Faster, Exact, More General Response-time Analysis for NVIDIA Holoscan Applications

Philip Schowitz
The University of British Columbia
Vancouver, BC, Canada
philpns@cs.ubc.ca

Shubhaankar Sharma*
The University of British Columbia
Vancouver, BC, Canada
ssharm52@student.ubc.ca

Siddharth Balodi*
The University of British Columbia
Vancouver, BC, Canada
sbalodi@student.ubc.ca

Soham Sinha
NVIDIA
Santa Clara, CA, USA
sohams@nvidia.com

Bruce Shepherd
The University of British Columbia
Vancouver, BC, Canada
fbrucesh@cs.ubc.ca

Arpan Gujarati
The University of British Columbia
Vancouver, BC, Canada
arpanbg@cs.ubc.ca

***Abstract*—We present a scalable method to compute exact worst-case end-to-end latency in applications built on the NVIDIA Holoscan SDK, a framework increasingly adopted for soft real-time ML workloads in medical devices, surgical instruments, and robotics. Holoscan applications are structured as directed acyclic graphs of non-preemptible task threads (*operators*) connected by FIFO queues, where execution depends not only on input availability but also on *downstream* buffer capacity – an atypical backpressure mechanism not captured by standard dataflow or middleware models. Existing analyses either lack convergence guarantees or rely on restrictive assumptions (e.g., fixed execution times, unit-sized buffers), resulting in overly conservative bounds.**

We show that Holoscan's scheduling semantics can be faithfully reduced to homogeneous synchronous dataflow graphs (HSDFGs), enabling exact end-to-end latency analysis. Building on this insight, we introduce a dynamic algorithm that computes tight upper bounds on response time across infinite input streams under variable task runtimes and arbitrary buffer sizes. We prove its correctness and convergence, and demonstrate that it outperforms HSDFG model checking with UPPAAL in runtime while avoiding the pessimism of prior Holoscan-specific analyses. Experiments on real Holoscan applications from NVIDIA HoloHub and large synthetic graphs confirm its scalability and precision.

I. Introduction

Machine Learning (ML) now underpins a wide range of domains—from the internet and mobile devices to robotics and medicine. This shift is driven by powerful accelerators like GPUs and TPUs [26], advanced models such as DNNs and LLMs, and large public datasets like ImageNet and Medical ImageNet. Equally important are ML frameworks and middleware, which simplify development by abstracting low-level software and hardware details. As Google researchers note, ML code is only a small part of real-world systems, which are dominated by complex supporting infrastructure [30].

We consider one such ML-focused sensor processing framework, the NVIDIA Holoscan SDK [13]. Originally developed for the medical device industry, Holoscan is now gaining adoption in robotics and other edge computing domains [12]. In soft real-time applications, it is not enough to achieve high average performance; predictable worst-case behavior is also essential. For instance, in digital endoscopy, Holoscan must deliver high throughput for real-time visual feedback while avoiding timing anomalies that could result in incorrect annotations due to misaligned sensor data. To address this, our objective is to *statically* estimate the worst-case end-to-end processing time of data streams in Holoscan-based applications.

Holoscan currently provides a Data Flow Tracking module [5] to help users evaluate end-to-end performance. However, it is unclear whether, and when, this tool converges to a reliable estimate of worst-case latency. Schowitz et al. [29] propose an approximate yet safe response-time analysis by modeling Holoscan applications as conditioned DAGs: vertices represent processing tasks, edges model data buffers, and execution depends on both input availability and output buffer capacity (Holoscan uses the latter dependency to implement *backpressure*, similar to TCP, but uniquely allowing static configuration). Their approach assumes fixed task execution times and unit-sized buffers, yielding conservative bounds that may force designers to accept reduced throughput.

Prior response-time analyses for ROS middleware [4] do not apply, as Holoscan diverges from ROS's publish-subscribe model and incorporates downstream conditions. Likewise, schedule abstraction graph methods [24] are inapplicable: they assume known job release times, whereas Holoscan tasks can be triggered dynamically. Holoscan's stream processing model more closely aligns with synchronous dataflow graphs (SDFGs) [22]. Yet, despite extensive literature, we found only one relevant prior work that applies directly. Specifically, we reduce Holoscan's analysis to the latency analysis of homogeneous SDFGs via timed automata [20], but encounter prohibitive model-checking costs at realistic scales. Constraint solvers like Google OR-Tools' CP-SAT [17] face similar issues: they can only bound latency over finite traces (e.g., the first 50 inputs) and remain computationally expensive.

We propose a novel worst-case end-to-end response-time

*Equal contribution

2576-3172/25/$31.00 ©2025 IEEE
DOI 10.1109/RTSS66672.2025.00018

TABLE I
BASELINES VS. OUR ALGORITHM FOR THE MULTIAI APPLICATION DESCRIBED IN FIG. 1. VET IMPLIES SUPPORT FOR VARIABLE EXECUTION TIMES. VQS IMPLIES SUPPORT FOR VARIABLE QUEUE SIZES.

	RTA [29]	SDFG [20]	SAT [17]	Paper
Runtime	51 μs	6055 s	5.58 s	136 ms
Pessimism	20.5%	0%	0%	0%
Safe	✓	✓	?	✓
Exact	✗	✓	?	✓
vET	✗	✓	✓	✓
vQS	✗	✓	✓	✓

analysis for NVIDIA Holoscan applications. Our algorithm is *faster* and *scales better* than Kuiper and Bekooij's [20] exact latency analysis of homogeneous SDFGs. It is also *more general* than Schowitz et al.'s [29] Holoscan-specific method, as it does not assume fixed execution times or unit-sized buffers. Crucially, our approach provides *exact* or *tight* upper bounds on worst-case response time (WCRT) and, unlike testing or solver-based techniques, guarantees correctness across all infinite execution traces. Table I summarizes the key differences between our approach and existing baselines.[1]

Following Schowitz et al. [29], we evaluate our analysis using applications from NVIDIA HoloHub [9], a community-driven repository of Holoscan applications. Our results show that the analysis of these apps completes within milliseconds, whereas existing methods often require seconds or fail to scale. Execution time parameters were obtained by profiling these applications on the NVIDIA IGX Orin platform [10], the primary target for Holoscan deployment. We also evaluate our method's scalability on large synthetic graphs.

We organize the paper as follows. §II introduces Holoscan's execution model and sets up the WCRT problem on Holoscan DAGs. §III presents our first technical contribution: a reduction of this problem to homogeneous SDFGs, enabling the use of Kuiper and Bekooij's method—an overlooked connection in prior work. §IV presents our main contribution: a dynamic algorithm that symbolically derives exact WCRT. §V proves its correctness and termination. §VI evaluates our method on real and synthetic applications. §VII concludes with some discussion on related work and future directions.

II. PROBLEM STATEMENT

Sinha et al. [31] and Schowitz et al. [29] describe the NVIDIA Holoscan SDK and its programming model in detail. Here, we provide a brief summary. Holoscan applications

[1]We used Google OR-Tools' CP-SAT solver [17] to encode all Holoscan SDK conditions, treating each operator's execution time as a decision variable. While the solver yields results with no noticeable pessimism in our example, it is marked with a '?' in the *Safe* and *Exact* rows of Table I. This is because, unlike our algorithm (and more akin to testing), it must be instantiated for a fixed, finite number of iterations, as each operator execution becomes a separate decision variable. Consequently, there is no guarantee that the computed WCRT holds across all possible (infinite) traces. Hence, this approach lacks formal safety or exactness guarantees and remains significantly more computationally expensive than our method.

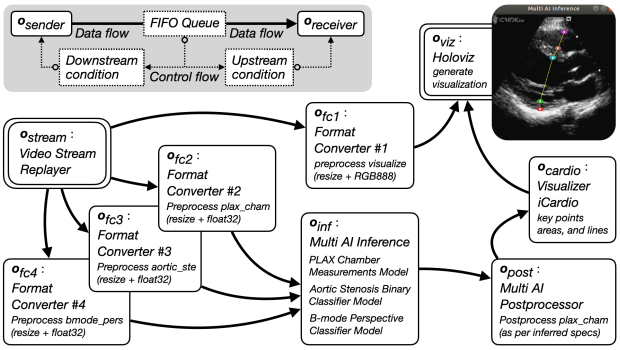

Fig. 1. We reproduce the workflow of the Multi AI Ultrasound application from NVIDIA HoloHub [9]. The pipeline processes ultrasound video input through parallel preprocessing stages, preparing frames for multiple AI models and rendering real-time visualizations with HoloViz (as shown in the sample output). GXF-internal objects, such as FIFO queues and scheduling conditions that apply to every edge, are shown inside the grey box.

consist of multiple *operators* that perform recurring tasks such as sensor data processing and image rendering. See Fig. 1 for an example Multi AI Ultrasound application. Operators communicate via input (receiver) and output (transmitter) ports, which are connected through FIFO queues managed by Holoscan's execution backend, the Graph Execution Framework (GXF) [8]. These queues abstract GXF's internal double-buffering mechanism to support concurrent producer-consumer behavior. While GXF implements multiple overflow handling strategies, our analysis focuses exclusively on the policy that preserves all data without any loss or drops.

