Quality/Latency-Aware Real-time Scheduling of Distributed Streaming IoT Applications

KAMYAR MIRZAZAD BARIJOUGH, ZHUORAN ZHAO, and ANDREAS GERSTLAUER, The University of Texas at Austin

Embedded systems are increasingly networked and distributed, often, such as in the Internet of Things (IoT), over open networks with potentially unbounded delays. A key challenge is the need for real-time guarantees over such inherently unreliable and unpredictable networks. Generally, timeouts are used to provide timing guarantees while trading off data losses and quality. The schedule of distributed task executions and network timeouts thereby determines a fundamental latency-quality trade-off that is, however, not taken into account by existing scheduling algorithms. In this paper, we propose an approach for scheduling of distributed, real-time streaming applications under quality-latency goals. We formulate this as a problem of analytically deriving a static worst-case schedule of a given distributed dataflow graph that minimizes quality loss while meeting guaranteed latency constraints. Towards this end, we first develop a quality model that estimates SNR of distributed streaming applications under given network characteristics and an overall linearity assumption. Using this quality model, we then formulate and solve the scheduling of distributed dataflow graphs as a numerical optimization problem. Simulation results with random graphs show that quality/latency-aware scheduling improves SNR over a baseline schedule by 50% on average. When applied to a distributed neural network application for handwritten digit recognition, our scheduling methodology can improve classification accuracy by 10% over a naive distribution under tight latency constraints.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Real-time systems*; • **Computing methodologies** → *Distributed computing methodologies.*

Additional Key Words and Phrases: scheduling, real-time, streaming, IoT, open network

ACM Reference Format:
Kamyar Mirzazad Barijough, Zhuoran Zhao, and Andreas Gerstlauer. 2019. Quality/Latency-Aware Real-time Scheduling of Distributed Streaming IoT Applications. *ACM Trans. Embedd. Comput. Syst.* 37, 4, Article 111 (October 2019), 23 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Embedded systems are becoming increasingly distributed and networked. In many cases, such as the Internet of Things (IoT), they use open networks that can have losses and unpredictable or potentially unbounded delays. At the same time, embedded applications often interact with the physical world and have hard real-time requirements. A key challenge in deployment of distributed embedded applications is to enable distributed hard real-time execution over such open networks. Traditional distributed computing and IoT frameworks do not support real-time execution [1, 10,

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2019.

Authors' address: Kamyar Mirzazad Barijough, kammirzazad@utexas.edu; Zhuoran Zhao, zhuoran@utexas.edu; Andreas Gerstlauer, gerstl@ece.utexas.edu, The University of Texas at Austin, Department of Electrical and Computer Engineering, 2501 Speedway, Austin, TX, 78712.

1539-9087/2019/10-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

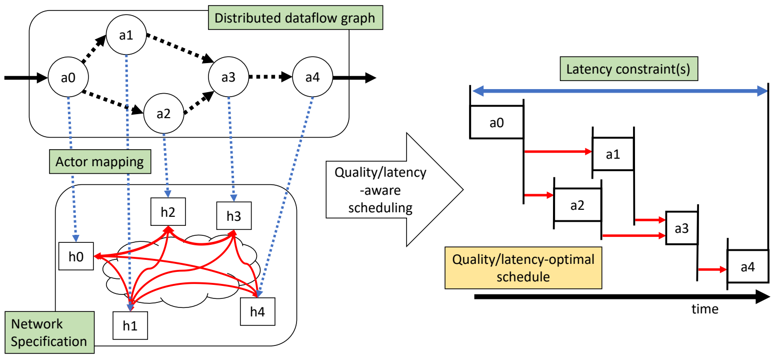

Distributed dataflow graph
a0
a1
a2
a3
a4
Actor mapping
h0
h1
h2
h3
h4
Network Specification
Quality/latency -aware scheduling
Latency constraint(s)
Quality/latency-optimal schedule
time

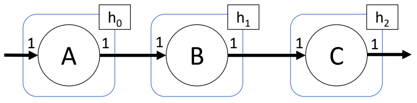
Fig. 1. Overview of quality/latency-aware scheduling.

17, 23]. Various extension have been proposed to address real-time constraints [22, 30], but their guarantees depend on worst-case network latency bounds. By contrast, there exist networking protocols for real-time data transfers over open networks, but they are intended for end-to-end communication and not distributed computing [11]. In the embedded community, there have been proposals to extend traditional Models of Computation (MoC) for distributed applications [30]. However, such approaches aim to fit existing models to distributed systems by requiring high precision time synchronization and reliable networks with bounded latency.

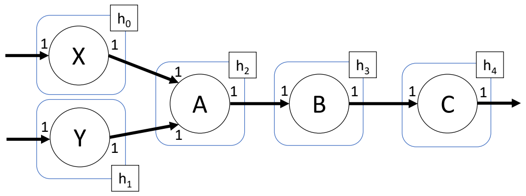
Many embedded applications are of streaming nature and, therefore, are best expressed as dataflow MoCs. In such models, when using a traditional data-driven schedule for dataflow graphs with actors mapped to open network hosts, no latency guarantee can be provided for the graph's output. To provide an end-to-end latency guarantee, the sink actor's wait time on input can be limited by allowing it to time out and fire without data similar to what is done in networking protocols [11]. The more relaxed this latency and hence timeout constraint is, the more likely it is that input data will arrive at the sink before it fires. If data is not present at the time of firing, the sink will have to handle the data losses, e.g. by interpolating the missing data from previous inputs, which will result in quality degradation. As such, there is a fundamental trade-off between latency and quality in distributed real-time execution. Existing work has not studied such quality/latency trade-offs. In the networking community, trade-offs have been explored for end-to-end actor pairs, but it has not been addressed how to extend such approaches to longer, distributed actor chains. A key question is thereby how to optimally distribute an end-to-end latency budget across multiple links to optimize overall application quality.

Recently, an extension of dataflow models targeting distributed execution called Reactive and Adaptive Dataflow (RADF) has been proposed [8]. It formalizes the notion of network losses and firing of actors without data in a deterministic and sound manner using concepts of lossy channels and empty tokens. This enables static analysis, but the authors do not discuss how to detect losses, assign timeouts and derive quality/latency-optimized RADF schedules.

In this paper, we present a quality/latency-aware methodology for real-time scheduling of distributed dataflow graphs over unpredictable and unreliable open public networks. To the best of our knowledge, this is the first work to address this problem. We propose a static scheduling and analysis approach that optimizes worst-case guarantees on average quality of distributed streaming applications while satisfying latency constraints. Resulting schedules provide a baseline that can be further improved at runtime. Figure 1 shows the overview of our methodology. Our scheduling algorithm takes streaming applications expressed as distributed dataflow graphs, a



(a) A graph with linear actor chain.



(b) A graph with multiple source actors.

Fig. 2. Examples of distributed dataflow graphs.

network specification, actor-to-host mapping, and latency constraints as inputs and produces a quality/latency optimal schedule.

Our main contributions in this paper are:

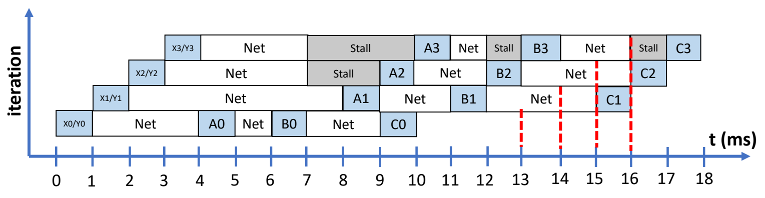
(1) We formalize the distributed real-time execution of streaming applications under latency constraints as an extension of existing RADF models.
(2) We develop an analytical quality model for translating channel delivery rates into overall application quality of distributed dataflow graphs.
(3) We formulate distributed scheduling under latency constraints and quality goals as an optimization problem and develop an approach to numerically solve such problems.
(4) We evaluate our approach on random graphs and a distributed neural network application, demonstrating that optimized schedules can improve quality of random graphs by 50% on average while increasing neural network classification accuracy by 10% under tight latency constraints over a baseline distribution.

The rest of the paper is organized as follows: In Sections 2 and 3, we first motivate the need for quality/latency-aware scheduling through an example and discuss related work. Then, we present our formalization of distributed dataflow in Sections 4. We discuss the quality model and formulation of the scheduling problem in Section 5. Evaluation of our methodology is described in Section 6. Finally, Section 7 concludes the paper with a summary and directions for future work.

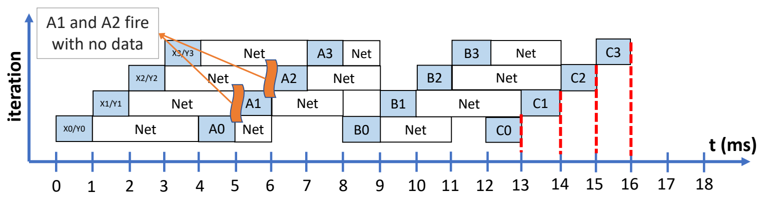
2 MOTIVATIONAL EXAMPLE

Figure 2a shows a simple dataflow graph with a linear chain of actors executed in a distributed fashion on three network nodes or hosts (h_0 through h_2). To provide a latency guarantee, we assign a timeout to the sink actor (C) and allow it to fire without data. Given that dataflow graphs often execute in steady state periodically, the timeout of actor C between firings translates into an offset between its firing time and that of actor A. This offset can be statically calculated by subtracting C's worst-case execution time from the latency constraint. The more relaxed this constraint is, the more likely it is that input data will arrive at C before it fires.

