Response-Time Analysis of a Soft Real-time NVIDIA Holoscan Application

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

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

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

***Abstract*—NVIDIA Holoscan SDK is a novel edge and embedded software development framework designed for NVIDIA System-on-Chips (SoCs), primarily targeting medical device applications. This SDK facilitates complex data processing workflows using Directed Acyclic Graphs (DAGs) composed of functional units termed operators. These operators, running in separate threads, are usually interconnected with intricate execution dependencies influenced by both upstream and downstream conditions on communication data buffers. Current methods to measure the response time of a complex Holoscan application rely on empirical benchmarking, which can be costly, time-consuming, and unreliable – limitations that are particularly critical in sectors where safety and certification concerns are paramount.**

This paper introduces a novel static analysis methodology to determine worst-case end-to-end response times in NVIDIA Holoscan applications. Our approach overcomes the drawbacks of existing empirical tools by providing a response-time analysis capable of handling complex operator interactions and communication buffering mechanisms inherent in Holoscan's architecture. Through rigorous theoretical analysis and empirical validation, our method not only ensures predictability in system behavior but also aids developers in identifying performance bottlenecks and optimizing system design. Evaluation using real-world NVIDIA HoloHub applications demonstrates the efficiency and accuracy of our analysis, achieving theoretical response times as close as 0.3% of empirically measured numbers on NVIDIA hardware using less than 1 ms computation time.

I. Introduction

The medical devices industry is increasingly adopting powerful edge and embedded computing platforms to meet the growing demand for computation power and real-time processing of streaming sensor data. Modern devices support complex workflows and are designed to be deployed on the premises of hospitals and medical care facilities, such that they can last for over a decade. NVIDIA Holoscan [1] is one such platform that is gaining traction across the industry [2, 3].

The Holoscan SDK [1] is a software development framework for ARM-based SoCs manufactured by NVIDIA, such as AGX and IGX Orin [4]. It presents a unique programming model that facilitates a highly modular application development to help medical device application developers program complex interconnected workflows. The model composes individual function blocks (as vertices) into a Directed Acyclic Graph (DAG). The vertices (known as *operators* in Holoscan) run as threads and the edges govern data flow and dependencies between the vertices. The operators can be reused across applications.

Holoscan's programming model enables rapid prototyping and development of applications for an eventual production-grade deployment. However, the intertwined architecture of queues, queue conditions, and operators in different threads complicates the analysis of a Holoscan application's behavior. For instance, Holoscan supports data transfers between operators through double-buffered queues and imposes restrictions on the execution of operators based on the status of these queues. Currently, Holoscan relies on empirical tools, such as the Data Flow Tracking feature [5], to measure end-to-end latency of a Holoscan application. As shown in Section III, such empirical tools are dependent on and limited by the testing setup and duration. In the regulated medical devices sector, deriving a static response-time analysis for Holoscan applications is highly desirable but missing.

Holoscan's application framework does not conform to well-known system models, such as existing real-time scheduling models for DAG tasks. The closest model is synchronous dataflow (SDF), but existing response-time analyses for SDF [6–9] cannot account for the use of queues and restrictions on operator execution based on communication buffer states found in Holoscan. Our paper is the first attempt to bound the worst-case end-to-end response time for any item (e.g., a video frame) within Holoscan's streaming workflow model, considering blocking due to communication buffer constraints.

We evaluate our analysis using example application structures from NVIDIA HoloHub [10], a community-populated repository for Holoscan applications. For comparison, we use empirical results from the Holoscan Data Flow Tracking feature on NVIDIA hardware and simulated results on a simplified Holoscan application runtime model. Evaluation results demonstrate that our analysis is efficient, taking less than 1 ms to process the small DAGs used in Holoscan, and closely bounds approximate empirical results for most variations of the eight tested DAG application structures in HoloHub.

II. Holoscan Internals

The Holoscan SDK's execution backend, called the Graph Execution Framework (GXF) [11] (also developed by NVIDIA) handles scheduling, data transfers, and synchronization of Holoscan operators. This paper analyzes the Holoscan SDK source code and the GXF C++ header files (both of which are open-source) in order to model Holoscan's internal design and