The Holoscan SDK programming model raises important questions about *timing anomalies*, where local improvements can unexpectedly degrade global performance. Fig. 2 illustrates such a case: reducing the execution time of one operator through a mode change increases the application's WCRT. In linear chains, the operator with the largest execution time becomes the bottleneck. In the upper configuration, the second operator is the bottleneck, creating a queuing delay of 400 in the preceding queue, while subsequent operators incur no delay because they process data faster than the bottleneck produces it. This results in a WCRT of 1500. In the lower configuration, reducing the second operator's execution time shifts the bottleneck to the fourth operator. As a result, data may queue in every preceding stage, raising the WCRT to 1600. Additional examples of such anomalies are provided in [29]. A robust response-time analysis is therefore essential for making performance optimizations with confidence, even in the presence of anomalies.

We model a Holoscan application as a directed acyclic graph (DAG). Each node represents an operator with known best- and worst-case execution times, and each edge corresponds to a fixed-capacity FIFO queue. We assume a single source and sink operator; DAGs with multiple sources or sinks can be accommodated by using dummy operators with zero execution time. Operator execution is governed by three conditions:

C1. The *sequential-execution* condition ensures that an operator can execute only one iteration at a time.

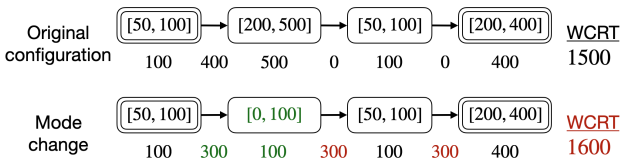

Fig. 2. Two configurations of a Holoscan application illustrating a timing anomaly in which a mode change (e.g., a multi-AI pipeline executing with only one of two models) reduces the execution time of an operator but increases the overall WCRT. Operators are shown as rectangles annotated with their execution-time ranges; FIFO queues of unit capacity are shown as arrows. The numbers below each operator and queue indicate the time spent there by an input experiencing the WCRT. The total WCRT is shown on the right.

C2. The *upstream* or *data-dependency* condition requires that all input queues contain at least one message.

C3. The *downstream* condition requires that all output queues have space to accept at least one message.

If multiple instances of an operator exist, each is modeled as a separate node in the DAG. Holoscan offers three user-level schedulers. The Greedy Scheduler [7] executes operators sequentially and is trivially analyzable. This work focuses on the Multi-Thread Scheduler [11] and the Event-Based Scheduler [6], which differ in mechanism (e.g., polling) but both allow operators to execute concurrently on separate threads. All Holoscan schedulers are *work-conserving*: an operator executes as soon as the execution conditions are satisfied.

Schowitz et al. [29] base their analysis on four assumptions:

A1. All FIFO queues have unit capacity, matching Holoscan's default and recommended setting.

A2. Operator execution times remain fixed within a run, with any variability (e.g., early completion) masked through mechanisms like busy-waiting.

A3. The platform has enough CPU cores to allow all operators to execute in parallel—an assumption that holds for typical Holoscan applications deployed on NVIDIA AGX/IGX Orin platforms with 12 cores.

A4. The DAG processes an infinite stream of inputs, meaning the source operator always has an input available when its downstream conditions are satisfied. Periodic input streams are naturally subsumed by this model.

The key contribution of our work is to relax assumptions A1 and A2 while still providing an exact and efficient response-time analysis. That is, our model retains only A3 and A4. We defer relaxing A3 to future work. Incorporating CPU scheduling into the analysis is of low priority, since even the most complex Holoscan applications are not constrained by CPU availability on typical deployment platforms. More broadly, in GPU-centric systems, CPUs increasingly serve as control planes for launching GPU inference requests. Finally, note that A4 captures a general worst-case scenario and therefore does not restrict the analysis to any particular input pattern.

With the system model and assumptions in place, we now state our goal. A Holoscan application processes a continuous stream of inputs. Each input enters at the source operator and exits at the sink operator. Due to queueing and execution delays, different inputs may take different amounts of time to complete. An input is guaranteed to be fully processed once it is accepted at the source operator due to C3 preventing buffer overflow. Our objective is to compute a tight upper bound on the WCRT, i.e., the maximum time that any successfully processed input takes from when its processing starts to when it exits.

III. Analyzing Holoscan DAGs using SDFGs

The idea of composing applications from smaller, modular components while making concurrency explicit to better exploit parallel hardware is not new. Signal processing systems [27] from the 1980s and 1990s often adopted this paradigm, leading to the development of a rich body of literature on synchronous dataflow graphs (SDFGs) [22, 28]. These focused on generating efficient dataflow schedules at compile time, enabling rigorous analysis of system behavior, particularly with respect to latency and throughput. In this section, we explore how Holoscan DAGs can be evaluated by modeling them as SDFGs.

An SDFG consists of *actors* (vertices), *channels* (edges) connecting them, and *tokens* held by the channels. An actor *fires* when sufficient tokens are available on all its input channels; after some execution time, it produces tokens on its output channels. Each channel acts as an unbounded FIFO buffer between a producing and a consuming actor. The initial number of tokens on each channel (possibly zero) influences queueing behavior and system dynamics. Fixed production and consumption rates are specified per channel. A special subclass of SDFGs, in which all rates are equal to one, is known as a *homogeneous* synchronous dataflow graphs (HSDFG)—the model of interest for Holoscan applications.

Extensive literature exists on SDFGs and their variants, but most focus on throughput and scheduling, areas less relevant to Holoscan, where core-constrained execution is not currently a practical concern. In contrast, few works address response-time analysis or the latency between actor firings. Ghamarian et al. [16] study SDFGs assuming that each actor executes at its worst-case execution time (WCET) in every iteration. They target latency minimization rather than maximization. Ali et al. [14] extract timing properties from HSDFGs for real-time scheduling but also assume fixed execution times. Moreira and Bekooij [23] allow variable execution times but provide only upper bounds on latency. Only Kuiper and Bekooij [20] satisfy both our key criteria: (i) exact worst-case latency analysis and (ii) support for variable execution times across iterations. To our knowledge, their work is the only one directly applicable to Holoscan DAG response-time analysis. Building on this, we present our first technical contribution: a reduction of the Holoscan DAG analysis problem to an HSDFG latency analysis problem, solved using Kuiper and Bekooij's [20] method. This models the system as a network of timed automata [15] and analyzes it using the UPPAAL model checker [2].

Let D denote the Holoscan DAG. Without loss of generality, we assume that all queues have equal capacity, denoted by δ. We use the term *iteration* to refer to successive executions of an operator. Let $s_i(o)$ and $f_i(o)$ denote the start and finish times of the i^{th} iteration of an operator $o \in D$, respectively. Consider

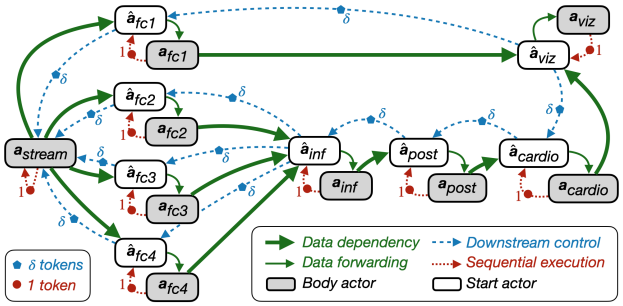

Fig. 3. The Multi AI Ultrasound workflow from Fig. 1, converted to an SDFG. Edge labels indicate initial token counts on each channel. Shaded boxes represent body actors corresponding to Holoscan operators; unshaded boxes are start actors, added to model downstream blocking. For simplicity, all queue capacities are assumed equal to δ.

two operators p (a predecessor of o) and s (a successor of o). The sequential-execution, upstream, and downstream conditions for operator execution introduced in §II impose the following constraints on the start time of operator o.

$$\text{(C1) Sequential-execution:} \quad s_i(o) \geq f_{i-1}(o) \quad (1)$$

$$\text{(C2) Upstream or data-dependency:} \quad s_i(o) \geq f_i(p) \quad (2)$$

$$\text{(C3) Downstream blocking:} \quad s_{i+\delta}(o) \geq s_i(s) \quad (3)$$

That is, Equation (1) ensures that an operator completes its previous iteration before starting the next; Equation (2) ensures that each predecessor of o has completed at least as many iterations as o has started; and Equation (3) ensures that no successor of o lags behind by more than δ iterations.

We now demonstrate an equivalence between the behavior of a Holoscan application's execution and that of an SDFG by constructing an SDFG S_D from the Holoscan DAG D. In what follows, we use the terms *operators* and *edges* when referring to the Holoscan DAG, and *actors* and *channels* when referring to the corresponding SDFG. We use a Holoscan DAG with nine operators (Fig. 1) and its corresponding SDFG (Fig. 3) as a running example to illustrate this mapping.

The three constraints introduced earlier determine when an operator is eligible to execute. Specifically, an operator may be triggered by the completion of another operator (C1 and C2), or by the start of another operator (C3). In an SDFG, however, an actor produces tokens only when it finishes firing; that is, upon completing execution. Modeling C3 is thus more challenging, as it requires an actor to trigger another based on its start, rather than finish. To enable this, we map each operator o to two actors: a *body actor* a and a *start actor* $\hat{a}$. The body actor models the actual execution of o and inherits its timing characteristics: if o has best- and worst-case execution times o^{lb} and o^{ub}, then $a^{lb} = o^{lb}$ and $a^{ub} = o^{ub}$. The start actor $\hat{a}$ has zero execution time ($\hat{a}^{lb} = \hat{a}^{ub} = 0$) and represents the moment o begins executing. The source operator only gets a body actor and no corresponding start actor. This is because no operators experience downstream blocking due to it.