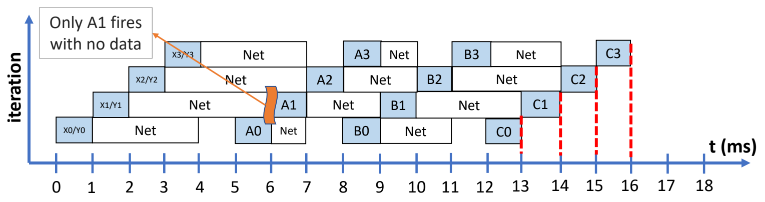
In addition to sink actor, intermediate actors might require their own timeout in cases where data order is important, such as actors with multiple inputs or with state. Figure 2b shows an example of such a graph, where actor A fuses data from X and Y. If A receives data on one input, it would need to buffer and wait until matching data is received on the other input. The RADF model and our work is based on allowing not only the sink but any actor in a graph to fire without (or only



(a) A pure data-driven schedule.



(b) A schedule with uniform latency budget distribution.



(c) A schedule with optimized latency budget distribution.

Fig. 3. Comparison of different schedules for graph of Figure 2b.

with partial) data and in turn produce outputs with reduced quality, e.g., by interpolating results from previous data. If delays exceed overall latency constraints, at least one of the actors in the chain needs to fire without data. Since it is better to allow partial computation with a single input at A rather than letting C fire without any data, A can be assigned its own timeout. This introduces further trade-offs: the smaller the offset of A relative to X/Y is, the higher the data loss at A and the lower the loss at C, and vice versa. Further extending the timeout concept, even actor B can time out to react to network delay and loss earlier in the chain.

Figure 3 shows three different schedules for the graph of Figure 2b under period and latency constraints of 1ms and 13ms, respectively. Figure 3a shows a pure data-driven schedule. Due to increased network delay in the second iteration, the constraint is violated in the second, third and, subsequently, all following iterations. Note that if actors A, B and C are allowed to fire out of order, violations in the third and subsequent iterations can be avoided. However, in general dataflow graphs, e.g., in case of actors with state, deterministic token and actor firing orders need to be maintained, where large network delays or network losses can lead to long-lasting or permanent latency violations in a data-driven schedule.

In Figure 3a, we can allow sink actor C to time out and thus avoid deadline violations. However, this would require C to fire without data in the second and all subsequent iterations. By contrast, Figure 3b shows a schedule where the latency budget is equally distributed between input channels of actors A, B and C, giving each a timeout of 3ms. As seen in the figure, the constraint is met in all four iterations, but in the second and third iteration, A fires without any data. Crucially, however, rather than violating the deadline in all subsequent iterations, this schedule avoids any losses in the fourth and following iterations. Noticing that increasing the timeout of actor A can eliminate its loss in the third iteration, Figure 3c shows an alternative schedule where delay budgets are shifted by 1ms from C to A. This schedule prevents A from firing without data in the third iteration and improves overall quality. This example shows that, depending on the network and application characteristics, there is a non-trivial schedule that optimizes the quality/latency trade-off.

For a simple graph such as the one in Figure 2a, the output quality can be defined as the number of iterations that execute without a loss. We can assign a probability of an actor firing without a loss as the probability of the random network delay on its input channel being smaller than the timeout. The probability of an iteration executing without a loss then becomes the product of loss-free probabilities over all actors and links in the chain. To maximize this product, all probabilities should be made equal. This requires timeouts to be selected in accordance with differences in network delays across channels. Furthermore, in general graphs, more complex, application-specific quality models will be needed to derive optimal timeouts.

3 RELATED WORK

There exist many frameworks that enable programming in the context of distributed systems by supporting Remote Procedure Call (RPC) [1, 10, 17, 20, 23], message-passing [6, 9] and streaming [3, 14, 19, 26, 28] semantics. However, the resulting programs cannot provide latency guarantees.

Stream processing frameworks [3, 14, 26, 28] claim to provide real-time dataflow processing, but they focus on throughput-oriented processing without hard real-time guarantees nor explicit consideration of network effects, and either require reliable communication or randomly discard data.

In the networking community, the real-time transfer protocol (RTP) [11] has been used for video and audio streaming. RTP delivers timing guarantees by exposing the occasional late and lost packets to applications. However, it is limited to end-to-end transfers and does not allow for implementation of distributed scenarios. RTP can be extended to distributed graphs using a data-driven schedule for intermediate nodes and only timing out the sink node. As described in Section 2, such a setup works if intermediate nodes are allowed to fire in any order, similar to how internet routers process packets. However, in distributed execution of dataflow graphs, such an approach can lead to large quality drops as shown in Figure 3a. Alternatively, RTP can be applied to every actor pair in a larger distributed graph. However, doing so requires optimally partitioning end-to-end latency budgets between pairs, which is the problem we address in our work.

In the embedded domain, extensions of RPC frameworks have been proposed to enable distributed real-time computing [22]. However, they assume underlying networks to provide real-time guarantees and do not consider quality trade-offs. Using more formal model-based methods, extensions of traditional MoCs have been proposed for distributed systems. PTIDES [30] builds upon a discrete-event model to provide deterministic and real-time execution for distributed cyber-physical systems through static analysis and global ordering of timestamped events. However, it requires high-precision time synchronization and reliable networks with bounded latency that is not generally achievable or only provided with very loose guarantees in open public networks. PTIDES abstracts away the network non-determinism using worst-case assumptions and does not have a notion of quality/latency trade-offs. There also has been proposals to extend data flow models to capture dynamic network effects by accounting for lossy scenarios and failure probabilities [7, 8, 29]. However, they either target the permanent link failures [29] or require analysis of multiple scenarios [7] rather than exposing latency/quality trade-offs. By contrast, RADF extends a data flow model with concepts of lossy channels and empty tokens supporting open network scenarios while guaranteeing deterministic execution and static analyzability [8]. However, their model is underspecified, the authors do not provide an implementation, and included analysis methods do not account for quality or latency associated with network communication. In this paper, we base our work on RADF but extend model semantics to support quality/latency-aware scheduling. We will discuss RADF semantics, limitations and our extensions in Section 4.

In the machine learning community, motivated by the increasing popularity of deep learning applications and limitations of embedded devices, there have been multiple proposals for specifically distributing inference of neural networks across network hosts [15, 21, 31]. These works, however, either ignore network effects and quality/latency trade-offs [15, 31] or are application-specific [21]. In this work, we use the example of a two-layer, distributed neural network for handwritten digit recognition [12, 24] to systematically optimize the quality/latency trade-off using our generic scheduling approach.

4 DISTRIBUTED DATAFLOW MODEL

In this section, we discuss the Reactive and Adaptive Dataflow Model (RADF) that we use for formalization of distributed streaming applications and our proposed timed extensions.

4.1 RADF Basis

RADF [8], in addition to traditional *lossless channels*, provides *lossy channels* that do not require communication to be reliable. Losses in these channels are represented by replacing lost token(s) with *empty token(s)*. This simple extension allows preserving analyzability and determinism of the underlying data flow model even in the presence of unreliable communication.

Although RADF can be based on top of any data flow model, it is introduced on a Synchronous Data Flow (SDF) basis [8]. Following SDF semantics, every actor has a *firing rule* that specifies firing conditions in terms of the number of tokens consumed from input channels and the number of tokens produced in output channels. Given the existence of both empty and non-empty tokens, RADF actors with lossy input channels can have multiple firing rules. Each of these rules correspond to a unique pattern of empty and non-empty input tokens and results in execution of a corresponding actor *variant*. Upon firing, an RADF actor can consume empty tokens as well as non-empty tokens, but is required to produce non-empty tokens regardless. Having multiple variants with potentially different execution characteristics allows applications to dynamically adapt to network losses.

To support modeling of reactivity to external events, RADF further allows the absence of data in external input channels to be modeled as generation of empty tokens. By allowing actors to have an *idle variant*, RADF does not require those actors to execute unless there is at least one non-empty token in their input channels. Idle variants are fired when all input tokens are empty and generate an all empty token sequence. Therefore, they can be used to model actors that are executed under data-driven schedule. Actors with idle variant can, in turn, form a *reactive island*, which refers to a largest chain of actors with idle variants and producer-consumer relationship, where none of the actors in the island executes unless the source actor(s) receive a non-empty token.

4.2 Timed RADF Extension

RADF semantics simplify the distributed execution of actors by basing it on only patterns of empty and non-empty tokens in input channels. However, in practice, any RADF implementation in turn requires an approach for detecting losses and injecting empty tokens. In particular, while empty tokens can make a self-timed and data-driven execution possible even in the presence of losses and unbounded delays, this in turn creates the challenge of detecting losses and injecting empty tokens while meeting real-time guarantees, which RADF itself does not address.

Since one can not wait for a token to not arrive, deciding on losses and empty tokens needs to be based on waiting until some other event indicates that one should give up doing so. In a distributed environment, these indicators need to be based on local information such as channel

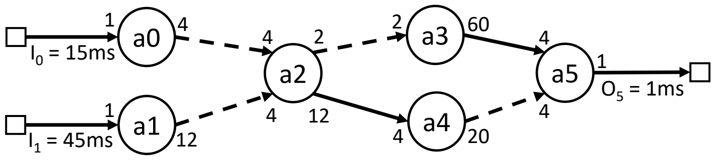


Fig. 4. An instance of a T-RADF graph.

state or time. Basing such decisions purely on local channel state, such as waiting for a certain number of non-empty tokens with higher sequence numbers to arrive, will not allow to provide timing guarantees. Instead, decisions about empty tokens need to be based on some notion of time, which locally can only translate to relative *timeouts between firings*. This is similar to RTP's [11] operation, where a receiver delivers a constant frame rate to an output device.