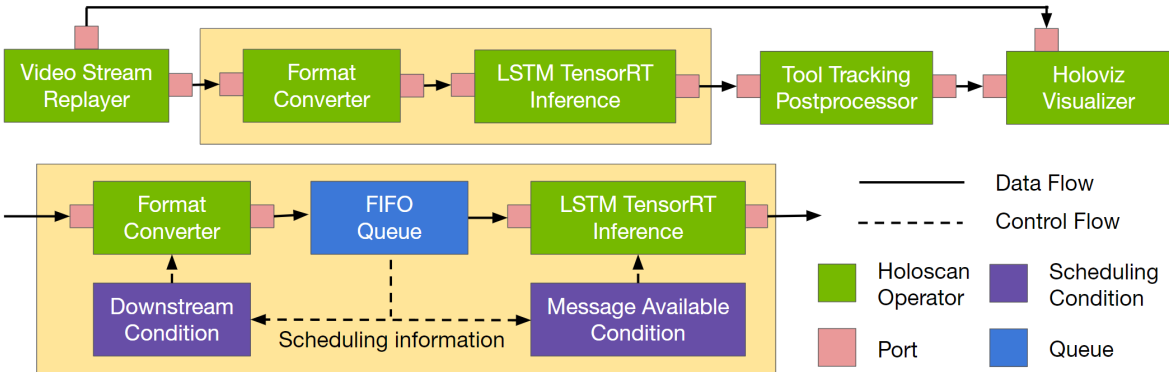

Fig. 1. Illustration of Holoscan internals using an endoscopy tool tracking application. The application is composed of five functional blocks, which are combined in the form of a DAG, as shown in the top part of the figure. The interaction between the Format Converter and LSTM TensorRT Inference operators is highlighted to illustrate the key Holoscan components underlying the DAG. The two operators share a FIFO queue between them. Format Converter is subject to a downstream condition, which means that it cannot start processing a new input if there is no space in the (output) FIFO queue. LSTM TensorRT Inference is subject to an upstream condition, which simply means that it cannot start processing a new input if no input is available in the (input) FIFO queue. Operator execution, thread allocation, and scheduling is managed by the GXF backend, which is not shown in the figure.

conduct a response-time analysis of a Holoscan application. Fig. 1 illustrates the high-level interactions between two communicating Holoscan operators using a Holoscan endoscopy tool tracking application [12] as an example.

Operators, the building blocks of a Holoscan application, perform recurring tasks such as processing sensor inputs and displaying a stream of images. These operators are interconnected in a DAG to facilitate complex interactions and dependencies in an application. Each operator features input and output ports, named *Receivers* and *Transmitters*, which handle data reception and transmission to other operators. The communication between each pair of output (transmitter) and input (receiver) ports (i.e., the data flow along an edge between two adjacent operators) is orchestrated by the backend using a First-In-First-Out (FIFO) *double-buffered queue* [13], which also serves as a data buffer for the ports.[1] If full, the queue can be configured to drop either old or new elements depending on the *overflow discipline* configured.[2] The queue implementation and the overflow discipline do not affect the FIFO property when considering items that are not dropped. Hence, we ignore these details in our worst-case response-time analysis.

Each operator is subjected to a number of conditions that can affect their execution. A condition could relate to an input port, determining if the connected input buffer has adequate space to accommodate a new message. Alternatively, it might be a periodic condition that verifies if a specified duration has elapsed since the last invocation of the operator. The two most prevalent conditions are *MessageAvailableCondition* and *DownstreamMessageAffordableCondition*, which we refer to as *upstream* and *downstream* conditions, respectively. The upstream condition ensures that an operator activates only when each of its input queue buffers is not empty. The downstream condition allows operator execution only when the output port buffers have space to accommodate new messages.

The Holoscan SDK offers two user-level schedulers for application operators: Greedy Scheduler [14] and Multi-threaded Scheduler [15]. The Greedy Scheduler processes operators in an application DAG in topological order using a single thread, preventing parallel operator execution. Response-time analysis is trivial as response times are simply the sum of individual operator execution times under default conditions. We focus on the Multi-threaded Scheduler. This scheduler also arranges operators in a DAG in topological order but executes them concurrently across multiple threads. For our response-time analysis, we assume that the number of threads matches the total number of operators, and each thread is pinned to a single exclusive CPU core. This assumption holds as our dataset, sourced from NVIDIA HoloHub, consists of applications where the operator and thread count do not surpass the core availability in NVIDIA AGX and IGX Orin platforms.

III. Motivating Examples

We first demonstrate how the downstream conditions in the Holoscan SDK add additional delays, which can have a substantial impact on worst-case response times.