To ensure that SDFG S_D faithfully models DAG D, each firing of a start actor (or the source body actor) during S_D's execution must satisfy Eqs. (1) to (3). We now explain how the channels in S_D enforce each of these constraints.

A. Sequential-execution

Each body actor a has a single tokenless incoming channel from its corresponding start actor $\hat{a}$, ensuring that it fires immediately once $\hat{a}$ fires (see the green *data forwarding* arrows in Fig. 3). To model the sequential-execution constraint, i.e., Eq. (1), we add a feedback channel from a to $\hat{a}$ (or a self-loop if o is the source operator) with one initial token. This enforces that a completes its i^{th} firing before $\hat{a}$ can initiate the $(i+1)^{\text{th}}$ firing (see the red dotted arrows in Fig. 3).

Eq. (1) is enforced through tight coupling between each start actor and its corresponding body actor. They exchange a single token over their shared channel, ensuring that neither can fire again until the other has fired. This enforces strict alternation: the start actor cannot initiate a new iteration until the body actor completes the previous one. For the source operator, this behavior is realized through a self-loop.

B. Upstream

To model data dependencies, i.e., Eq. (2), each body actor a has outgoing channels to the start actors of all operators that o connects to in the DAG. These channels deliver tokens upon completion of a's execution, enabling its successors to start once all their inputs are ready. For example, in Fig. 3, observe the bold green edge from a_{fc1} to $\hat{a}_{viz}$.

Eq. (2) ensures that an operator can start execution only when input data is available. In S_D, a body actor can only fire after its start actor fires, and the start actor requires a token from each of its incoming channels. Thus, all dependencies must have completed at least as many iterations as the current actor is about to begin. This condition is trivially satisfied for the source actor, which has no input dependencies.

C. Downstream

Finally, to capture the downstream blocking constraint, i.e., Eq. (3), we add channels from a start actor $\hat{a}$ to the start actors of the upstream operators from which o receives data (or to the body actor of the source operator if o receives data from the source). These are shown as dashed blue arrows in Fig. 3, such as the edge from $\hat{a}_{\text{inf}}$ to $\hat{a}_{\text{fc2}}$. This construction allows the start of one operator's execution to block another if downstream queue space is unavailable. The initial number of tokens on these channels corresponds to the queue capacity δ.

Eq. (3) captures backpressure due to bounded queues. As illustrated in Fig. 3, a downstream start actor limits how often its upstream counterparts can fire. For example, if $\hat{a}_{\text{fc2}}$ fires δ times without $\hat{a}_{\text{inf}}$ firing once, the token count on their connecting channel will be exhausted, preventing $\hat{a}_{\text{fc2}}$ from firing again until $\hat{a}_{\text{inf}}$ fires and releases a token. Note that a start actor $\hat{a}$ might be blocked by multiple downstream dependencies. However, regardless of which channel causes the delay, the downstream blocking constraint still holds.

D. Response-time analysis

After constructing an equivalent HSDFG from the Holoscan DAG, we use the method of Kuiper and Bekooij [20] to transform the HSDFG into a network of timed automata. This enables latency analysis via the UPPAAL model checker [2]. We reproduced their approach based on the detailed description in Kuiper's PhD thesis [19].[2] Evaluation results using this SDFG-based method are presented in §VI. While the method provides precise response-time analysis and accommodates variable execution conditions, it does not scale well to realistic Holoscan applications, underscoring the need for more scalable, domain-specific algorithms.

IV. Dynamic Algorithm

To address the limitations of prior approaches, we introduce a custom dynamic programming algorithm for computing the WCRT of a Holoscan DAG. Our algorithm can also be adapted for any HSDFG with a matching structure as described in the previous section. For an intuitive overview, consider a simple DAG composed of three operators connected in a linear chain: $D_{radar} : o_{source} \rightarrow o_{filter} \rightarrow o_{sink}$. This resembles the *Simple Radar Pipeline* from NVIDIA HoloHub [9], where streaming data from a phased array system is processed through compression, filtering, and analysis stages.

A. Trace Graph

We first introduce the *trace graph* Γ_D, a new representation for any Holoscan DAG D. Fig. 4 illustrates a subset of the trace graph $\Gamma_{D_{radar}}$ for D_{radar}. In the trace graph, each iteration of an operator is represented as a distinct node, resulting in an infinite stack of *layers*, each structurally identical to the original DAG. For example, the i^{th} layer in $\Gamma_{D_{radar}}$, denoted $\Gamma^i_{D_{radar}}$, corresponds to operators processing the i^{th} input item. Thus, operators o_{source}, o_{filter}, and o_{sink} in D_{radar} correspond to nodes n^i_{source}, n^i_{filter}, and n^i_{sink} in each layer $\Gamma^i_{D_{radar}}$. From this point forward, we refer to vertices in the trace graph as *nodes*, reserving the term *operator* for vertices in the original DAG D. Accordingly, we use the letter n to denote nodes in the trace graph, and o to denote operators in the DAG. In both cases, subscripts uniquely identify the operator, while superscripts refer to specific iterations. We define three types of edges in the trace graph, corresponding C1–C3 described earlier.

E1. Sequential-execution edges connect the $(i-1)^{\text{th}}$ iteration of an operator to its i^{th} iteration. In Fig. 4, these are shown using red-colored dotted lines, such as the edges from n^{i-2}_{sink} to n^{i-1}_{sink} and from n^{i-1}_{sink} to n^i_{sink}.

[2]We provide a brief implementation overview. Each actor in the HSDFG is mapped to a timed automaton with two states: *idle* and *executing*. Token dependencies are enforced by preventing state transitions until sufficient tokens are available, while ensuring transitions occur as soon as tokens are ready—replicating the self-timed behavior of HSDFGs. Token counts are maintained in shared data structures rather than modeled as separate automata.

To measure end-to-end latency, a dedicated timing automaton is introduced. It non-deterministically starts measuring when the source actor begins execution and stops when the corresponding sink actor completes. A clock, active only in the measuring state, records the duration between matching source and sink iterations. This nondeterminism ensures that all possible execution paths are explored, yielding a safe bound on worst-case latency.

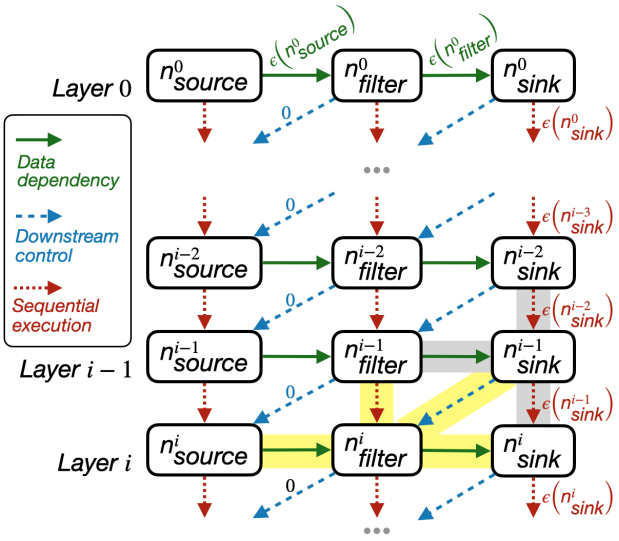

Fig. 4. A segment of the infinite trace graph $\Gamma_{D_{radar}}$ corresponding to D_{radar}. Some edges are labeled with their weights for illustration; others are shown without labels to reduce clutter, though all edges are analogously weighted according to their type. A few edges are highlighted in gray and yellow to match corresponding edges in Fig. 5 and are referenced later in its caption.

E2. Data-dependency edges follow the data flow of the original DAG. In Fig. 4, they are shown using green-colored solid lines, such as the edges from n^i_{source} to n^i_{filter} and n^i_{filter} to n^i_{sink}, reflecting data dependencies during the i^{th} iteration.

E3. *Downstream-blocking* edges connect the $(i-\delta)^{\text{th}}$ iteration of a receiving operator o_{recv} to the i^{th} iteration of each upstream producer o_{send}. Here, δ denotes the capacity of the FIFO buffer between them. Assuming $\delta = 1$, examples in Fig. 4 include blue-colored dashed edges from n^{i-1}_{filter} to n^i_{source} and from n^{i-1}_{sink} to n^i_{filter}.

Unlike in the DAG or SDFG, edges in the trace graph are weighted. Sequential-execution and data-dependency edges are assigned weights equal to the execution time of their source node, as they represent dependencies where one node's completion enables the next's start. In Fig. 4, these weights are denoted by the function $\epsilon(*)$, e.g., the edge from n^{i-2}_{sink} to n^{i-1}_{sink} has weight $\epsilon(n^{i-2}_{sink})$. If $\omega(o_{sink})$ denotes the WCET of o_{sink}, then for any i, $\epsilon(n^i_{sink}) \leq \omega(o_{sink})$. Downstream-blocking edges have zero weight, since they do not represent execution time but enforce buffer constraints by delaying upstream execution until downstream space is available. Each node $n^i_k \in \Gamma_D$ is also associated with a *start time* $s(n^i_k)$, defined as the earliest point at which that node may begin execution.