Motivated by these observations, we propose *timed RADF* (T-RADF) as a model that extends RADF graphs with constant rates attached to external input and output channels. External rates provide a complete specification. With an SDF base, intermediate rates and thus timeouts for all actors can be derived from them. This is also consistent with cyber-physical systems in which sensors and actuators attached to external inputs and outputs come with specified timing constraints. Note that external rates provide exact periods for actors interfacing to them, but only specify an average period of intermediate actors. Therefore, as long as the implementation conforms to external rates, it can vary the firing period of intermediate actors around a *default* period given by external rates.

Figure 4 shows a T-RADF graph with I_0, I_1 and O_5 as external inputs and outputs. Dashed lines between actors depict lossy channels. Solid lines, by contrast, represent lossless channels. I_0 and I_1 sample inputs with rates of 15ms and 45ms, respectively. O_5 produces outputs at a rate of 1ms. Based on the input rates and repetition relationships between producer-consumer pairs, periods for $a0 - a5$ can be derived as 15, 45, 15, 15, 5 and 1 ms, respectively, where the period of $a5$ is consistent with the rate of O_5.

5 DISTRIBUTED DATAFLOW SCHEDULING

For distributed execution of T-RADF graphs under given latency constraints, we need to derive timeouts and other implementation parameters. In addition to the graph, this requires worst-case execution time (WCET) information about each actor, mapping information, latency constraints and a network specification to be known. Without loss of generality, we perform scheduling of graphs assuming a given, pre-determined partitioning, mapping and distribution of the graph across network hosts. Mapping information specifies how actors and network channels are assigned to hosts and network interfaces, respectively. Latency constraints are defined per primary input-output pair as the time offset between start of execution of the input actor and end of execution of the output actor in any iteration. The network specification lists delay and loss characteristic of network paths between hosts. Network delays are assumed to be continuous random variables that are specified in terms of a Network Delay Distribution (NDD), i.e. a probabilistic distribution model that specifies the likelihood of a given one-way network delay in absence of any retransmission [2].

To provide static guarantees, we derive and analyze a baseline firing schedule of T-RADF actors. Meeting latency constraints requires upper bounding of computation and communication times. Analysis of worst-case execution times is a well-studied topic. By contrast, communication delays in public networks are in general unbounded. This fundamentally does not allow any upper bound for latency of lossless channels to be assumed. By contrast, with lossy channels, a given delay limit can be imposed through timeouts.

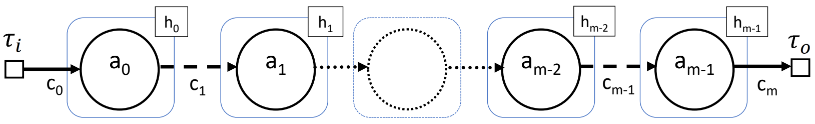


Fig. 5. A mapped T-RADF graph with a linear chain of actors.

The random nature of network delays has two implications on the implementation of lossy channels. Firstly, tokens with delays larger than the set limit will be exposed to the application as empty tokens and therefore affect the result quality. Although using larger timeouts can reduce the number of late tokens, it increases end-to-end latency. We optimize this trade-off between latency and result quality by calculating timeouts and a baseline schedule to maximize output quality under given period and latency constraints. Note that this does not prevent any runtime from further optimizing the latency or quality dynamically. Instead, our approach aims to provide a static schedule that guarantees analytically derived worst-case quality and latency as long as an implementation does not violate the timeouts, i.e. the upper bounds on offsets between actor executions computed by our analysis.

In addition to schedule and timeout computation, since tokens in open networks might arrive out-of-order, maintaining FIFO channel semantics requires buffering of tokens at the destination. We thus further ensure that no token will be lost due to buffer overflows by calculating the maximum required buffer size statically.

5.1 Timeout and Schedule Computation

In the current work, we assume T-RADF graphs to be homogeneous and each host to execute only one actor. Note that as with other dataflow scheduling approaches, general graphs can be explicitly or implicitly converted to homogeneous equivalents during scheduling [13], but this may come at the cost of exponential complexity in graph sizes. The assumption of one actor per host can be satisfied by statically scheduling multiple actors mapped onto the same host into a super-actor. In the general case, hierarchical composition of SDF actors can lead to deadlocks, which can be addressed using relaxed cyclo-static dataflow semantics [18]. We plan to incorporate such relaxations in future work.

Following T-RADF semantics, source and sink actors of a graph will always fire with a constant period. However, analyzing a graph to derive a schedule that provides static guarantees requires instantaneous timeouts of all intermediate actors, which are not specified in a T-RADF model, to be statically derived. We perform a conservative analysis assuming a fixed schedule in which all intermediate actors fire with a constant period as given by the specification. This reduces the timeout problem to determining offsets between periodic actor executions while allowing for a static analysis that provides upper bounds on latency and token losses. In practice, a schedule can be dynamically adjusted to further optimize latency or quality at runtime, e.g. by firing actors and sending outputs early if input tokens arrive before the start of the next period.

Figure 5 shows a graph with a chain of actors connected by lossy channels between actors and hosts. This graph has only one input-output pair (a_0,a_{m-1}) whose latency (l) is equal to the time interval between consumption of a token by a_0 and production of the corresponding token by a_{m-1}. Given the execution time e_i of actor a_i and communication delay d_j of channel c_j:

$$l = \sum_{i=0}^{m-1} e_i + \sum_{j=1}^{m-1} d_j \leq l', \tag{1}$$

where l' is the latency constraint associated with the pair (a_0,a_{m-1}). Note that external channels c_0 and c_m are assumed to have zero communication delay and thus are excluded from the sum.

Satisfying this constraint requires actor WCET bounds $e_i \leq e'_i$ and channel delay bounds $d_j \leq d'_j$ to be known or derived, respectively. At the same time, the choice of d'_j determines the probability of tokens being delivered empty as a function of the NDD and average packet loss rate. A channel c_j will be able to capture all the packets that do not get lost and have a delay of less than or equal to d'_j. Thus, the probability p_j of tokens in channel c_j being delivered (be non-empty) as function of its latency budget is:

$$p_j(d'_j) = (1 - \mu_j) \cdot F_{D_j}(d'_j), \tag{2}$$

where $F_{D_j}(d_j)$ is the cumulative distribution function (CDF) of the random delay variable D_j associated with channel c_j's NDD, and μ_j is c_j's average packet loss rate. Note that this equation assumes that all packet losses and delays are independent.

To minimize the probability of empty tokens, d'_j should be maximized. For a given latency constraint, optimal assignment of d'_j values such that result quality is maximized is generally application-specific. How empty tokens are interpreted depends on actors' *replacement functions*. Therefore, to derive the optimal assignment, we need a quality model that relates individual p_j to overall application quality. In the following, we first develop a quality model that can be expressed in closed form, which then allows us, using Equation 2, to formulate the scheduling problem as a numerical optimization problem with delay assignments as decision variables and the quality model as maximization goal. Tables 1 and 2 provide a summary of notations used for given and derived variables in our formulation, respectively.

5.2 Quality Model

To define a quality model, first we need to choose a quality metric. In this work, we target typical streaming and signal processing applications. As such, we use the signal-to-noise ratio (SNR) of the output actor as the quality metric to optimize. Since analyzing SNR of streaming applications for all possible cases is difficult, we limit ourselves to linear systems, i.e. cases where both actors and replacement functions are linear time series of previously seen values. Other systems can in most cases be supported by approximating them as linear. In the remainder of this section, we formulate an efficient, closed-form quality model based on these assumptions.

We first investigate the simplest case of a graph with a linear chain of actors. In the graph of Figure 5, we can quantify the potential noise $n_j[i]$ due to delivery failure in any channel c_j in graph iteration i as the absolute difference between the value $s_j[i]$ transmitted over channel c_j in a lossless execution and the estimate provided by the consumer actor's replacement function $R_j()$:

$$n_j[i] = |s_j[i] - R_j(x_j[0], \ldots, x_j[i-1])|. \tag{3}$$

The replacement function $R_j(x_j[0], \ldots, x_j[i-1])$ determines the value that the actor substitutes for the lost token as, in general, a function of previously seen actual, i.e. noisy values $x_j[i] = s_j[i] + n_j[i]$, where noise is assumed to be additive.

To estimate overall quality, we need to determine the total noise at the output of the graph, i.e. in channel c_m. Towards this goal, we first quantify the relationship between loss in any intermediate channel and noise at the graph output. In a linear system, the output of an actor is in general a product of its input multiplied by a constant factor. We can formalize this assumption by associating a weight w_j with each actor and hence its output channel c_j in the linear chain. With this, a value $s_h[i]$ in any intermediate channel $c_h, h < m$ propagates to the output as $s_m[i] = s_h[i] \cdot \prod_{j=h+1}^{m} w_j$. Hence, if there is a failure in a channel c_h in the chain, the failure noise at the output of the graph becomes:

$$n_m[i] = |s_h[i] - R_h(x_h)| \cdot \prod_{j=h+1}^{m} w_j, \tag{4}$$

Table 1. Summary of input variables.