Example 1. Consider a DAG with just two connected operators O_1 and O_2 requiring execution times of $e_1 = 100\,ms$ and $e_2 = 1000\,ms$, respectively. Queue Q_1 buffers inputs before O_1 and queue $Q_{1,2}$ buffers intermediate data between O_1 and O_2. Suppose both queues have a capacity of $C = 1$. A new input arrives every $100\,ms$. The overflow discipline ensures that if Q_1 is full, the input is dropped, else, the input is buffered in Q_1. Fig. 2 illustrates the DAG structure and the state of its queues and operators at different times, as inputs I_1, I_2, …, I_{32} arrive periodically. We assume multiple events

[1]Each double-buffered queue configured to hold C items is actually divided into two stages, the main stage and the backstage, each of which has capacity C and which operate together as a FIFO ring buffer. When a new item is pushed, it is placed into the backstage. When an item is popped, the oldest item in the main stage is removed and returned. A `sync` function moves all the items in the backstage to the main stage whenever an operator reads from an input port or pushes to an output port. Despite the space for $2C$ items, the queue effectively operates as a single FIFO queue with capacity C.

[2]Three overflow disciplines are allowed: *kFault* throws an error in case of an overflow, *kReject* drops the new item without disturbing any old items, whereas *kPop* removes the oldest item to make space for the new item.

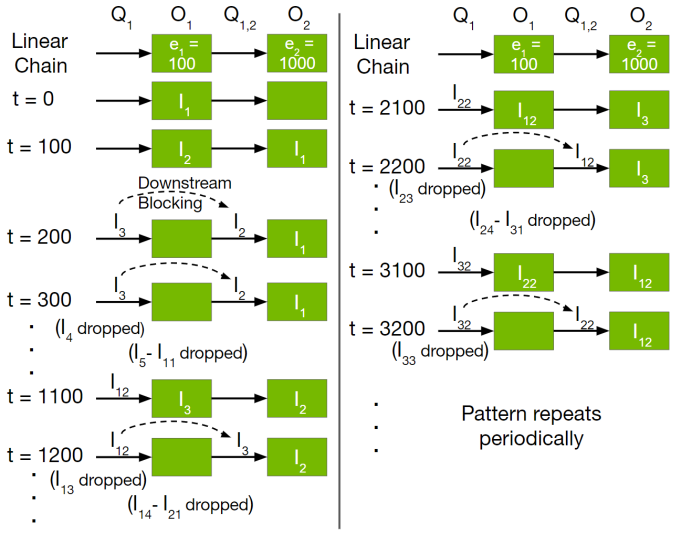

Fig. 2. An example showing how the downstream condition can affect operator execution. The figure shows the state of a simple DAG with two operators and with periodic inputs at different times, with the timestamps shown on the left. Operators O_1 and O_2 have execution times 100 ms and 1000 ms, respectively, and are denoted with green boxes. An idle operator is shown with an empty box, whereas an operator processing an input is labeled with that input inside the box. The arrows show the queues of O_1 and O_2, Q_1, and $Q_{1,2}$. A queue holding an input is labeled with that input on the top of its corresponding arrow, whereas an empty queue is not labeled at all.

can happen simultaneously and with no overhead. For example, input I_1 can leave operator O_1 and arrive at O_2 and another input I_2 can arrive at O_1, all at the same time instant.

Initially, at $t = 0\,ms$, input I_1 arrives; operator O_1 is idle at this time, so it immediately starts processing I_1. At $t = 100\,ms$, O_1 finishes processing I_1 and I_2 arrives; O_1 and O_2 hence immediately start processing I_2 and I_1, respectively. After another $100\,ms$, at $t = 200\,ms$, O_1 finishes processing I_2 and I_3 arrives; however, O_2 is still processing I_1; hence, I_2 is queued in $Q_{1,2}$, and I_3 is queued in Q_1. Note that I_3 is queued even though O_1 is idle because the downstream condition prevents O_1 from processing a new input until its downstream queue $Q_{1,2}$ has at least one empty space. All subsequent inputs I_4, I_5, …, I_{11} are dropped, as the state of the DAG does not change during this time, and Q_1 remains full. Finally, at $t = 1100\,ms$, when O_2 finishes processing I_1, it consumes I_2 from $Q_{1,2}$ and starts processing it; this empties $Q_{1,2}$, which satisfies O_1's downstream condition, which in turn allows O_1 to start processing I_3. The new input I_{12} can now be buffered and need not be dropped, as Q_1 is empty. This behavior repeats periodically, as shown in the figure.