B. Response Time as Longest Path Problem

So far, we have outlined the construction of the trace graph. Its key advantage over prior work by Schowitz et al. [29] is that it explicitly models each iteration in a separate layer, allowing for fine-grained analysis of execution behavior over time. By encoding time variability in edge weights, we can formulate response time as a longest-path problem: the start time of a

node corresponds to the cost of the longest path from the initial source node n^0_{source} to that node. For example, in Fig. 4, n^i_{sink}'s start time can be recursively defined using:

$$s(n^i_{sink}) = \max\left(s(n^{i-1}_{sink}) + \epsilon(n^{i-1}_{sink}), s(n^i_{filter}) + \epsilon(n^i_{filter})\right) \quad (4)$$

Eq. (4) states that the i^{th} execution of o_{sink} can begin only after both its $(i-1)^{\text{th}}$ execution and the i^{th} execution of o_{filter} finishes. The start time is thus determined by the later of these two events. The longest-path computation guarantees that all preconditions are met before a node executes, capturing the readiness of operators and forming the foundation of our algorithm. However, Eq. (4) alone is insufficient to compute the WCRT, which is instead defined as the *difference* between the start times of two nodes in the trace graph, i.e.,

$$R = \max_i \left(s(n^i_{sink}) + \epsilon(n^i_{sink}) - s(n^i_{source})\right). \quad (5)$$

C. TG-DFS Algorithm

We now present an algorithm to compute R for Holoscan DAG D. We present the correctness and termination roofs later in §V. Suppose that the worst-case response time first occurs during iteration i. Our algorithm *TG-DFS* (see Algorithm 1) computes the response time for this iteration as:

$$R^i = s(n^i_{sink}) + \epsilon(n^i_{sink}) - s(n^i_{source}). \quad (6)$$

In particular, *TG-DFS* focuses on computing

$$R^i - \epsilon(n^i_{sink}) = s(n^i_{sink}) - s(n^i_{source}), \quad (7)$$

as $\epsilon(n^i_{sink})$ is trivially computable.

The algorithm performs a depth-first search (DFS) on the original trace graph starting from node n^i_{sink} and following *incoming* rather than outgoing edges (that is, on the *transpose* of Γ_D). This traversal yields a tree structure, illustrated in Fig. 5 for the example trace graph $\Gamma_{D_{radar}}$. The algorithm uses a *value* field to track response times during traversal.

In the simplest case, the DFS reaches n^i_{source} (which marks the beginning of iteration i), terminates along that branch, and sets the node's value to zero: $n^i_{source}.value = 0$ (Lines 10 and 11). As the DFS backtracks toward the root node n^i_{sink}, it accumulates the edge weights along the path (Line 19). The final *value* at the sink node then reflects the total cost from n^i_{source} to n^i_{sink}, corresponding to $R^i - \epsilon(n^i_{sink})$ in Eq. (6).

However, not all traversals lead to n^i_{source}, which is what allows us to directly compute Eq. (6). In the more complex case, the DFS instead reaches only an ancestor of n^i_{source}. Lemma 1 (§V-A) proves why this is guaranteed. For such cases, the algorithm applies the following optimization to estimate Eq. (6). We define an *ancestor set* as the set of all nodes in the trace graph that have a path to n^i_{source}. When the algorithm encounters a node in this set during traversal, it terminates the search at that point (Line 12). Upon termination, the algorithm invokes the function $path\_specific\_value(c, n^i_{source})$ (Line 13), which enforces a subtle condition that is necessary for the algorithm to give an exact result. Intuitively, $path\_specific\_value(c, n^i_{source})$ determines whether the ancestor has a path to the source beginning with a sequential-execution or a data dependency

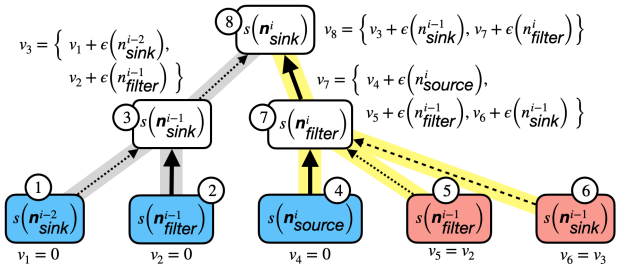

Fig. 5. The traversal tree constructed by the *TG-DFS* algorithm when analyzing D_{radar}. Each node is labeled with a symbolic response-time expression. Green and yellow edges correspond to the matching colored edges in Fig. 4, aiding visual alignment between the traversal tree and the trace graph. Blue nodes indicate ancestors of n^i_{source} where traversal terminates. Red nodes represent previously visited states and reuse cached values.

Algorithm 1 Depth-First Search with Memoization

```
1:  procedure TG-DFS(n, Γ_D, M)                          ▷ n: node
2:                                                       ▷ Γ_D: trace graph
3:                                                       ▷ M: memoization cache
4:      if n.tgid ∈ M then   ▷ Use cached value, if available
5:          n.value ← M[n.tgid]
6:          return
7:      end if
8:      C ← children[n.tgid, Γ_D]                        ▷ Instantiate children
9:      for all c ∈ C do
10:         if c = n^i_source then                       ▷ Terminate
11:             c.value ← 0
12:         else if c ∈ ancestors(n^i_source) then       ▷ Terminate
13:             c.value ← path_specific_value(c, n^i_source)
14:         else
15:             TG-DFS(c, Γ_D, M)                        ▷ Recursive expansion
16:         end if
17:     end for
18:     for all c ∈ C do                                 ▷ Update own response time
19:         n.value ← n.value ∪ {c.value + (c → n).weight}
20:     end for
21:     M[n.tgid] ← n.value                              ▷ Cache result for reuse
22: end procedure
```

edge. If such a path exists, the function returns the negated execution time of the ancestor node, ensuring its contribution cancels out in the final response-time expression. If, however, the ancestor can only reach the source via paths starting with a downstream edge, the function returns zero—allowing the node to contribute positively to the response-time expression of n^i_{sink}. The proof of Lemma 3 (§V-C) formally demonstrates why this distinction is needed for exactness.

As the traversal completes, n^i_{sink} may accumulate multiple response-time expressions—one from each branch. The same applies to any intermediate node with multiple children. Rather than merging these expressions, the algorithm maintains them as distinct symbolic expressions (Line 19), as shown in Fig. 5.

To avoid redundant computation, the algorithm employs memoization using a cache M of values. Nodes in the trace

graph may appear on multiple DFS paths. For instance, nodes 3 and 6 in Fig. 5 both correspond to n^{i-1}_{sink}, and nodes 2 and 5 both correspond to n^{i-1}_{filter}. The algorithm caches each node's *value* (Line 21), indexed by operator name and iteration (denoted *tgid* in Algorithm 1). This allows reuse of previously computed results and enables pruning (Line 5).

Once TG-DFS completes, the root node n^i_{sink} holds a set of expressions representing $R^i - \epsilon(n^i_{sink})$, each derived from a different branch of the execution tree. To compute the worst-case response time, we add the execution time of the sink node back to each expression and take the maximum. We prove in that this maximum results in a safe (Lemma 2 in §V-B) and tight (Lemma 3 in §V-C) upper bound on the WCRT.

As an example, in Fig. 5, let v_i denote the value of node i. We set v_1, v_2, v_4, and v_5 to zero, since these nodes belong to the ancestor set and can reach the source node n^i_{source} only via paths that begin with a downstream-blocking edge. v_6 and v_3 refer to the same node. Then, by resolving for v_8 (the value at the root), we obtain the following expression for R^i:

$$R^i = \max_i \begin{pmatrix} \epsilon(n^i_{sink}) + \epsilon(n^i_{filter}) + \epsilon(n^i_{source}), \\ \epsilon(n^i_{sink}) + \epsilon(n^i_{filter}) + \epsilon(n^{i-2}_{sink}), \\ \epsilon(n^i_{sink}) + \epsilon(n^i_{filter}) + \epsilon(n^{i-1}_{filter}), \\ \epsilon(n^i_{sink}) + \epsilon(n^{i-1}_{sink}) + \epsilon(n^{i-1}_{filter}), \\ \epsilon(n^i_{sink}) + \epsilon(n^{i-1}_{sink}) + \epsilon(n^{i-2}_{sink}), \end{pmatrix}. \quad (8)$$

The first term in Eq. (8), for instance, corresponds to the branch from node 4 to node 7 to node 8 in Fig. 5. This general solution can be instantiated for a specific problem by substituting known worst-case operator execution times.

We implement *TG-DFS* in Python as a recursive function that updates a shared tree data structure to represent the ongoing traversal, along with a dictionary used as the memoization cache. Rather than beginning from an arbitrary iteration i and traversing backwards, we label the root node of the tree as the 0^{th} node and increment the iteration index as we move deeper into the tree. This is simply a convenient implementation detail. The remainder of the tree is recursively built by expanding the children of each node and computing expressions for their response times. To efficiently check membership in the ancestor set, we leverage the repetitive structure of the trace graph to perform this check in linear time. For instance, if a node has a path to the source, its previous versions must also have a path to the source via sequential-execution. Once *TG-DFS* completes, we solve the resulting maximization problem straightforwardly, by iterating through each expression and evaluating it while keeping track of the maximum value encountered so far.

V. Proofs

We begin by defining expressions used in our lemmas and provide an overview of the following subsections.