Variable	Description
$e'(a)$	worst-case execution time of actor a
$l'(a, a')$	latency constraint of actor pair (a, a')
u_j	producer (actor) of channel j
v_j	consumer (actor) of channel j
$\overrightarrow{w_j}$	weight vector of channel j
$s_j[i]$	noise-free value of channel j at iteration i
$R_j(x_j)$	replacement function of v_j for channel j
α	maximum $n_j[i]/s_j[i]$ ratio across all iterations
$P_{s_j}[i]$	noise-free signal power of channel j
$\mathbf{k}(j)$	set of paths leading to channel j
$in(k)$	input channel of path k
$out(k)$	output channel of path k
$\widetilde{w}_k$	weight of path k
q_j	contribution of output channel j to overall quality
$\mathbf{I}$	set of input channels of graph
$\mathbf{O}$	set of output channels of graph
$\mathbf{B}$	set of channels with initial tokens (backedges)
$\mathbf{F}$	set of channels w/o initial tokens (forward edges)
$\mathbf{L}$	set of actor pairs with constrained latency

Table 2. Summary of derived variables.

Variable	Description
$t_s(a)$	start time of actor a
$\mathbf{t_s}$	set of start times $\{t_s(a_0), t_s(a_1), \ldots\}$
d'_j	latency budget of channel j
p_j	delivery probability of channel j
$\widetilde{p}_k$	delivery probability of path k
$\widetilde{p}_{k\|k'}$	joint delivery probability of paths k and k'
$n_j[i]$	noise in channel j at iteration i
$x_j[i]$	noisy signal value of channel j at iteration i
$N_j[i]$	random noise of channel j at iteration i
$\widetilde{N}_k[i]$	random noise associated with path k at iteration i
$P_{n_j}[i]$	expected noise power of channel j at iteration i
$P_{s_j}[i]$	noise-free signal power of channel j at iteration i
$P_{s_j}^{path}[i]$	signal power delivered on channel j along individual paths at iteration i
$P_{s_j}^{joint}[i]$	signal power delivered on channel j along combinations of paths at iteration i
Q	weighted average of output SNRs

where $x_h = x_h[0], \ldots, x_h[i-1]$ is the signal, i.e. time series of previously seen values in channel c_h.

Assuming that replacement functions are chosen such that noise in any channel is bounded by the noise-free signal value, i.e. there is a constant α such that $n_j[i] \leq \alpha \cdot s_j[i]$, we can derive an upper bound on the noise at the graph output c_m induced by a failure in an intermediate channel c_h as follows:

$$n_m[i] \leq \alpha \cdot s_h[i] \cdot \prod_{j=h+1}^{m} w_j \tag{5}$$

Since the signal $s_h[i]$ in intermediate channel c_h is itself a linear function of the graph's input, the noise bound can be re-written as a function of the input signal s_0:

$$n_m[i] \leq \alpha \cdot s_0[i] \cdot \prod_{j=1}^{m} w_j = \alpha \cdot \widetilde{w} \cdot s_0[i], \tag{6}$$

where $\widetilde{w} = \prod_{j=1}^{m} w_j$. As such, the noise bound becomes independent of the location h of the failure. It can be further shown that the upper bound given by Equation 6 holds generally, regardless of the number of failures.

Due to the probabilistic nature of lossy channels, the noise $n_m[i]$ is in reality a random variable $N_m[i]$, where the noise power $P_{n_m}[i] = E[N_m^2[i]]$ is computed as the expected value of the squared noise. With probability of $\widetilde{p} = \prod_{j=1}^{m} p_j$, none of the channels will fail and $N_m[i]$ will be zero. By contrast, with probability $(1-\widetilde{p})$, there will be at least one loss in a channel in the chain, with failure noise that is upper bounded according to (6). As such, we can bound the noise power $P_{n_m}[i]$ at the output of the linear chain as:

$$P_{n_m}[i] = E\left[N_m^2[i]\right] \leq (1-\widetilde{p})(\alpha \cdot \widetilde{w} \cdot s_0[i])^2. \tag{7}$$

In the general case, actors of the graph might have more than one input or output. We can generalize channel weights to a weight vector $\overrightarrow{w_j}$, where every output of an actor is computed as a weighted sum of its inputs. For each distinct path $k = (c_{in(k)}, \ldots, c_{out(k)})$ in the graph from input channel $c_{in(k)}$ to output channel $c_{out(k)}$, we can thus define a path weight $\widetilde{w}_k$ as the product of all weight vector elements along the channels of the path. Similarly, $\widetilde{p}_k$ is calculated as multiplication of the p_j's along the channels of path k. Finally, we can compute the noise $\widetilde{N}_k[i]$ caused by path k at its output channel $c_{out(k)}$ as before. Under the linearity assumption, we can then express the output noise in such generalized graphs as the sum of noises caused by paths $\mathbf{k}(m) = \{k_m | out(k_m) = m\}$ ending at output c_m. Therefore, and given that the expected value of a sum is equal to the sum of expected values of its individual terms, we can calculate $P_{n_m}[i]$ at output c_m as:

$$\begin{aligned} P_{n_m}[i] = E\left[N_m^2[i]\right] &= E\left[\left(\sum_{k \in \mathbf{k}(m)} \widetilde{N}_k[i]\right)^2\right] \\ &= E\left[\sum_{k \in \mathbf{k}(m)} \widetilde{N}_k^2[i] + 2 \sum_{k \neq k',\, k,k' \in \mathbf{k}(m)} \widetilde{N}_k[i]\widetilde{N}_{k'}[i]\right] \\ &= \sum_{k \in \mathbf{k}(m)} E\left[\widetilde{N}_k^2[i]\right] + 2 \sum_{\substack{k \neq k', \\ k,k' \in \mathbf{k}(m)}} E\left[\widetilde{N}_k[i]\widetilde{N}_{k'}[i]\right]. \end{aligned} \tag{8}$$

Substituting the bound on the noise power of an actor chain from (7) and noting that $\widetilde{N}_k[i]\widetilde{N}_{k'}[i]$ is non-zero only when both paths k and k' fail to deliver, we can obtain an upper bound for $P_{n_m}[i]$ as:

$$\begin{aligned} P_{n_m}[i] \leq &\sum_{k \in \mathbf{k}(m)} (1-\widetilde{p}_k)\left(\alpha^2 \widetilde{w}_k^2 s_{in(k)}^2[i]\right) \\ &+2\sum_{\substack{k\neq k',\\ k,k'\in\mathbf{k}(m)}} (1-\widetilde{p}_{k|k'})\left(\alpha^2 \widetilde{w}_k \widetilde{w}_{k'} s_{in(k)}[i] s_{in(k')}[i]\right), \end{aligned} \tag{9}$$

where $\widetilde{p}_{k|k'}$ is the probability that at least one of the paths k or k' will deliver. We can simplify Equation 9 by rearranging the terms:

$$P_{n_m}[i] \leq \alpha^2 \left(P_{s_m}[i] - \left(P_{s_m}^{path}[i] + 2P_{s_m}^{joint}[i]\right)\right), \tag{10}$$

and factoring out different components $P_{s_m}[i]$, $P_{s_m}^{path}[i]$ and $P_{s_m}^{joint}[i]$ contributing to the noise power bound given as:

$$\begin{aligned} P_{s_m}[i] &= \left(\sum_{k\in\mathbf{k}(m)} \widetilde{w}_k s_{in(k)}[i]\right)^2 \\ P_{s_m}^{path}[i] &= \sum_{k\in\mathbf{k}(m)} \widetilde{p}_k \widetilde{w}_k^2 s_{in(k)}^2[i] \\ P_{s_m}^{joint}[i] &= \sum_{k\neq k', k,k'\in\mathbf{k}(m)} \widetilde{p}_{k|k'} \widetilde{w}_k \widetilde{w}_{k'} s_{in(k)}[i] s_{in(k')}[i]. \end{aligned} \tag{11}$$

Here, P_{s_m} is the total noise-free signal power at output c_m, and intuitively, $P_{s_m}^{path}$ and $P_{s_m}^{joint}$ are the portions of the signal power at c_m that are delivered along individual or combinations of paths, respectively, when they do not experience any losses.

Finally, we can obtain a lower bound on the SNR in iteration i at output channel m by dividing the noise-free signal power by the noise power given by Equation 10:

$$SNR_m[i] = \frac{P_{s_m}[i]}{P_{n_m}[i]} \geq \frac{\alpha^{-2}}{1 - Z_j[i]}, \tag{12}$$

where $Z_m[i]$ is:

$$Z_m[i] = \frac{P_{s_m}^{path}[i] + 2P_{s_m}^{joint}[i]}{P_{s_m}[i]}. \tag{13}$$

Assuming that the noise is upper bounded by the signal value (i.e., $\alpha \leq 1$), individual terms dominate combinations (i.e., $P_{s_m}^{path}[i] \gg P_{s_m}^{joint}[i] > 0$) and inputs to different paths are comparable ($\forall k, k' : s_{in(k)}[i] \approx s_{in(k')}[i]$), we can further simplify Equation 13 and obtain a time-independent conservative estimate for SNR_m:

$$\begin{aligned} SNR_m &\approx \frac{1}{1 - (P_{s_m}^{path}[i]/P_{s_m}[i])} \\ &\approx \frac{1}{1 - (\sum_{k\in\mathbf{k}(m)} \widetilde{p}_k \widetilde{w}_k^2)/(\sum_{k\in\mathbf{k}(m)} \widetilde{w}_k)^2}. \end{aligned} \tag{14}$$

For graphs with multiple output channels, a single quality metric Q can be defined as weighted average of individual output SNRs:

$$Q = \frac{\sum_{m\in\mathbf{O}}(q_m \cdot SNR_m)}{\sum_{m\in\mathbf{O}} q_m}, \tag{15}$$

where $\mathbf{O}$ is the set of output channels and q_m describe their relative quality contributions.