Ignoring inputs that are dropped, the worst-case response time in the above example is $2900\,ms$, e.g., I_3 arrived at $t = 200\,ms$ and O_2 finished processing it at $t = 3100\,ms$. Developers may not be able to determine this worst-case response time easily. In fact, they may naively assume the worst-case response time to be $2100\,ms$: $1100\,ms$ being the total processing time and $1000\,ms$ being the queuing delay ahead of O_2. The downstream condition, however, induces additional blocking that adds an extra delay of $800\,ms$.

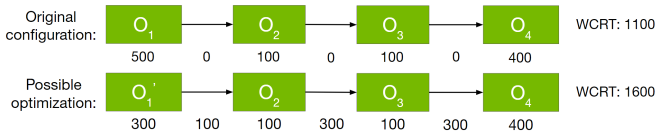

Fig. 3. An example showing a performance anomaly where a local optimization increases the overall response time. The top chain shows a linear chain with four operators; the execution times are shown below respective operators and the queuing delays are shown below respective edges. Ignoring delays prior to the first operator, the response time is simply the sum of all these delays, and indicated on the right of the chain. In the lower chain, the first operator has a lower execution time, but the worst-case response time is nonetheless higher due to queuing delays throughout the chain.

In this example, we used a simple linear chain with two operators. With more complex applications, reasoning about response times becomes even more difficult. In the following example, we show that the worst-case response time in a linear chain can often be sensitive to the ordering of execution times (specifically, it may depend on the operator in the chain with the greatest execution time), which can lead to performance anomalies that result in degraded response times.

Example 2. Consider four operators O_1, O_2, O_3, and O_4 connected in a linear chain, with execution times $e_1 = 500\,ms$, $e_2 = 100\,ms$, $e_3 = 100\,ms$, and $e_4 = 400\,ms$, respectively. Like in Example 1, suppose all queues have a capacity of $C = 1$ and that a new input arrives every $100\,ms$. Fig. 3 (top) illustrates this linear chain along with the queuing delays incurred between operators. Ignoring the queuing delay prior to the first operator for this example, all other queuing delays are always zero. However, this is just a side-effect of the specific execution times. Operator O_1 consumes a new input only every $500\,ms$, i.e., at $0\,ms, 500\,ms, 1000\,ms$ and so on. Consequently, O_2, O_3, and O_4 are also able to consume a new input only every $500\,ms$, i.e., O_2 at times $500\,ms, 1000\,ms, \ldots$, O_3 at times $600\,ms, 1100\,ms, \ldots$, and O_4 at times $700\,ms, 1200\,ms, \ldots$ Since $e_4 = 400\,ms < 500\,ms$, O_4 is always able to process the inputs faster than they are available, resulting in zero queuing delays between each pair of operators. In other words, ignoring delays prior to the first operator, the worst-case response time is the sum of all execution times, i.e., $1100\,ms$.

Suppose we wish to further improve the overall response time of this linear chain. We decide to optimize the implementation of the first operator, as it has the greatest execution time. Succeeding, we manage to reduce its execution time from $e_1 = 500\,ms$ to $e_1' = 300\,ms$. We expect an improvement of at least $200\,ms$ in the end-to-end response time. In reality, though, the linear chain's worst-case response time is now $500\,ms$ more than that of the original configuration. The new configuration is shown in Fig. 3 (bottom). A lower worst-case execution time for the first operator shifts the "bottleneck" of the linear chain from the first operator to the last operator, causing queuing delays throughout the chain (which was not the case before), resulting in this performance anomaly. The queuing delays shown in the figure are not encountered by the first few inputs, but the queues gradually start filling up, starting at the last operator, eventually resulting in these queuing delays.

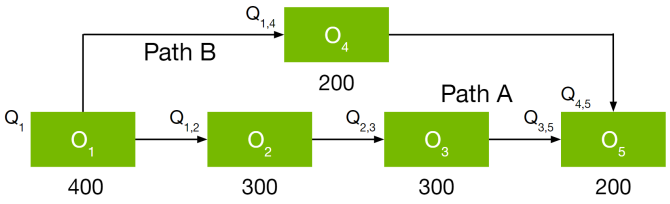

Fig. 4. An example DAG where the interaction of two paths results in a greater execution time that either have individually.