Recall from the previous section that both sequential-execution edges (E1) and data-dependency edges (E2) in the trace graph are weighted—each assigned a weight equal to the execution time of its source node. In contrast, downstream-blocking edges (E3) have a weight of zero. Therefore, every path between any two nodes n^x_a and n^y_b in the trace graph has an associated cost, defined as the sum of the weights of the edges along that path. We define $\text{LP}(n^x_a, n^y_b)$ as the longest such path (i.e., a path with the highest total cost) from node n^x_a to node n^y_b, and let $\text{cLP}(n^x_a, n^y_b)$ denote the cost of this longest path. Additionally, we use $\text{c}(P)$ to denote the cost of any arbitrary path P.

Also recall that each node $n^i_k \in \Gamma_D$ is associated with a start time $s(n^i_k)$, defined as the earliest time the node can begin execution. The structure of the trace graph ensures that this start time corresponds to the cost of the longest path from the initial source node n^0_{source} to n^i_k, i.e.,

$$s(n^i_k) = \text{cLP}(n^0_{source}, n^i_k). \quad (9)$$

Using this fact and referring to Eq. (6) from §IV, the objective of TG-DFS is equivalent to computing:

$$R^i = \text{cLP}(n^0_{source}, n^i_{sink}) + \epsilon(n^i_{sink}) - \text{cLP}(n^0_{source}, n^i_{source}). \quad (10)$$

Since $\epsilon(n^i_{sink})$ can be trivially maximized, TG-DFS focuses on computing a simplified expression, which, from Eqs. (7) and (9), is equivalent to computing:

$$\bar{R}^i = R^i - \epsilon(n^i_{sink}) = \text{cLP}(n^0_{source}, n^i_{sink}) - \text{cLP}(n^0_{source}, n^i_{source}). \quad (11)$$

In theory, TG-DFS could *directly* compute $\text{cLP}(n^0_{source}, n^i_{sink})$ and $\text{cLP}(n^0_{source}, n^i_{source})$ in Eq. (11), but doing so becomes prohibitively expensive for large values of i. To improve efficiency, TG-DFS employs an optimization that terminates the traversal along any branch as soon as it encounters an ancestor of n^i_{source}. We explain using Lemma 1 (§V-A) why this strategy guarantees fast termination and ensures that the algorithm's time complexity depends not on i, but only on the structure of the original Holoscan DAG D. We then show using Lemma 2 (§V-B) and Lemma 3 (§V-C) that despite this optimization the response time computed using TG-DFS's output remains both safe and tight (respectively).

A. Termination

The TG-DFS algorithm terminates if each DFS branch in Algorithm 1 eventually reaches either the source node n^i_{source} (Line 10) or a node in its ancestor set (Line 12) within a finite number of steps. While the algorithm distinguishes between these two cases for clarity, the second condition alone is sufficient, since n^i_{source} is itself an element of $\textit{ancestors}(n^i_{source})$.

Let $N(o_{source}, o_{sink})$ denote the length of the longest path in D, measured in number of edges from the source to the sink operator, and define $z = N(o_{source}, o_{sink})$. In the example from Fig. 4 for D_{radar}, we have $z = 2$. In Lemma 1, we prove that if we go back $z \cdot \delta$ iterations from iteration i in the trace graph Γ_D, then every node $n^{i-z\delta}_k$ in that layer belongs to $\textit{ancestors}(n^i_{source})$. This guarantees that all DFS branches terminate. Fig. 6 illustrates the core steps of the proof.

Lemma 1. $\forall n^{i-z\delta}_k \in \Gamma_D,\ n^{i-z\delta}_k \in \textit{ancestors}\left(n^i_{source}\right)$.

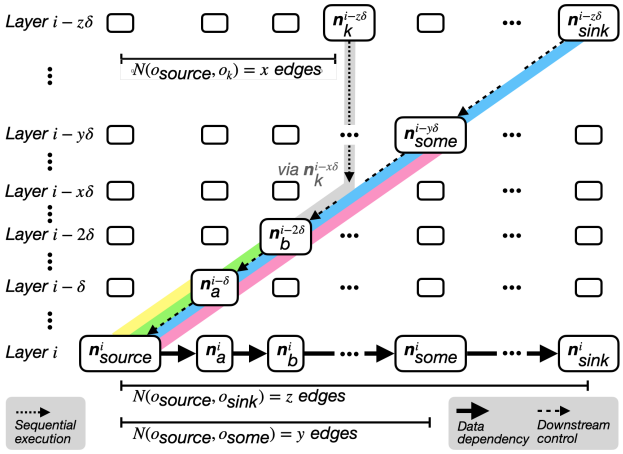

Fig. 6. A segment of the trace graph with $z\delta + 1$ layers, where z is the number of edges on the path from the source to the sink operator, and δ is the queue size. Each colored line represents a distinct ancestor's path to n^i_{source}. For instance, $n^{i-z\delta}_{sink}$ reaches n^i_{source} by traversing z downstream edges.

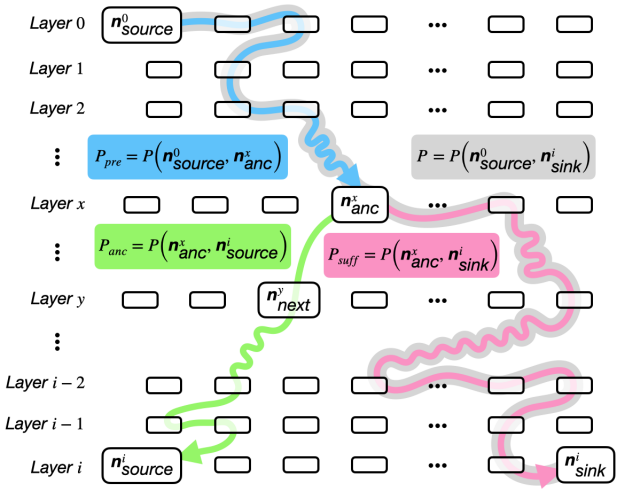

Fig. 7. The paths used in the proofs of Lemmas 2 and 3. The green and pink paths, P_{anc} and P_{suff}, begin from a shared ancestor, n^x_{anc}, and continue to n^i_{source} and n^i_{sink} respectively. The blue path, P_{pre}, connects the first node in the execution history, n^0_{source}, to n^x_{anc}. Finally, the gray path P is the combination of P_{pre} and P_{suff}.

Proof. Recall from §IV that the downstream-blocking edges (E3) in Γ_D guarantee that if an edge (o_a, o_b) exists in D, then for all i, an edge $(n_b^{i-\delta}, n_a^i)$ exists in Γ_D.

We begin by considering a direct successor of o_{source} in D, say o_a, where $N(o_{source}, o_a) = 1$. To reach n^i_{source} via one of o_a's nodes, we must go back δ layers to $n_a^{i-\delta}$, which connects to n^i_{source} via a downstream edge (see the yellow path in Fig. 6). Now consider o_a's direct successor o_b, where $N(o_{source}, o_b) = 2$. To reach n^i_{source} from one of o_b's nodes, we go back 2δ layers to $n_b^{i-2\delta}$, which first connects to $n_a^{i-\delta}$ and then to n^i_{source}, each over δ layers (see the green path).

In general, for any operator o_{some} with $N(o_{source}, o_{some}) = y \leq z$, a path to n^i_{source} exists from $n^{i-y\delta}_{some}$, via successive downstream-blocking edges (pink path). Since z is the length of the longest path, this argument applies to all operators in D, including the sink operator o_{sink} (blue path). Formally,

$$\forall o_{some} \in D, \exists y \in [0, z] : n^{i-y\delta}_{some} \in ancestors(n^i_{source}). \quad (12)$$

We now leverage the sequential-execution edges (E1), which connect n_k^{i-1} to n_k^i for each operator o_k in D, across all layers in Γ_D. Take any node $n_k^{i-z\delta} \in \Gamma_D^{i-z\delta}$. If $k = sink$, we already showed that $n^{i-z\delta}_{sink} \in ancestors(n^i_{source})$. For other nodes, follow the sequence of edges $(n_k^{i-z\delta}, n_k^{i-z\delta+1}, \ldots, n_k^{i-x\delta})$ until reaching some $n_k^{i-x\delta} \in ancestors(n^i_{source})$ (as guaranteed to exist by Eq. (12)). From there, downstream-blocking edges lead to n^i_{source} (see the grey path in Fig. 6, which merges into the green path). Thus, every $n_k^{i-z\delta}$ has a path to an ancestor of n^i_{source}, and is therefore itself an ancestor of n^i_{source}. □

Note that we keep δ, the queue size, constant throughout Lemma 1 for ease of understanding. Allowing δ to vary across operators only changes the structure of the trace graph (without violating key assumptions such as acyclicity), so Algorithm 1, which bases its decisions on edges between nodes in the trace graph, still applies unchanged. The core of the proof is that there will always exist a path from some sufficiently distant sink node (e.g., $n^{i-z\delta}_{sink}$ when δ is constant) to n^i_{source}. Consulting Fig. 6, it should be apparent that even with variable queue sizes the paths we highlight will still exist in some form. In other words, the proof depends on a property of the downstream condition, not on the specific choice of δ.