5.3 Scheduling Formulation

We further aim to derive a static schedule that maximizes quality. We formulate the scheduling as determining the start times $t_s(a)$ of actors within one graph iteration relative to the beginning of the period. In Equations 14 and 15, $\widetilde{p}_k$ depend on individual channels' delivery probabilities p_j, which following Equation 2, in turn depend on their latency budgets d'_j. In acyclic graphs, latency budgets can be calculated from the differences in start times $t_s(a)$ of a channel's producer and consumer actors minus the WCET $e'(a)$ of the producer actor as follows:

$$d'_j = \ t_s(v_j) - (t_s(u_j) + e'(u_j)), \quad j \in \mathbf{F}, \tag{16}$$

where u_j and v_j are the producer and consumer actors of channel j, respectively, and $\mathbf{F}$ is the set of (forward) channels of the graph.

In case of cyclic graphs, the budget of channels with no initial tokens can be calculated similarly. However, since tokens generated by the producer of a backedge channel with an initial token are received by the consumer in the next iteration, the delay budget for such backedges $(u_j, v_j) \in \mathbf{B}$ also depends on the period τ of the graph:

$$d'_j = \ (t_s(v_j) + \tau) - (t_s(u_j) + e'(u_j)), \quad j \in \mathbf{B}. \tag{17}$$

Using Equations 16 and 17, Q from Equation 15 can be expressed as a function of a set of start times $\mathbf{t_s} = \{t_s(a_0), t_s(a_1), \ldots\}$ and a period τ. Consequently, to derive the optimal schedule under given latency and period constraints, we can formulate scheduling as an optimization problem with start times as decision variables, $Q(\mathbf{t_s}, \tau)$ as maximization goal, and producer-consumer dependency, latency and period constraints:

$$\begin{aligned} \underset{\mathbf{t}_s}{\text{maximize}} \quad & Q(\mathbf{t}_s, \tau) \\ \text{subject to} \quad & t_s(v_j) \geq t_s(u_j) + e'(u_j), \ \forall j \in \mathbf{F}. \\ & t_s(v_j) + \tau \geq t_s(u_j) + e'(u_j), \ \forall j \in \mathbf{B}. \\ & (t_s(a') + e'(a')) - t_s(a) \leq l'(a, a'), \ \forall (a, a') \in \mathbf{L}, \end{aligned} \tag{18}$$

where $l'(a, a')$ is a latency constraint between actor pair (a, a').

Note that, due to Equation 2, the optimization problem of Equation 18 is in general non-linear and non-convex. However, since variables are continuous and the cost function is differentiable and expressed in closed form, the optimization can be solved via iterative numerical approaches. This requires repeatedly evaluating the quality function and computing its gradient with respect to changes in start times. As shown in Equations 14 and 15, accounting for the noise-free signal value ($\widetilde{w}_k$) and delivered signal power ($\widetilde{p}_k \widetilde{w}_k^2$) contributions of every path to every output in each such iteration would normally incur exponential complexity. In practice, we can compute the quality function and its gradient by accumulating the partial contributions of different paths at each intermediate channel and propagating signal and power values down the graph. This can be achieved by traversing the graph in a breadth-first manner with linear complexity visiting each channel only once. In case of acyclic graphs, the precedence graph for one iteration is traversed in breadth-first order. In case of cyclic graphs, feedback through backedges has to be accounted for through repeated traversals until either a fixed-point or sufficient convergence is reached.

Algorithm 1 gives the pseudocode for evaluation of Q and its gradient. The algorithm takes a T-RADF graph G, start times $\mathbf{t_s}$, period τ, the cumulative distribution functions (CDFs) $\mathbf{F_D}$ and probability density functions (PDFs) $\mathbf{f_D}$ of channel NDDs, the loss rates μ associated with each channel, actor WCETs $\mathbf{e}'$, and the number of iterations for cyclic graphs I (set to $I = 1$ otherwise) as input. The algorithm first initializes variables that maintain the partial noise-free signal values s_j, the partial delivered signal powers $P_{s_j}^{path}$ and their gradients $\nabla P_{s_j}^{path}$ at each channel j to zero. Note that signal values s_j are independent of actor start times. To compute the gradient of the quality function we thus only need to maintain the $P_{s_j}^{path}$ gradients. Then, each channel's p_j's and their derivatives $\widehat{p_j}$'s with respect to d'_j are computed from start times using Equations 16 and 17. Finally, signal values and powers of graph inputs and backedges are set to 1.0. Note that as shown in Equation 14, initial values will later cancel out when computing final SNR and Q, i.e. their choice does not matter.

Following the initialization, the algorithm starts performing iterations of breadth-first search (BFS) traversals over the channels in the graph. In case of cyclic graphs, traversals are performed in precedence graph order, i.e. starting from input and back edges in each iteration. For each channel j, partial s_j, $P_{s_j}^{path}$ and $\nabla P_{s_j}^{path}$ are computed by iterating over all input channels of the channel's producer actor u_j, i.e. all channels j' where $v_{j'} = u_j$. Signal values are propagated from input channel j' to channel j by adding the signal value of j' multiplied by the weight $w_{j,j'}$ to s_j. Likewise, delivered signal power is calculated by multiplying the power of j' with the squared weight $w_{j,j'}^2$ and the delivery probability $p_{j'}$ of j'.

Computation of power gradients with respect to actor start times is more involved. In general, the partial derivative of path k's power contribution $\widetilde{p_k}\widetilde{w}_k^2$ with respect to the start time $t_s(a_z)$ of an actor a_z in that path can be derived as:

$$\frac{\partial \widetilde{p_k}\widetilde{w}_k^2}{\partial t_s(a_z)} = \widetilde{w}_k^2 \frac{\partial \widetilde{p_k}}{\partial t_s(a_z)} = \widetilde{w}_k^2 \Big(\prod_{j \neq x,y} p_j\Big) \times \frac{\partial p_x p_y}{\partial t_s(a_z)}, \tag{19}$$

where x and y are the input and output channels of actor a_z in path k, respectively. According to Equations 16 and 17, gradients with respect to start times of an actor translate into positive or negative gradients with respect to the delay budgets of its output or input channels, respectively. In other words:

$$\begin{aligned} \frac{\partial \widetilde{p_k}\widetilde{w}_k^2}{\partial t_s(a_z)} &= \widetilde{w}_k^2 \Big(\prod_{j \neq x,y} p_j\Big) \cdot \Big(p_x \frac{\partial p_y}{\partial t_s(a_z)} + p_y \frac{\partial p_x}{\partial t_s(a_z)}\Big) \\ &= \widetilde{w}_k^2 \Big(\prod_{j \neq x,y} p_j\Big) \cdot (-p_x \widehat{p_y} + p_y \widehat{p_x}), \end{aligned} \tag{20}$$

where $\widehat{p_x}$ and $\widehat{p_y}$ are given as $\partial p_x / \partial d'_x$ and $\partial p_y / \partial d'_y$, respectively. This shows that the input channel x adds a positive term to the partial derivative with respect to its consumer's (i.e., a_z's) start time. Likewise, the output channel y adds a similar negative term with respect to what is in this case its producer. Further refactoring these terms:

$$\begin{aligned} \frac{\partial \widetilde{p_k}\widetilde{w}_k^2}{\partial t_s(a_z)} &= -\widetilde{w}_k^2 \Big(\prod_{j \neq x,y} p_j\Big) \cdot p_x \widehat{p_y} + \widetilde{w}_k^2 \Big(\prod_{j \neq x,y} p_j\Big) \cdot p_y \widehat{p_x} \\ &= -\Big(\prod_{j \neq y} w_j^2 p_j\Big) \cdot w_y^2 \widehat{p_y} + \Big(\prod_{j \neq x} w_j^2 p_j\Big) \cdot w_x^2 \widehat{p_x}. \end{aligned} \tag{21}$$

As such, the positive term added by channel x to path k's power derivative with respect to a_z's start time is given by the multiplication of $w_x^2 \widehat{p_x}$ with the power contribution of path k excluding x. The output channel y adds a similar negative term. Looking at this from a channel perspective, we can conclude that each channel j in path k contributes a negative and positive term to the derivative of path k's power contribution with respect to j's producer (u_j) and consumer (v_j) start times, respectively. During graph traversal, we in turn accumulate partial contributions to derivatives across all paths with respect to all start times at each channel. Contributions are propagated by adding the partial terms contributed by each channel to its producer and consumer derivatives, while scaling previously accumulated partial contributions w.r.t. other start times by $w_j^2 p_j$. Following this observation, the graph traversal algorithm propagates gradients of channels j' by first defining a temporary variable $\overrightarrow{\delta P}$ that holds the gradient components of channel j' each scaled by $w_{j,j'}^2 p_{j'}$. The algorithm then adjusts the partial derivatives w.r.t. j''s producer and consumer by adding terms $w_{j,j'}^2 \widehat{p_{j'}}$ multiplied by the partial power contribution at j'. Finally, $\overrightarrow{\delta P}$ is added to the gradient of channel j.

At the end of each traversal, the algorithm resets s_j, $P_{s_j}^{path}$ and $\nabla P_{s_j}^{path}$ of all channels that are not inputs or backedges for the next iteration. Once all iterations are finished, the algorithm computes and returns the final quality estimate Q and its gradient ∇Q as the weighted average over all graph outputs according to Equation 15.

For a graph with n actors and m channels, the total space complexity of this algorithm is $O(n \cdot m)$. Since for each channel j, the algorithm loops over all incoming channels j' of the producer actor ($v_{j'} = u_j$), the time complexity is equal to the average indegree of actors multiplied by the number of channels m in the graph. In directed graphs, each channel contributes one input, and the average indegree is m/n. As such, the time complexity of the algorithm is $O(I \cdot m^2/n)$.