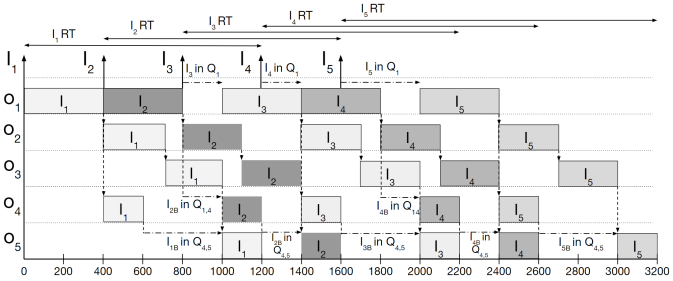

Fig. 5. Schedule for the DAG in Fig. 4, when inputs $I_1, I_2, \ldots, I_5$ are released every $400\,ms$. Release times are indicated by upward-pointing arrows and response-time intervals are shown above. I_1 refers to the first input, while I_{1B} refers to the first input when moving through path B. Data transfer between two operators is indicated by a vertical dashed line, with a horizontal dashed line representing queuing and showing the name of both queue and item.

The response times in this example are calculated using our linear chain analysis presented in Section V.

In the next example, we motivate why a worst-case response-time analysis for DAGs with both upstream and downstream conditions is nontrivial and requires a careful consideration of multiple possible scenarios, and that individually analysing all linear chains (paths) in a DAG is insufficient.

Example 3. Consider the DAG shown in Fig. 4 consisting of five operators O_1–O_5, with O_1 being the source and O_5 being the sink, and two paths from source to sink, Path A, $O_1 \to O_2 \to O_3 \to O_5$, and Path B, $O_1 \to O_4 \to O_5$. The operator execution times are $e_1 = 400\,ms$, $e_2 = e_3 = 300\,ms$, and $e_4 = e_5 = 200\,ms$. A new input arrives every $400\,ms$.

Path A is clearly the longer path among the two and, if analyzed individually as a linear chain, yields a worst-case response time of $1200\,ms$. However, as the timing diagram in Fig. 5 suggests, the response time of an input can be as high as $1600\,ms$. Further examination of the diagram reveals why this is the case: during the time window $[800\,ms, 1000\,ms)$, O_1 is blocked from executing I_3 because $Q_{1,4}$ is full (i.e., holding I_{2B}), meaning that execution is forbidden by the downstream condition; $Q_{1,4}$, in turn, is full because $Q_{4,5}$ is full (i.e., holding I_{1B}), meaning that O_4 cannot execute. In short, because Path B is shorter than Path A, items arrive at O_5 from Path B more quickly than from Path A, periodically blocking O_1's execution and increasing the worst-case response time.

Hence, in a DAG, one of the paths may cause delays in another path due to precedence constraints, and, in conjunction with downstream conditions, these delays may affect input processing in a subsequent iteration. As a result of such

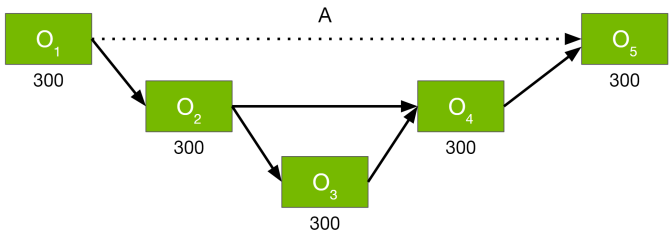

Fig. 6. Example showing that, due to downstream conditions, adding an edge to the DAG can significantly change its end-to-end response time (both increase or decrease depending on the operator execution times).

interactions, the actual processing rate of an operator (O_1 in this example) may depend on all paths leading out of it up to the point where the paths merge (O_5 in this example).

In the final example, we show that even simple changes to the application program, such as adding or dropping a data flow edge in the program DAG, can change the response time in unintuitive ways. We also highlight the limitations of Holoscan's latency profiling tool using this example.

Example 4. Consider the DAG shown in Fig. 6. We consider two versions of the DAG, with and without the edge labeled `A`. For each version, we use Holoscan's Data Flow Tracking tool to profile the response times across several iterations, and report the maximum observed response time. With edge `A`, the maximum observed response time is $2700\,ms$. Removing the edge reduces it to $2400\,ms$. To make matters worse, if the execution time of the second operator O_2 is increased from $300\,ms$ to $700\,ms$, removing edge `A` actually increases the maximum observed response time from $3500\,ms$ to $3600\,ms$. These complex timing behaviors highlight that application developers may overlook many potential timing issues during the design phase and encounter them only post-development (as the profiling tool cannot be used at design time).

