analysis in order to applied DNC in different algorithms – non-iterative ones and iterative ones – that explore potential improvements by rerouting of flows in a fixed network. The design alternatives created by these algorithms show vastly different characteristics and the algorithms themselves show different potential for scalability to larger networks. Overall, the iterative design space explorations provide best results, yet at a computational cost penalty. In numerical evaluations with networks resembling AFDX data networks, we quantify these aspects in more detail such that a profound decision about the most suitable algorithm can be made in the design phase of a network.

References

[1] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, “Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once,” in *Proc. of GI/ITG MMB*, 2008.

[2] J. B. Schmitt, N. Gollan, S. Bondorf, and I. Martinovic, “Pay Bursts Only Once Holds for (Some) non-FIFO Systems,” in *Proc. IEEE INFOCOM*, April 2011.

[3] S. Bondorf and J. B. Schmitt, “Calculating Accurate End-to-End Delay Bounds – You Better Know Your Cross-Traffic,” in *Proc. of EAI ValueTools*, 2015.

[4] ——, “Improving Cross-Traffic Bounds in Feed-Forward Networks – There is a Job for Everyone,” in *Proc. of GI/ITG MMB & DFT*, 2016.

[5] S. Bondorf, “Better bounds by worse assumptions – improving network calculus accuracy by adding pessimism to the network model,” in *Proc. of IEEE ICC*, 2017.

[6] S. Bondorf, P. Nikolaus, and J. B. Schmitt, “Quality and Cost of Deterministic Network Calculus – Design and Evaluation of an Accurate and Fast Analysis,” in *Proc. of ACM SIGMETRICS*, 2017.

[7] J. Grieu, “Analyse et évaluation de techniques de commutation ethernet pour l’interconnexion des systèmes avioniques,” Ph.D. dissertation, INPT, 2004.

[8] F. Frances, C. Fraboul, and J. Grieu, “Using Network Calculus to Optimize AFDX Network,” in *Proc. of ERTS*, 2006.

[9] S. Bondorf and J. B. Schmitt, “Boosting Sensor Network Calculus by Thoroughly Bounding Cross-Traffic,” in *Proc. of IEEE INFOCOM*, 2015.

[10] ——, “The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus,” in *Proc. of ValueTools*, 2014.

[11] C.-S. Chang, *Performance Guarantees in Communication Networks*. Springer, 2000.

[12] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[13] A. Bouillard, “Composition of service curves in network calculus,” in *Proc. of WCTT Workshop*, 2011.

[14] M. Fidler, “Survey of deterministic and stochastic service curve models in the network calculus,” *Communications Surveys & Tutorials*, 2010.

[15] S. Bondorf and J. B. Schmitt, “Should network calculus relocate? an assessment of current algebraic and optimization-based analyses,” in *Proc. of QEST*, 2016.

[16] S. Bondorf and F. Geyer, “Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows,” in *Proc. of EAI ValueTools*, 2016.

[17] M. Boyer, N. Navet, and M. Fumey, “Experimental Assessment of Timing Verification Techniques for AFDX,” in *Proc. of ERTS*, 2012.

[18] W. Y. Poe, M. A. Beck, and J. B. Schmitt, “Achieving High Lifetime and Low Delay in Very Large Sensor Networks using Mobile Sinks,” in *Proc. of IEEE DCOSS*, 2012.

[19] R. F. Coelho, G. Fohler, and J.-L. Scharbarg, “Worst-Case Backlog for AFDX Network with n-Priorities,” in *Proc. of RTN Workshop*, 2014.

[20] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, “PriorityMeister: Tail Latency QoS for Shared Networked Storage,” in *Proc. of ACM SOCC*, 2014.

[21] A. Bouillard, L. Jouhet, and E. Thierry, “Tight Performance Bounds in the Worst-Case Analysis of Feed-Forward Networks,” in *Proc. of IEEE INFOCOM*, 2010.

[22] D. Starobinski, M. Karpovsky, and L. A. Zakrevski, “Application of Network Calculus to General Topologies Using Turn-Prohibition,” *IEEE/ACM Transactions on Networking*, 2003.

[23] C. J. Glass and L. M. Ni, “The Turn Model for Adaptive Routing,” in *Proce. of ACM ISCA*, 1992.

[24] M. Parsa, Q. Zhu, and Garcia-Luna-Aceves, “An iterative algorithm for delay-constrained an iterative algorithm for delay-constrained minimum-cost multicasting,” *IEEE/ACM Transactions on Networking*, 1998.

[25] A. Riedl, “A hybrid genetic algorithm for routing optimization in ip networks utilizing bandwidth and delay metrics,” in *Proc. of IEEE IPOM Workshop*, 2002.

[26] M. Boyer, N. Navet, X. Olive, and E. Thierry, “The PEGASE project: precise and scalable temporal analysis for aerospace communication systems with network calculus,” in *Proc. of ISoLA*, 2010.

[27] L. Fernandez-Olmos, F. Burrull, and P. Pavon-Marino, “Net2Plan-AFDX: An open-source tool for optimization and performance evaluation of AFDX networks,” in *Proc. of IEEE/AIAA DASC*, 2016.