Given an efficient way to compute quality estimates and their gradients, we can apply a numerical optimization algorithm to solve the problem from Equation 18. As mentioned above, due to the probabilistic distribution of random variables, $Q(\mathbf{t_s}, \tau)$, is neither linear nor convex. We apply a constrained trust region (CTR) [5] algorithm to maximize Q. CTR is a gradient-based algorithm that enables minimizing a generic, non-linear function subject to constraints. The termination condition for this method is based on the norm of the Lagrangian gradient and it stops once a stationary point for the Lagrangian has been found. Similar to other gradient-based methods, the CTR algorithm can get stuck in local optima, which depends on the starting condition and can be addressed by restarting optimizations from different points.

5.4 Buffer Sizing

To derive the size of the reorder buffer of a consumer of channel j, it is enough to note that in steady-state periodic execution, during $\Delta t_j = t_s(v_j) - t_s(u_j)$, actor u_j can fire a maximum of $\Delta t_j/\tau$ times. Additionally, the jitter of the channel c_j can cause tokens from $(D_j^{max} - D_j^{min})/\tau$ earlier firings to arrive during this time, where D_j^{min} and D_j^{max} stand for minimum and maximum delay of channel j and can be approximated by evaluating $F_{D_j}^{-1}$ for small and large enough probabilities, respectively. Combined, the reorder buffer size b_j of v_j can be computed as:

$$b_j = \lceil (\Delta t_j + (D_j^{max} - D_j^{min}))/\tau \rceil. \tag{22}$$

Note that tokens not fitting into the buffer due to the approximation in D_j^{min} and D_j^{max} will translate into empty tokens.

6 EXPERIMENTS AND RESULTS

We implemented our scheduling approach in Python using the *NetworkX* library and *SciPy* for optimization. We have released our framework in open-source form at [16]. We perform five iterations of graph traversals when evaluating the quality model for cyclic graphs. The starting condition for all optimizations is a baseline schedule with uniform allocation that equally partitions latency budgets across all channels of the paths between constrained actor pairs. In case of multiple paths constraining a channel, the minimum budget is chosen. We set the threshold on the Langrangian gradient norm for optimization to 10^{-5}.

Algorithm 1 Evaluation of Q and ∇Q.

```
1: procedure EVALQ($G$, $\mathbf{t_s}$, $\tau$, $\mathbf{F_D}$, $\mathbf{f_D}$, $\mu$, $\mathbf{e}'$, $I$)
2:     Initialize $\forall j : s_j \leftarrow 0$
3:     Initialize $\forall j : P_{s_j}^{path} \leftarrow 0, \nabla P_{s_j}^{path} \leftarrow [0.0, \ldots, 0.0]$
4:     Initialize $\forall j : p_j \leftarrow \mu_j \times F_{D_j}(d_j'(\mathbf{t_s}, \tau, \mathbf{e}`))$
5:     Initialize $\forall j : \widehat{p_j} \leftarrow \partial p_j / \partial d_j' \leftarrow \mu_j \times f_{D_j}(d_j'(\mathbf{t_s}, \tau, \mathbf{e}'))$
6:     Initialize $\forall j \in \mathbf{I}(G) \cup \mathbf{B}(G) : s_j, P_{s_j}^{path} \leftarrow 1.0$
7:     for $i$ in 1, 2, ..., $I$ do
8:         for channel $j$ in BFS_EDGE_TRAVERSAL($G$) do
9:             for all channels $j', v_{j'} = u_j$ do
10:                $s_j$ += $w_{j,j'} \cdot s_{j'}$
11:                $P_{s_j}^{path} + = p_{j'} \cdot w_{j,j'}^2 \cdot P_{s_{j'}}^{path}$
12:                $\overrightarrow{\delta P} \leftarrow w_{j,j'}^2 p_{j'} \cdot \nabla P_{s_{j'}}^{path}$
13:                $\delta P_{u_{j'}} - = w_{j,j'}^2 \widehat{p_{j'}} \cdot P_{s_{j'}}^{path}$
14:                $\delta P_{v_{j'}} + = w_{j,j'}^2 \widehat{p_{j'}} \cdot P_{s_{j'}}^{path}$
15:                $\nabla P_{s_j}^{path}$ += $\overrightarrow{\delta P}$
16:            end for
17:        end for
18:        $\forall j \notin \mathbf{I}(G) \cup \mathbf{B}(G) : s_j \leftarrow 0$
19:        $\forall j \notin \mathbf{I}(G) \cup \mathbf{B}(G) : P_{s_j}^{path} \leftarrow 0, \nabla P_{s_j}^{path} \leftarrow [0.0, \ldots, 0.0]$
20:    end for
21:    $W \leftarrow \sum_{j \in \mathbf{O}(G)} (q_j)$
22:    $Q \leftarrow \frac{1}{W} \sum_{j \in \mathbf{O}(G)} \left( q_j / \left( 1 - (P_{s_j}^{path} / s_j^2) \right) \right)$
23:    $\nabla Q \leftarrow \frac{1}{W} \sum_{j \in \mathbf{O}(G)} \left( (q_j \cdot \nabla P_{s_j}^{path}) / \left( s_j \cdot \left( 1 - (P_{s_j}^{path} / s_j^2) \right) \right)^2 \right)$
24:    return $Q$, $\nabla Q$
25: end procedure
```

We evaluate our approach on a set of random graphs and a distributed neural network application. We generate small, medium and large sets of acyclic and cyclic random graphs with 10, 50 and 100 nodes using sdf3 [25]. To enable direct comparison of acyclic and cyclic graphs, we first generate cyclic graphs and then remove backedges to create corresponding acyclic graphs. To assure that all acyclic graphs are (weakly) connected, we remove any cyclic graphs that have multiple components without backedges. Each graph set consists of 100 randomly generated graphs, where we add self-loops to all actors to enforce their sequential execution while marking source and sink nodes i.e. nodes with in and out degrees of zero as input and output actors, respectively. We set the WCET of all actors to 10ms. For quality modeling, we assign a single weight to forward and backward edges randomly in intervals [0.1,1] and [0.01,0.1], respectively.

For modeling of networks and network parameters, we set bandwidth of all links to 5Mbps. Between each host pair in the network, we assume an average loss rate μ of 1% and a Gamma distribution of network delays with shape parameter (α) and scale factor (β) randomly chosen in intervals [2,3] and [1,1.5]ms, respectively, similar to the work in [4]. To enable quantitative comparison of latency constraints, we introduce a constraint factor ρ. For a given ρ, we derive latency constraints on all input-output pairs $(a_i, a_o), \forall i \in \mathbf{I}, o \in \mathbf{O}$ of a graph as

$$l'(a_i, a_o) = l_{min}(a_i, a_o) + \rho \times (l_{max}(a_i, a_o) - l_{min}(a_i, a_o)), \tag{23}$$

where $l_{min}(a_i, a_o)$ and $l_{max}(a_i, a_o)$ are the minimal and maximal latencies between a_i and a_o when all channels j in the graph have delay budgets that correspond to a p_j of 0.001 and 0.999, respectively. We define the period constraints similarly as $\tau' = \tau_{min} + \rho \times (\tau_{max} - \tau_{min})$, where τ_{min} and τ_{max} are the minimal and maximal periods of the graph with channel p_j of 0.001 and 0.999, respectively.

To verify our approach, we developed a simulation model for mapped and scheduled T-RADF graphs in OMNET++ [27]. Token types for simulation of random graphs are 8-byte doubles, and inputs were chosen as sinusoidal signals with the same offset and amplitude but different phase offsets based on the input index selected from 10 possible options. Our simulation model has support for three different replacement functions:

- R_{static}: replaces empty tokens with zeros
- R_{last}: replaces empty tokens with the last received value
- R_{avg}: replaces with the running average of received values

Note that combinations of these replacement functions and inputs satisfies the assumptions made earlier in Section 5.1. Through emulation of lossless execution of a graph, the simulation model also supports calculating reference values, in addition to actual values, and, therefore, SNR.

In the following, we first describe how we verified our quality model, then optimization results for random graphs. Finally, we demonstrate the effects of quality/latency-aware scheduling on a distributed neural network application.

6.1 Fidelity of Quality Model

To see how well estimated SNR bounds by our quality model track measured SNRs, we chose a subset of 10 graphs from each set of 100 graphs and generated 100 random schedules with ρ randomly chosen in the interval [0.1,0.9]. We generate a random schedule for a given ρ by randomly partitioning latency budgets of input-output pairs across forward edges along each path between the pair and use the resulting link budgets to determine the start times.

For optimization, we are concerned about the relative fidelity, but not absolute accuracy of the quality model. To quantify the correlation between estimated and measured SNRs, we use Spearman's and Pearson's correlation coefficients that measure monotonic and linear correlation, respectively. As Table 3 shows, average correlation is very high and the model tracks SNR well across cyclic and acyclic graphs of different sizes. Spearman coefficients are slightly higher than Pearson's. They only measure monotonicity of the relationship between estimated and measured SNR, but this is what matters for optimization. Note that without feedback, SNR estimates for acyclic graphs are generally more conservative, i.e. lower than for cyclic ones. Thus, estimation errors are relatively higher.

6.2 Scheduling Optimizations

In the following, we discuss optimization results for random graphs, where we compare the improvements in estimated and measured SNR with three different replacement policies for various ρ.