Furthermore, Holoscan's Data Flow Tracking tool (or any empirical tool as a matter of fact) may need to be run for an extended duration to achieve a trustworthy and reliable upper bound on the response time. For example, consider the same DAG as shown in Fig. 6 but without edge `A`, and suppose that operators O_1-O_5 have execution times $191\,ms$, $128\,ms$, $215\,ms$, $300\,ms$, and $414\,ms$, respectively. Holoscan's profiling tool required seven hours to converge on the worst-case response time bound for this DAG. Initial measurements after 30 minutes recorded a maximum response time of $2069\,ms$, which increased to $2075\,ms$ after two hours, and reached $2080\,ms$ after seven hours. Of course, with empirical tools, it is unclear if the observed bounds may increase further with a more prolonged execution. Therefore, a rigorous response-time analysis is essential, especially in critical and regulated sectors like medical devices and industrial automation, which is currently missing for Holoscan-like frameworks.

IV. System Model

We model each Holoscan application as a directed acyclic graph $D = (V, E)$, where the set of all vertices V corresponds to the set of all Holoscan operators and the set of all edges

E corresponds to the set of all precedence relationships. Specifically, V is a *partially ordered set* of operators, such that if there is an edge from O_x to O_y, then O_x is before O_y in V. A *segment* $S \subseteq V$ (also called a *path*) is an ordered set of operators such that each adjacent pair of operators $O_x, O_y \in S$ is connected by an edge from O_x to O_y.

Each operator $O_i \in V$ is characterized by its best-case and worst-case execution times, denoted e_i^{lb} and e_i^{ub}, respectively. If the application uses two instances of the same Holoscan operator, they are mapped to separate vertices in V. We assume that D has exactly one source operator and exactly one sink (terminal) operator, which we commonly denote as O_{src} and O_{sink}, respectively. Any DAG with multiple sources or sinks can be easily reduced to our model by appending auxiliary source and sink nodes with zero execution times.

Each edge $E_{i,j} = (O_i \to O_j)$ is characterized by its sending and receiving operators O_i and O_j, respectively. The edges imply a dependency in the Holoscan application, e.g., O_j needs an input from O_i to execute. However, since O_j may need to buffer this input until all conditions are met (discussed below), we assume that each edge $E_{i,j}$ has a dedicated FIFO queue $Q_{i,j}$ with capacity $C_{i,j}$. The FIFO queue abstracts the double-buffered queue in the Holoscan implementation.

Both message available (upstream) and downstream conditions are typically set with a default threshold of one message. For example, if operator O_i has three incoming edges, $E_{a,i} = (O_a \to O_i)$, $E_{b,i} = (O_b \to O_i)$, and $E_{c,i} = (O_c \to O_i)$, and two outgoing edges $E_{i,x} = (O_i \to O_x)$ and $E_{i,y} = (O_i \to O_y)$, the message available condition on each incoming edge of O_i ensures that it can start execution only if there is an input available in each incoming queue $Q_{a,i}$, $Q_{b,i}$, and $Q_{c,i}$, and the downstream condition ensures that O_i does not execute until there is an empty slot available in each outgoing queue $Q_{i,x}$ and $Q_{i,y}$. Holoscan's scheduler is work conserving, i.e., if for any operator O_i both message available and downstream conditions hold, the operator must execute.

Assumptions. We make four key assumptions based on how Holoscan is deployed today. **A1.** We assume that the queue capacity is 1 for each queue $Q_{i,j}$, as Holoscan currently does not allow edge-specific queue sizes by default, and it is recommended to use the default queue capacity of 1 for an application. **A2.** We assume that the platform contains sufficient cores so that every operator can be executed in parallel. This is the de facto deployment scenario as the NVIDIA embedded platforms in use today typically have twelve or more cores whereas the Holoscan applications have fewer operators (e.g., see Figure 9 in Section VII).[3] **A3.** We assume that the Holoscan application, or DAG D, continuously processes an infinite stream of inputs. The inputs may arrive as frequently as possible with a minimum inter-arrival time of 0. However, this implies that some inputs may be dropped at source. For our worst-case response-time analysis, we ignore such inputs. Specifically, we assume that if O_{src} is idle and all its downstream conditions are satisfied (note that O_{src} does not have any upstream conditions), it never needs to wait for an input; and input will always be available. This also means that periodic arrivals are possible. For example, the period of $400\,ms$ in Fig. 5 is the maximum period allowed for this graph, since any increase would cause the source operator to become idle at some point in the execution history. **A4.** We assume that the execution time of an operator, irrespective of the input it is processing, remains fixed throughout a single streaming run (during which the DAG may process several inputs continuously). This can be easily achieved by spinning (if needed) at the end of each operator execution.