B. Safety

TG-DFS ultimately returns a set of expressions; however, we focus for now on a single one. Each such expression corresponds to the cost of a path $P_{suff} = P(n^x_{anc}, n^k_{sink})$, which begins at some node $n^x_{anc} \in ancestors(n^i_{source})$. Fig. 7 illustrates P_{suff} and other paths that we use in this section. The existence of such an ancestor is guaranteed by Lemma 1, provided i is sufficiently large. Moreover, n^x_{anc} is the only ancestor of n^i_{source} on this path; otherwise, the algorithm would have terminated earlier along the branch and not reached n^x_{anc}.

In addition to the cost of the path P_{suff}, each expression includes a path-specific adjustment, given by $path\_specific\_value(n^x_{anc}, n^i_{source})$ (see Lines 13 and 19 in Algorithm 1). As explained earlier, this function checks whether there exists a valid path from n^x_{anc} to n^i_{source} that does not leave n^x_{anc} via a downstream-blocking edge. If such a path exists, the function returns the negated execution time of n^x_{anc}, effectively subtracting the execution cost of this node from the total cost of P_{suff}. As a result, each expression returned by the algorithm takes the form $c(P_{suff}) - \rho$, where ρ is either 0 or $\epsilon(n^x_{anc})$.

Now, let P denote the longest path from n^0_{source} to n^i_{sink}. Clearly, TG-DFS would have traversed a suffix of this path during its depth-first search. Therefore, among all expressions collected by TG-DFS, there must exist one corresponding to this suffix. Without loss of generality, let P_{suff} denote this suffix, let n^x_{anc} be the most recent ancestor on this path closest

to n^i_{sink}—that is, the node at which the algorithm terminates the branch traversal—and let $\mathrm{c}(P_{suff}) - \rho$ be the expression resulting from this traversal. We now show that this expression, $\mathrm{c}(P_{suff}) - \rho$, upper-bounds $\bar{R}^i$ as defined in Eq. (11).

The following lemma also refers to the paths P_{pre} and P_{anc}. Specifically, $P_{pre} = P(n^0_{source}, n^x_{anc})$ denotes the prefix of P from the initial source node to n^x_{anc}, and $P_{anc} = P(n^x_{anc}, n^i_{source})$ represents the path from the ancestor node to n^i_{source}. Fig. 7 also illustrates P, P_{pre}, and P_{anc} in addition to P_{suff}.

Lemma 2. *If $P = LP(n^0_{source}, n^i_{sink})$, $\bar{R}^i \leq \mathrm{c}(P_{suff}) - \rho$.*

Proof. From Eq. (11),

$$\bar{R}^i = \mathrm{cLP}(n^0_{source}, n^i_{sink}) - \mathrm{cLP}(n^0_{source}, n^i_{source}). \tag{13}$$

Since $P = \mathrm{LP}(n^0_{source}, n^i_{sink})$,

$$\bar{R}^i = c(P) - \mathrm{cLP}(n^0_{source}, n^i_{source}). \tag{14}$$

By definition of $\mathrm{cLP}(*)$,

$$\mathrm{cLP}(n^0_{source}, n^i_{source}) \geq \mathrm{c}(P_{pre}) + \mathrm{c}(P_{anc}). \tag{15}$$

Multiplying by -1 on both sides.

$$-\mathrm{cLP}(n^0_{source}, n^i_{source}) \leq -\mathrm{c}(P_{pre}) - \mathrm{c}(P_{anc}). \tag{16}$$

From Eqs. (14) and (16).

$$\bar{R}^i \leq c(P) - \mathrm{c}(P_{pre}) - \mathrm{c}(P_{anc}) \tag{17}$$

$$= \mathrm{c}(P_{suff}) - \mathrm{c}(P_{anc}). \tag{18}$$

Since n^x_{anc} is on P_{anc}, whether ρ is 0 or $\epsilon(n^x_{anc})$,

$$c(P_{anc}) \geq \rho \implies -\mathrm{c}(P_{anc}) \leq -\rho \tag{19}$$

Eqs. (18) and (19) together prove the lemma. □

Lemma 2 shows that $\bar{R}^i$ is upper-bounded by the TG-DFS expression corresponding to the longest path P. Since the algorithm returns the maximum among all expressions gathered during its depth-first traversal, even if a different expression is ultimately selected as the maximum, it will be greater than or equal to $\mathrm{c}(P_{suff}) - \rho$, and thus also upper-bound $\bar{R}^i$.

C. Exactness of Bounds

Here, we show that regardless of which expression the algorithm selects, there always exists a configuration that yields a response time exactly equal to that expression. In other words, the algorithm always computes a tight bound.

We reuse the paths from the previous lemma but no longer assume upfront that P is the longest path. Instead, recognizing that node execution times may realistically vary between 0 and their respective WCETs, we construct a valid configuration of actual execution times (denoted by $\epsilon(*)$) that causes P to become the longest path, and proceed from this configuration. As in the previous lemma, we consider a response-time bound of the form $\mathrm{c}(P_{suff}) - \rho$, resulting from a branch that terminates at the ancestor node n^x_{anc}, from which a path P_{anc} diverges toward n^i_{source}. We additionally define n^y_{next} as the node immediately following n^x_{anc} on P_{anc} (see Fig. 7).

Lemma 3. *There exists a configuration of node execution times such that $\bar{R}^i = \mathrm{c}(P_{suff}) - \rho$.*

Proof. We prove the lemma by constructing a valid configuration of execution times for which the response time bound is tight. Let all nodes not on path P_{suff} have zero execution time, i.e., for every node $n^j_k \notin P_{suff}$, set $\epsilon(n^j_k) = 0$. Additionally, let every node on P_{suff} whose execution time does not contribute to $\mathrm{c}(P_{suff})$ also have zero execution time.

Consider two cases for the edge (n^x_{anc}, n^y_{next}), where n^y_{next} is the immediate successor of n^x_{anc} on P_{anc}: **(i)** If this edge is downstream-blocking, then $\rho = 0$, and since all other nodes on P_{anc} have zero execution time, $c(P_{anc}) = 0 = \rho$. **(ii)** If the edge is not downstream-blocking (e.g., a data or sequential dependency), then $\rho = \epsilon(n^x_{anc})$, and since all other nodes on P_{anc} have zero cost, $c(P_{anc}) = \rho$. In both cases: $c(P_{anc}) = \rho$.

Now consider the start times of the two key nodes. Under this configuration, the start time of n^i_{source} is determined by the completion of the path $P_{pre} \cup P_{anc}$. Thus,

$$s(n^i_{source}) = \mathrm{c}(P_{pre}) + \mathrm{c}(P_{anc}) = \mathrm{c}(P_{pre}) + \rho. \tag{20}$$

The start time of n^i_{sink} is determined by the completion of path P, which is the concatenation of P_{pre} and P_{suff}:

$$s(n^i_{sink}) = \mathrm{c}(P) = \mathrm{c}(P_{pre}) + \mathrm{c}(P_{suff}). \tag{21}$$

By Eq. (9), we know that $s(n^i_{sink}) = \mathrm{cLP}(n^0_{source}, n^i_{sink})$ as well as $s(n^i_{source}) = \mathrm{cLP}(n^0_{source}, n^i_{source})$. Substituting these into the definition of response time from Eq. (11), we get:

$$\begin{aligned}\bar{R}^i &= s(n^i_{sink}) - s(n^i_{source}) \\ &= (\mathrm{c}(P_{pre}) + \mathrm{c}(P_{suff})) - (\mathrm{c}(P_{pre}) + \rho) \\ &= \mathrm{c}(P_{suff}) - \rho.\end{aligned} \tag{22}$$

Thus, the bound computed by the algorithm is tight under this configuration. This completes the proof. □

Lemma 3 demonstrates why TG-DFS must include the $path\_specific\_value(c, n^i_{source})$ check. Without that calculation, the algorithm could only return $\mathrm{c}(P_{suff})$. Then we would be incorrectly considering $\mathrm{c}(P_{anc})$ as 0 instead of ρ, and the output of our algorithm would only upper bound the exact expression in Eq. (22). This would mean we could report a response time impossible to actually achieve. We already describe after Lemma 2 why it is always safe to subtract ρ from $\mathrm{c}(P_{suff})$.

Together, Lemmas 2 and 3 establish a reachable upper bound on $\bar{R}^i$, the response time in iteration i. The true worst-case response time of the system is the maximum value of $\bar{R}^i$ across all iterations. Notably, our analysis makes no assumptions specific to iteration i, so the bound computed by our algorithm applies to almost any iteration. The only exception concerns early iterations: for instance, the algorithm may explore paths involving three prior iterations, which is not feasible if $i = 1$. To precisely bound the response time of early iterations, one would need to terminate DFS branches that traverse too far into the past. Since early termination yields fewer and potentially smaller expressions, and the algorithm always returns the

TABLE II
AVERAGE PESSIMISM IN SCHOWITZ ET AL.'S [29] ANALYSIS

Graph	Pessimism	# Nodes	# Edges
A	1.13%	6	6
B	1.77%	4	3
C	0.69%	5	5
D	1.63%	5	5
E	12.46%	7	9
F	18.23%	7	7
G	11.84%	8	10
H	17.97%	9	11

maximum among them, the final result can never increase. Thus, the bound remains valid for all i.