6.2.1 Optimization Summary. Figure 6 shows the percentages of cyclic and acyclic graphs whose average measured SNR under different replacement policies improve, stay constant and decrease as result of optimization, for various graph sizes and ρ. In the majority of cases, measured SNR improves and, in very few cases, optimization fails to find a better schedule and hence SNR does not change. However, there are also cases where the optimization may find a better schedule,

Table 3. Correlation of estimated and measured SNRs.

type	size	Spearman's			Pearson's		
		R_{static}	R_{avg}	R_{last}	R_{static}	R_{avg}	R_{last}
acyclic	s	0.853	0.928	0.945	0.861	0.904	0.93
	m	0.908	0.914	0.95	0.898	0.911	0.944
	l	0.882	0.901	0.939	0.885	0.906	0.948
cyclic	s	0.975	0.98	0.981	0.937	0.945	0.952
	m	0.924	0.925	0.917	0.906	0.918	0.912
	l	0.9	0.901	0.935	0.89	0.882	0.92

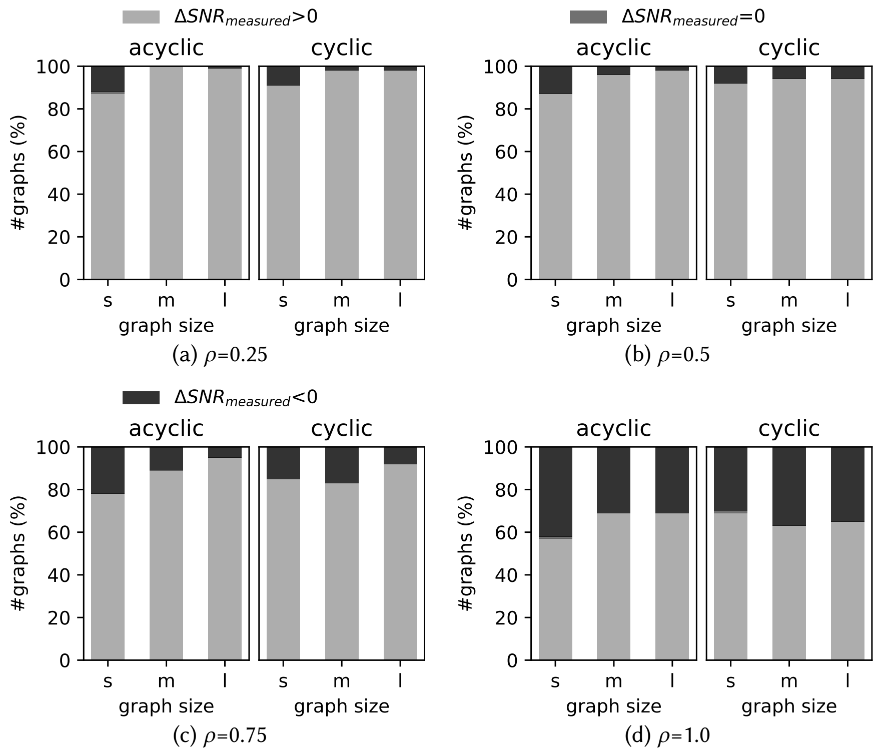


Fig. 6. Breakdown of optimization results that improve, do not affect or decrease measured SNR vs. the baseline.

but measured SNR decreases. This happens more frequently in cases with large ρ. With relaxed latency constraints, the improvements that can be achieved by the optimization are small and therefore a small error in model fidelity can cause the measured SNR to decrease. We believe that an improved quality model that accounts for $P_{s_m}^{joint}$ and does not assume inputs to be comparable ($s_{in(k)}[i] \neq s_{in(k')}$) can address this problem. We will explore more detailed quality models in future work.

6.2.2 Optimization Results. Figure 7 further details the average ΔSNR achieved by optimizations across various replacement policies. Results show that the average change in the measured SNR is always larger than that of estimated SNR, which further indicates that conservative quality estimates

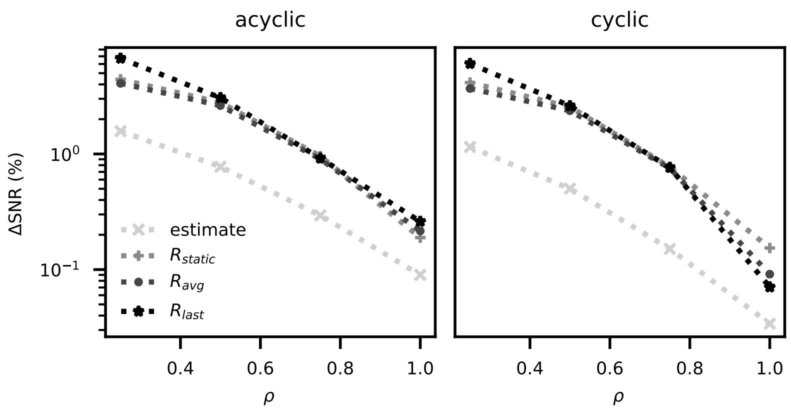


Fig. 7. Average SNR improvement across replacement policies.

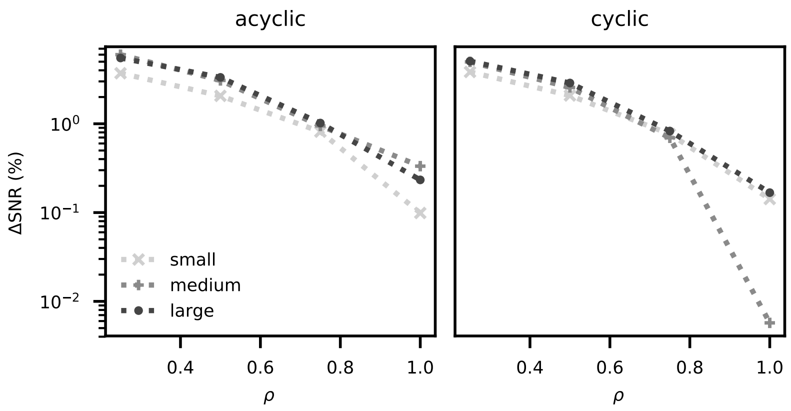


Fig. 8. Average SNR improvement for different graph sizes.

provided by our analysis hold. As can be seen, optimization benefits smaller ρ more. With large ρ, already with a uniform distribution, channels have $p_j \approx 1$, i.e. experience few losses and there is little room for optimizations. Among the measured SNRs, for most values of ρ, all replacement policies benefit equally from the optimization under decreasing ρ. However, for $\rho = 0.25$, R_{last} improves more than R_{static} and R_{avg}. This is due to the fact that R_{last} depends on the previous values to function well, and with small p_j, it cannot provide a good estimate and optimization improves its SNR by allowing it to function better.

To investigate the effects of graph size on ΔSNR, we show the average improvement in measured SNR across three replacement policies for various graph sizes in Figure 8. As this figure shows, in general, different graph sizes are very close to each other and therefore we can conclude that graph size is not of much relevance. However, in case of small acyclic and medium cyclic graphs, due to the larger number of $\Delta SNR < 0$ cases with $\rho = 0.75$ and $\rho = 1.0$, improvement in SNR is less compared to other graphs.

6.2.3 Optimization Runtime. To measure optimization overhead, we collected the average execution time of the trust-region constrained optimization for cyclic and acyclic graphs under various ρ on Intel Core i7-920. As Figure 9 shows, execution time is larger for smaller ρ since finding the optimal

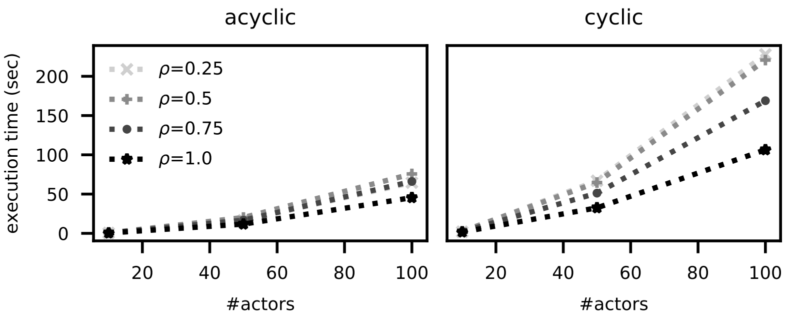


Fig. 9. Average time for optimal scheduling.

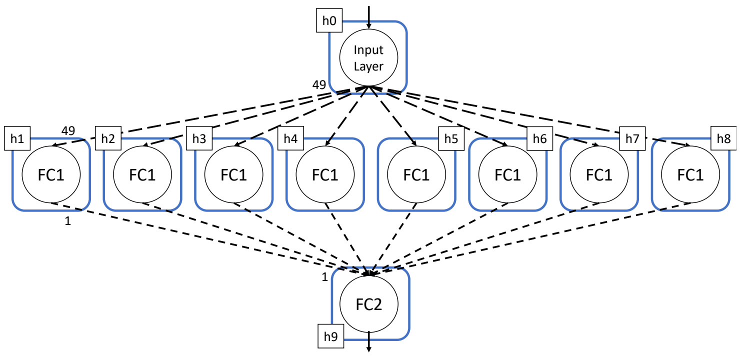


Fig. 10. Partition/mapping of digit classification neural network.

schedule becomes harder under tight latency constraints. Furthermore, as expected, it takes more time to schedule cyclic graphs compared to acyclic ones as they require multiple iterations.