[3]In short, we leave the problem of interference and scheduling of more operators on fewer cores for future work; rather, our goal in this paper is to understand how the combination of upstream and downstream edge conditions affect the worst-case response times of a Holoscan application.

Problem statement. Suppose D processes an infinite stream of inputs $I = \{I_1, I_2, \ldots\}$, where each input I_k is queued in front of the source operator O_{src} at time $start_k$ and is finished being processed by the sink operator O_{sink} at time $finish_k$. Let $I_{dropped} \subseteq I$ denote the set of inputs in I that are dropped. For each input $I_x \in I_{dropped}$, we can ignore its start and finish times $start_x$ and $finish_x$, or let $start_x = finish_x = \infty$. Let $R(D)$ denote the worst-case response time (WCRT) experienced by any input that is successfully processed through the DAG. Given assumptions **A1**-**A4**, our objective is to find an upper-bound $R^{ub}(D)$ on this worst-case response time, i.e.,

$$R^{ub}(D) \geq R(D) = \max_{\forall I_k \in I \setminus I_{dropped}} (finish_k - start_k). \quad (1)$$

MAX and OPT scenarios. Despite assumption **A4**, users may want to analyze different DAG instances with different execution time profiles, e.g., to understand when is it beneficial to optimize and reduce the execution time of certain operators. Also, the platform configuration, such as the clock frequencies, may affect the execution times. Hence, we consider two distinct scenarios: MAX and OPT. Let $e_i \in [e_i^{lb}, e_i^{ub}]$ denote the execution time of operator O_i in every iteration of a single streaming run. In the MAX scenario, we assume that $e_i = e_i^{ub}$ is always the *maximum* possible value. In the OPT scenario, we assume that $e_i \in [e_i^{lb}, e_i^{ub}]$, say, due to some optimizing configurations. To see why OPT scenarios are relevant, recall Example 2 from Section III. The OPT scenario allows identifying performance anomalies at design time, which helps developers pursue productive optimizations.

V. LINEAR CHAIN MODELING

We divide our response-time analysis into two parts. We discuss linear chains first and then in the next section extend our linear chain analysis for arbitrary DAGs. We model a linear chain using a simplified DAG.

$$\begin{aligned} D_{chain} = (V_{chain}, E_{chain}) : \\ V_{chain} = \{O_1, O_2, \ldots, O_N\}, \\ E_{chain} = \{(O_1 \to O_2), \ldots, (O_{N-1} \to O_N)\}, \\ O_{src} = O_1 \text{ and } O_{sink} = O_N. \end{aligned} \quad (2)$$

We start with some preliminary definitions.

Definition 1. Given a segment $S = \{O_{k+1}, \ldots, O_{k+x}\} \subseteq V_{chain}$ of length x, with $k \geq 0$ and $k + x \leq N$, *bottleneck* $\mathcal{B}(S)$ denotes the operator in S with the maximum execution time (ties broken in favor of the preceding operator), i.e.,

$$\mathcal{B}(S) = O_b \text{ s.t. } e_b^{ub} = \max_{1 \leq i \leq x} (e_{k+i}^{ub}) \text{ and}$$
$$\forall j \in [1, x], e_b^{ub} = e_{k+j}^{ub} \implies b \leq k + j. \quad (3)$$

Definition 2. For any DAG $D = (V, E)$, the *service rate* μ_i of any operator $O_i \in V$ is defined as the inverse of its worst-case execution time, i.e., $\mu_i = 1/e_i^{ub}$.

Recall from Section IV, assumption **A3**, that the source operator O_1 never needs to wait for an input, and may receive inputs as frequently as possible. If inputs arrive faster than the rate at which they can be processed, they may be dropped, and are not considered by our response-time analysis. However, subsequent operators in the linear chain may have to wait for an input owing to processing delays in the earlier part of the linear chain. For example, in a chain with only two operators O_1 and O_2, if $e_1 = 100\,ms$ and $e_2 = 1000\,ms$, because $e_1 < e_2$, assumption **A3** implies that O_2 too never needs to wait for an input. In contrast, if $e_1 = 1000\,ms$ and $e_2 = 100\,ms$, because $e_1 > e_2$, O_2 needs to wait for O_1 to finish executing. Generalizing this example, the bottleneck operator in the linear chain (as computed using Definition 1) limits the service rate $\mu(D_{chain})$ of the entire linear chain.

Definition 3. For any $D_{chain} = (V_{chain}, E_{chain})$, its service rate $\mu(D_{chain})$ is defined as the service rate of its bottleneck operator, i.e., if $O_b = \mathcal{B}(V_{chain})$, $\mu(D_{chain}) = \mu_b$.