VI. EVALUATION

We evaluate the scalability of our dynamic programming-based approach and compare it with prior methods to highlight its advantages. Our experiments are conducted on a server equipped with dual Intel Xeon Gold 6326 CPUs (32 cores total), 1 TB of RAM, and running Ubuntu 22.04.5 LTS. The CPUs have a maximum frequency of 3.5 GHz. Operator execution times are profiled on an NVIDIA IGX Orin Developer Kit, which features a 12-core Arm Cortex-A78AE CPU, 64 GB RAM, and an NVIDIA RTX 6000 Ada discrete GPU.

To evaluate performance across realistic and synthetic workloads, we begin with real-world graphs from NVIDIA's HoloHub repository [9], following the setup in Schowitz et al. [29]. The dataset includes eight representative applications, mostly in the medical domain, each with distinct graph structures. The Multi AI Ultrasound application shown in Fig. 1 is the most complex, with 9 nodes and 11 edges; the others range from 4–8 nodes and 3–10 edges.

For the scalability analysis, we generate synthetic DAGs as follows: we create a sequence of nodes, each assigned a worst-case execution time drawn uniformly at random from [100,1000]. Each node is then connected to a randomly selected successor to ensure at least one source-to-sink path. To avoid isolated nodes, any node (except the source) without an incoming edge receives one from a randomly chosen predecessor. Finally, we iterate through the nodes and add edges to successors with 20% probability, producing dense graphs with many more edges than nodes as graph size increases.

A. Comparative Experiments

Schowitz et al. [29] present safe upper bounds on Holoscan DAG response times. However, the analyses from which they derive these bounds are overly pessimistic for some graphs. Specifically, they assume the existence of edges in graphs that may not actually be present in order to facilitate their analyses.

To compare their work with our own, we evaluated their methodology using their open-source artifact [25]. Of the eight HoloHub graphs in their evaluation, Table II shows the percent pessimism for each (for full names of all the graphs, see Fig. 8).

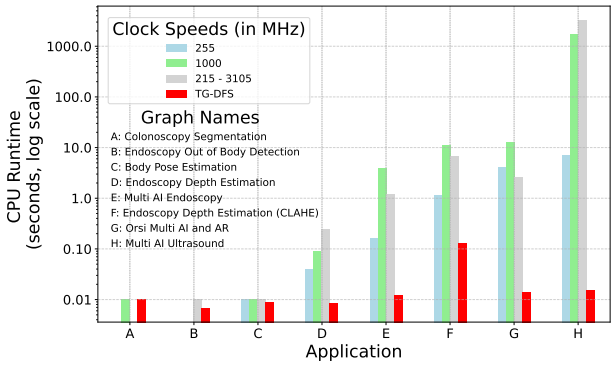

Fig. 8. CPU runtimes in UPPAAL for eight HoloHub applications. If no bar appears, UPPAAL reported the runtime as 0. We also show the runtime of the TG-DFS algorithm on the same applications. The algorithm is execution-time agnostic and hence is only plotted once for each application.

Their analysis demonstrates an average pessimism of up to 18% on the HoloHub applications, and it fails to provide any bound on the possible pessimism. This results in very loose bounds for certain graphs, as shown in their scalability analysis [29]. In contrast, our dynamic algorithm remains exact for all DAGs.

Next, we present a runtime scalability evaluation for bounding the latency of Holoscan DAGs, using the UPPAAL model checker to analyze the equivalent HSDFG. We employ the graph conversion methodology and implementation outlined in §III, using UPPAAL version 4.1.26-2. As with the previous experiment, our dataset consists of the eight HoloHub applications. Since UPPAAL reports CPU runtime rather than wall-clock time, we use CPU runtime in our results. UPPAAL uses clock variables to track state transitions (critical for modeling operator execution time), and we hypothesize that varying execution time configurations may affect the runtime of model checking. Rather than using randomly generated values, we profile realistic operator execution times on the NVIDIA IGX Orin under three configurations, each with a different clock speed. In the first configuration, we allow the GPU clock speed to vary between 215 MHz and 3105 MHz. In the other two configurations, we lock the clock speed at 255 MHz and 1000 MHz, respectively. We run the HoloHub applications in each configuration to profile the minimum and maximum operator execution times.[3] Fig. 8 shows the results.

For smaller graphs, like Body Pose Estimation (C), UPPAAL completes its computation in mere milliseconds. However, as the size of the graph increases, the model checker's state space expands rapidly. For instance, there is a two-order-of-magnitude increase in runtime between the Endoscopy Depth Estimation (D) and its CLAHE version (F), despite the difference between the graphs being only two nodes and two edges. Adding two more nodes, for a total of nine in the Multi AI Ultrasound (H) graph, results in another two-order-of-magnitude increase

[3]We were unable to profile operator execution times for the Orsi Multi AI and AR application (G), as it crashes on current versions of Holoscan. Instead, we use execution times from similar operators in other examples and apply them to the operators of this app.

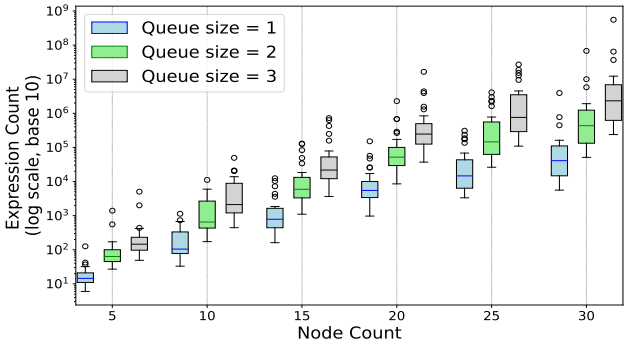

Fig. 9. Node count vs expression count returned by TG-DFS

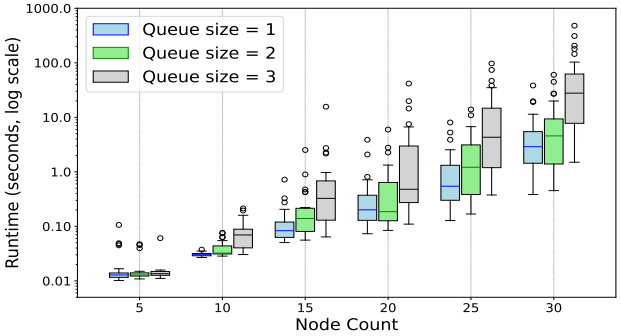

Fig. 10. Node count vs runtime for our full algorithm (TG-DFS and maximization step) on randomly generated DAGs.

in runtime, making the analysis of this graph take over an hour in the default configuration. Given that UPPAAL does not scale well with a queue size of 1, we did not conduct additional experiments with larger queue sizes. We also observe that specific execution time values can significantly impact UPPAAL's runtime. For instance, when limiting the GPU clock speed to 255 MHz, the longer execution times actually cause UPPAAL to finish faster than when execution times are shorter. This unpredictability is a notable downside, as even slight changes in the units used to record execution times could affect the duration of UPPAAL's computation. In comparison, our algorithm produces the same result as UPPAAL for the Multi AI Ultrasound (H) application in 0.015 seconds. Furthermore, our algorithm is execution-time agnostic, as integer addition on fixed-size types operates in constant time.

B. Scalability Experiments

We begin the scalability study by examining a fundamental measure of complexity of our algorithm. As described in §IV, TG-DFS returns a set of expressions when invoked on a DAG, where each expression represents a candidate for the worst-case response time (WCRT). As the size and structural complexity of a graph increase, the number of these expressions also grows, leading to increased computational effort both in generating and evaluating them. Thus, the total number of expressions returned by TG-DFS serves as a natural proxy for the difficulty of analyzing a graph with our algorithm.

Fig. 9 illustrates the total number of expressions generated by TG-DFS for randomly constructed graphs, with the node count shown on the x-axis. Each box plot is based on 100 independently generated graphs. We observe an exponential increase in the number of expressions as the node count grows. Furthermore, increasing the queue size significantly amplifies the number of possible expressions, by more than an order of magnitude for some graphs.

There is substantial variation in expression count even among graphs with the same number of nodes. This is expected, as the total number of paths through a graph depends not just on node and edge count, but also on their specific arrangement. Consequently, graphs with seemingly similar parameters can differ dramatically in expression count. Having established how expression count scales with graph and queue size, we now turn to the study of execution times.

Fig. 10 presents the execution time of our algorithm under the same experimental setup as Fig. 9. For graphs with only five nodes, runtime differences across queue sizes are minimal. However, as graphs grow larger, the effect of queue size on execution time becomes increasingly significant. Overall, the trends in execution time align closely with the observed growth in expression count.

Under the same experimental conditions, Fig. 11 breaks down execution time into two components: the time spent generating expressions with TG-DFS, and the time spent evaluating them to determine the WCRT. When the queue size is set to 1, nearly all computation time is devoted to TG-DFS. But as the queue size increases, especially in combination with larger node counts, the maximization step consumes a greater portion of the total runtime. This is consistent with our theoretical understanding of the algorithm: as queue size increases, response time expressions become longer. Specifically, Lemma 1 bounds the depth of the required trace graph traversal to $z \cdot \delta$, where δ is the queue size. Each increment in δ can thus require the algorithm to explore z additional layers, producing longer expressions and increasing the cost of computing each candidate response time.