6.3 Distributed Neural Net

To examine the effects of quality/latency-aware scheduling on a real application with non-linearities, we developed a simulation model of a two-layer neural network for classification of handwritten digits in the MNIST dataset [12] as a representative example of a typical image classification network architecture. We base our model on the C++ implementation provided by [24]. We distribute this neural network by mapping each layer to a different host and further partitioning the hidden layer across eight hosts. Figure 10 shows the resulting T-RADF graph along with its mapping. In this graph, a single input layer actor tiles the image consisting of 784 integers into 49 tokens of 16 integers each and sends the tokens to a hidden fully-connected (FC1) layer. Each FC1 actor corresponds to 16 neurons and uses 49 tokens to produce 16 integers that are sent as single token. A second fully-connected layer (FC2) combines the partial results, completes the inference and assigns a label to the image. We compare the assigned labels with ground truth to measure the accuracy. WCET of all FC1 and FC2 actors are assumed to be 1s. We account for the serialization time of data by measuring it in the simulation (85ms) and assigning it as the input layer's WCET. For optimization, we approximate this graph as linear by attaching the sum of the neuron weights of each partition to its incoming links and using a static replacement function for all actors. We choose the NDD of all outgoing links of $h0$ to be a Gamma distribution with $\alpha = 2$ and $\beta = 1ms$. For the NDD of incoming links of $h9$, we use a Gamma distribution with $\alpha = 3$ and $\beta = 1.5ms$.

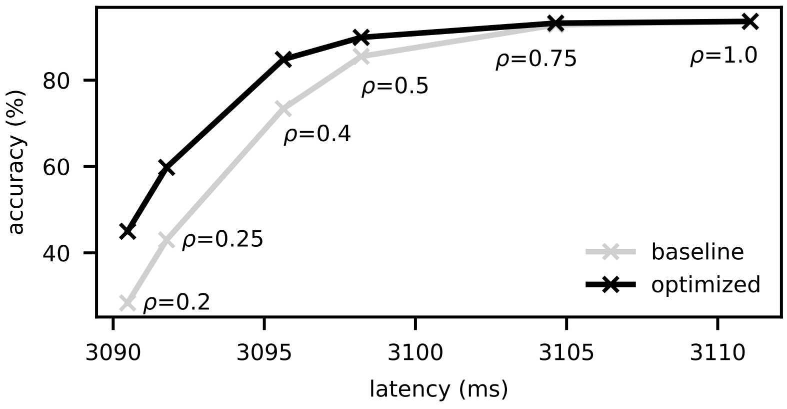


Fig. 11. Quality/latency trade-off in digit classification.

Note that these α and β represent the two extremes of the range used for experiments with random graphs. For all links, we assume μ to be 1%.

To evaluate the quality/latency trade-off, we generate baseline and optimal schedules for the graph under various latency constraints. Similar to Section 6.2, we employ a uniform latency distribution as baseline. We simulate the resulting schedules for 1000 iterations. As shown in Figure 11, at tight latency constraints, an optimized schedule can improve accuracy by up to 10%. However, as constraints are relaxed and there is less opportunity for optimization, schedules becomes similar and saturate at 94% of the accuracy of the original implementation from [24]. Overall, this example shows that our approach can achieve significant improvements in realistic applications even when they are non-linear.

7 SUMMARY, CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach for quality/latency-aware scheduling of distributed streaming applications. To the best of our knowledge, this is the first work to address the problem of quality-optimized scheduling of distributed dataflow models while providing real-time guarantees over unbounded open networks. We formalized the execution of distributed streaming real-time applications by adapting and extending existing dataflow models. To optimize the quality/latency trade-off, we developed an analytical quality model and formulated scheduling as a numerical optimization problem. Simulation of random graphs and a neural network show that our approach can improve SNR by 50% on average and recognition accuracy by up to 10% over a baseline schedule. We have released our scripts, simulation models and graphs in open-source form at [16].

Our current work is based on a number of simplifying assumptions. This includes homogeneous graph and one-actor-per-host limitations. These restrict the set of applications, application-to-network mappings and communication architectures that can be modeled and optimized using our approach. As discussed in Section 5.1, the assumption of one actor per host can be satisfied by statically scheduling multiple actors mapped to the same host into a super-actor used for our analysis. This will, however, make super-actors generally non-homogeneous. We plan to add support for direct analysis of non-homogeneous graphs with more complex communication patterns and integrated mapping optimizations in future work. In the process, we plan to improve quality models and optimization heuristics.

Our work currently performs a strictly static scheduling assuming per-link NDDs that are profiled offline and independent. In non-stationary networks, delays can become correlated. This can be addressed in a real-world deployment by dynamically measuring NDDs and regenerating the schedule at regular intervals. Similarly, statically computed schedules can be optimized at runtime by dynamically adjusting timeouts and delay budgets in response to instantaneous variations in network delays. We plan to develop a corresponding runtime system for deployment of T-RADF models.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their valuable comments and helpful suggestions. This work was partially supported by NSF under grant CNS-1421642 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1421642&HistoricalAwards=false).

REFERENCES

[1] J. Beal, D. Pianini, and M. Viroli. 2015. Aggregate Programming for the Internet of Things. *Computer* 48, 9 (2015), 22–30.

[2] C.J. Bovy, H.T. Mertodimedjo, G. Hooghiemstra, H. Uijterwaal, and P. Van Mieghem. 2002. Analysis of End-to-end Delay Measurements in Internet. In *The Passive and Active Measurement Workshop-PAM*, Vol. 2002. sn.

[3] Junguk Cho, Hyunseok Chang, Sarit Mukherjee, TV Lakshman, and Jacobus Van der Merwe. 2017. Typhoon: An SDN Enhanced Real-Time Big Data Streaming Framework. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 310–322.

[4] A. Coluccia and F. Ricciato. 2018. On the estimation of link delay distributions by cumulant-based moment matching. *Internet Technology Letters* 1, 1 (2018).

[5] A.R. Conn, N.I.M. Gould, and P.L. Toint. 2000. *Trust Region Methods*. Vol. 1. Siam.

[6] J. Dean and S. Ghemawat. 2010. MapReduce: a flexible data processing tool. *Commun. ACM* 53, 1 (2010), 72–77.

[7] Pascal Fradet, Alain Girault, Leila Jamshidian, Xavier Nicollin, and Arash Shafiei. 2018. Lossy Channels in a Dataflow Model of Computation. In *Principles of Modeling*. Springer, 254–266.

[8] S. Francis and A. Gerstlauer. 2017. A Reactive and Adaptive Data Flow Model For Network-of-System Specification. *IEEE Embedded Systems Letters* (2017).

[9] W. Gropp, E. Lusk, and A. Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[10] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. 2013. Mobile fog: A Programming model for large-scale applications on the internet of things. In *SIGCOMM Workshop on Mobile Cloud Computing*. 15–20.

[11] V. Jacobson, R. Frederick, S. Casner, and H. Schulzrinne. 2003. RTP: A transport protocol for real-time applications. (2003).

[12] Y. LeCun, C. Cortes, and C.J. Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010), 18.

[13] E. Lee and D. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[14] Changfu Lin, Jingjing Zhan, Hanhua Chen, Jie Tan, and Hai Jin. 2018. Ares: A High Performance and Fault-Tolerant Distributed Stream Processing System. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 176–186.

[15] X. Mao, J.and Chen, K.W. Nixon, C. Krieger, and Y. Chen. 2017. MoDNN: Local distributed mobile computing system for Deep Neural Network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[16] Kamyar Mirzazad Barijough. 2019. QLA-RTS. https://github.com/SLAM-Lab/QLA-RTS. Online.

[17] S. Nastic, S. Sehic, M. Vogler, H. Truong, and S. Dustdar. 2013. PatRICIA–A Novel Programming Model for IoT Applications on Cloud Platforms. In *International Conference on Service-Oriented Computing and Applications (SOCA)*. 53–60.

[18] T.M. Parks, J.L. Pino, and E. Lee. 1995. A comparison of synchronous and cycle-static dataflow. In *Asilomar Conference on Signals, Systems and Computers*.

[19] P. Persson and O. Angelsmark. 2015. Calvin–Merging Cloud and IoT. *Procedia Computer Science* 52 (2015), 210–217.

[20] E. Pitt and K. McNiff. 2001. *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc.

[21] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. 2018. DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics. In *IEEE Conference on Computer Communications (INFOCOM)*.

[22] D.G. Schmidt and F. Kuhns. 2000. An overview of the Real-Time CORBA specification. *Computer* 33, 6 (2000), 56–63.

[23] J. Siegel and D. Frantz. 2000. *CORBA 3 Fundamentals and Programming*. Vol. 2. John Wiley & Sons New York, NY, USA:.

[24] H.T. Son. 2015. Neural Network implementation in C++ running for MNIST database. https://github.com/HyTruongSon/Neural-Network-MNIST-CPP. Online.

[25] S. Stuijk, M. Geilen, and T. Basten. 2006. Sdfˆ 3: SDF For Free. In *International Conference on Application of Concurrency to System Design (ACSD)*.

[26] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. 2014. Storm@twitter. In *International Conference on Management of Data (SIGMOD)*.

[27] A. Varga and R. Hornig. 2008. An overview of the OMNeT++ simulation environment. In *International conference on Simulation tools and techniques for communications, networks and systems & workshops*.

[28] Guolu Wang, Jungang Xu, Renfeng Liu, and Shanshan Huang. 2018. A hard real-time scheduler for Spark on YARN. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '18)*. IEEE Press, 645–652. https://doi.org/10.1109/CCGRID.2018.00096

[29] Yuankun Xue, Ji Li, Shahin Nazarian, and Paul Bogdan. 2017. Fundamental Challenges Toward Making the IoT A Reachable Reality: A Model-centric Investigation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22, 3 (2017), 53.

[30] Y. Zhao, J. Liu, and E. Lee. 2007. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 259–268.

[31] Z. Zhao, K. Mirzazad Barijough, and A. Gerstlauer. 2018. DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.