While the previous experiments focused on increasing node count under relatively small queue sizes, we now explore the opposite scenario: analyzing small graphs while scaling queue sizes significantly. Fig. 12 shows execution time as a function of queue size for four representative HoloHub graphs. We begin at a queue size of 10, as smaller sizes produce runtimes too brief to be reliably measured due to jitter. Among these, Multi AI Ultrasound (H), having more nodes and edges, unsurprisingly exhibits the steepest growth. However, Endoscopy Depth Estimation (CLAHE) (F) scales worse than either Orsi Multi AI and AR (G) or Multi AI Endoscopy (E), despite being structurally less complex. This reinforces our earlier point: the difficulty of analyzing a graph is not determined solely by node and edge counts. The results also support our previous claim that for small graphs, increasing the

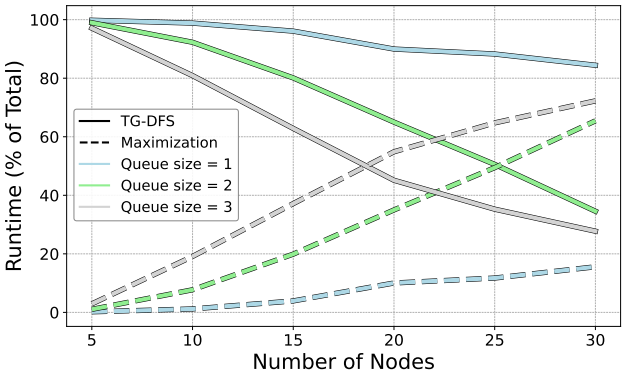

Fig. 11. Proportion of runtime spent in TG-DFS vs maximization

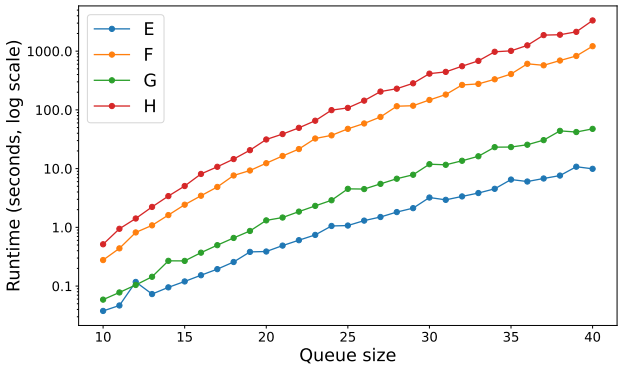

Fig. 12. Queue size vs runtime for four HoloHub graphs

queue size by one does not necessarily lead to an exponential increase in runtime, as can happen in larger graphs (cf. Fig. 10). This analysis enables, for the first time, Holoscan developers to investigate the impact of varying buffer sizes without the need for time-consuming empirical testing.

VII. Related Work and Conclusion

There is a substantial body of work on exact response-time analysis. While we have already discussed methods relevant to Holoscan and synchronous dataflow graphs (SDFGs) in §I–III, schedule abstraction graph (SAG) techniques [24] warrant further mention. SAGs represent all possible execution scenarios under timing uncertainties by merging semantically equivalent states, thereby avoiding state-space explosion. However, they target periodic workloads with known release patterns, unlike our dynamic, input-driven setting. Nonetheless, we plan to adapt their approach in future work, leveraging parallelism [1] and libraries such as Threading Building Blocks [18] to improve scalability.

Framing timing analysis as a graph problem also connects naturally to graph-theoretic techniques. We explored modeling response-time computation as a bilevel optimization problem, inspired by Stackelberg shortest-path formulations [21], where execution times are chosen to maximize response time under feasibility constraints. Despite its elegance, this approach proved computationally impractical for realistic workloads.

Finally, an important assumption we make is that WCETs are known, but these are by no means trivial to determine. The WCET analysis of a real-time task must consider many complex factors such as memory and other hardware microarchitectural interferences. There is an extensive body of literature on this subject, along with tools and methods used to determine upper bounds on execution times in practice [32]. However, we consider incorporating these methods as beyond the scope of our analysis, which is focused on higher-level interactions between tasks in an application graph.

In conclusion, the complex interactions between Holoscan operators motivated our development of a custom, scalable algorithm tailored to Holoscan's execution model, a domain that continues to grow in adoption and complexity. By producing tight, iteration-aware upper bounds on the worst-case response time for any given input, our analysis enables estimation of the maximum sustainable throughput, defined as the inverse of this bound. This provides application developers with actionable guidance on the highest input rate their pipeline can support while still meeting timing guarantees.

Although our focus is on Holoscan, the algorithm is broadly applicable to other domains that employ SDFGs, such as digital signal processing or workloads on multiprocessor system-on-chip (MPSoC) platforms [16], provided the input graph remains acyclic. Looking ahead, we aim to relax remaining assumptions and extend our analysis to support a wider range of scheduling policies and hardware constraints. This would further enhance the practical applicability of our approach to increasingly diverse Holoscan deployments. Our implementation of the algorithm and all evaluation scripts are available online.[4]

VIII. Acknowledgements

We would like to thank our anonymous reviewers for their comments, and our summer intern Shuyi Li for reviewing the proofs in this paper. This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia (UBC) as well as the Natural Sciences and Engineering Research Council of Canada (NSERC). OpenAI ChatGPT [3] was used to assist with grammar and language editing across all sections, based on a complete and polished draft written by the authors.

References

[1] schedule_abstraction-main: An implementation of schedulability tests for non-preemptive job sets. https://github.com/SAG-org/schedule_abstraction-main.
[2] UPPAAL model checker. https://uppaal.org/, 2024.
[3] https://chat.openai.com, 2024.
[4] ROS. https://www.ros.org/, 2025.
[5] Data Flow Tracking. https://docs.nvidia.com/holoscan/sdk-user-guide/flow_tracking.html, 2025.
[6] Event-Based Scheduler C++ Class Definition. https://docs.nvidia.com/holoscan/sdk-user-guide/api/cpp/

[4] https://github.com/ubc-systopia/rt-holoscan-artifacts

classholoscan_1_1eventbasedscheduler.html#exhale-class-classholoscan-1-1eventbasedscheduler, 2025.
[7] Greedy Scheduler C++ Class Definition. https://docs.nvidia.com/holoscan/sdk-user-guide/api/cpp/classholoscan_1_1greedyscheduler.html#exhale-class-classholoscan-1-1greedyscheduler, 2025.
[8] GXF Core Concepts . https://docs.nvidia.com/holoscan/sdk-user-guide/gxf/gxf_core_concepts.html, 2025.
[9] HoloHub - GitHub. https://github.com/nvidia-holoscan/holohub, 2025.
[10] NVIDIA IGX Orin. https://www.nvidia.com/en-us/edge-computing/products/igx/, 2025.
[11] Multi-Thread Scheduler C++ Class Definition. https://docs.nvidia.com/holoscan/sdk-user-guide/api/cpp/classholoscan_1_1multithreadscheduler.html#exhale-class-classholoscan-1-1multithreadscheduler, 2025.
[12] Designing Real-Time AI Streaming Pipelines for Healthcare and Robotics. https://www.nvidia.com/en-us/on-demand/session/gtc25-dlit71305/, 2025. GTC 2025 Session DLIT71305.
[13] Holoscan Documentation. https://docs.nvidia.com/holoscan/index.html, 2025.
[14] H. I. Ali, B. Akesson, and L. M. Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2015)*.
[15] R. Alur. Timed automata. In *Computer Aided Verification 1999*.
[16] A. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. Theelen. Latency minimization for synchronous data flow graphs. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*.
[17] Google. Google OR-Tools: Constraint Programming Solver. https://developers.google.com/optimization/cp/cp_solver, 2025.
[18] Intel Corporation. onetbb: Threading building blocks. https://github.com/uxlfoundation/oneTBB, 2021.
[19] G. Kuiper. *Accurate analysis of real-time stream processing applications: using dataflow models and timed automata*. PhD Thesis, University of Twente, Netherlands, 2019.
[20] G. Kuiper and M. J. Bekooij. Latency analysis of homogeneous synchronous dataflow graphs using timed automata. In *Design, Automation & Test in Europe Conference & Exhibition (DATE 2017)*.
[21] M. Labbe, P. Marcotte, and G. Savard. A bilevel model of taxation and its application to optimal highway pricing. *Management Science*, 44(12):1608–1622, 1998.
[22] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, 1987.
[23] O. M. Moreira and M. J. G. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.
[24] M. Nasri and B. B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.
[25] P. Schowitz, S. Sinha, and A. Gujarati. Holoscan SDK Response-Time Analysis. https://github.com/nvidia-holoscan/holohub/tree/main/tutorials/holoscan_response_time_analysis/artifact, 2025.
[26] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. AI and ml accelerator survey and trends. In *High Performance Extreme Computing (HPEC)*, 2022.
[27] R. A. Roberts and C. T. Mullis. *Digital signal processing*. Addison-Wesley Longman Publishing Co., Inc., 1987.
[28] G. Roumage, S. Azaiez, and S. Louise. A survey of main dataflow moccs for cps design and verification. In *15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2022)*.
[29] P. Schowitz, S. Sinha, and A. Gujarati. Response-time analysis of a soft real-time NVIDIA Holoscan application. In *45th IEEE Real-Time Systems Symposium (RTSS 2024)*.
[30] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. *Neural Information Processing Systems (NeurIPS 2015)*.
[31] S. Sinha, S. Dwivedi, and M. Azizian. Towards deterministic end-to-end latency for medical AI systems in NVIDIA Holoscan. In *15th International Conference on Cyber-Physical Systems (ICCPS 2024)*.
[32] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.