A. Linear Chain with MAX Execution Times

In the following, D_{chain} refers to the linear chain with N operators as defined in Eq. (2), and O_b is commonly used to refer to its bottleneck operator $\mathcal{B}(V_{chain})$. The bottleneck operator allows us to divide a linear chain into two segments and analyse the response time of each segment separately. We first consider a special case in Lemma 1 where the bottleneck operator is the last operator, i.e., $O_b = O_N$.

Lemma 1. *Given D_{chain}, if $O_b = O_N = \mathcal{B}(V_{chain})$, the worst-case response time in the* MAX *case is bounded by*

$$R_{MAX}^{ub}(D_{chain}) = e_b^{ub} \cdot (b + 1). \quad (4)$$

Proof. Consider O_b and its preceding operator, O_{b-1}. Since O_b is the bottleneck, $\mu_{b-1} > \mu_b$. As such, O_{b-1} will produce its outputs more often than O_b can consume its inputs. This causes the number of inputs in queue $Q_{b-1,b}$ between these operators to grow without bound and eventually overflow.

However, since all operators are subjected to a default downstream condition, an operator does not start execution if there is no space to push an item to the queue of the communication buffer. Therefore, the service rate of O_{b-1} effectively decreases to $\mu'_{b-1} = \mu_b$. As O_b is the bottleneck and has the minimum service rate, we can also conclude that $\mu_{b-2} > \mu'_{b-1} = \mu_b$. Hence, the service rate of O_{b-2} also effectively decreases to $\mu'_{b-2} = \mu_b$.

More generally, for each operator O_{b-m} in the linear chain ($1 \leq b - m < b$), its service rate $\mu_{b-m} > \mu'_{b-m+1} = \mu_b$; and in order for the downstream condition to hold across the chain, its effective service rate decreases to $\mu'_{b-m} = \mu_b$.

Since we have b operators in D_{chain}, and each operator's effective service rate decreases to μ_b, i.e., each operator releases an output only every e_b^{ub} units of time, the longest time it takes an input to move through the linear chain is $e_b^{ub} \cdot (b + 1)$, corresponding to b operators and the queue preceding O_1. □

We consider another special case in Lemma 2. We consider a linear chain D'_{chain} of length $N + 1$. D'_{chain} is identical to D_{chain} for up to the first N operators, but has an extra operator O_{N+1} and an extra edge $(O_N \rightarrow O_{N+1})$. However, both chains share the same bottleneck operator O_N.

Lemma 2. *Given D_{chain} as in Lemma 1, and another linear chain $D'_{chain} = (V_{chain} \cup \{O_{N+1}\}, E_{chain} \cup \{(O_N \rightarrow O_{N+1})\})$, if $O_b = O_N = \mathcal{B}(V_{chain}) = \mathcal{B}(V'_{chain})$, the worst-case response time of D'_{chain} in the* MAX *case is bounded by*

$$R_{MAX}^{ub}(D'_{chain}) = R_{MAX}^{ub}(D_{chain}) + e_{N+1}^{ub}. \quad (5)$$

Proof. Since $O_b = O_N$ is also the bottleneck operator in the new linear chain D'_{chain}, the rate at which operator O_N produces its outputs is lower than the rate at which the newly added operator O_{N+1} consumes its inputs. Hence, no data item will ever have to wait in queue $Q_{N,N+1}$ between these operators, i.e., no queuing delay can occur between operators O_N and O_{N+1}. Hence, the worst-case response-time bound of D'_{chain} is simply the worst-case response time-bound of D_{chain}, i.e., $R_{MAX}^{ub}(D_{chain})$, plus the worst-case execution delay at the last operator O_{N+1}, i.e., e_{N+1}^{ub}. □

We generalize Lemma 2 by considering another linear chain D''_{chain}, which, like before, is an extended version of D_{chain} with $N+1$ operators, and with the same bottleneck operator as D_{chain}, i.e., $O_b = \mathcal{B}(V_{chain}) = \mathcal{B}(V''_{chain})$. However, unlike before, the bottleneck operator may not necessarily be the N^{th} operator O_N, but can be any operator from O_1 to O_N.

Lemma 3. *Given D_{chain}, and another linear chain $D''_{chain} = (V_{chain} \cup \{O_{N+1}\}, E_{chain} \cup \{(O_N \rightarrow O_{N+1})\})$, if $O_b = \mathcal{B}(V_{chain}) = \mathcal{B}(V''_{chain})$ for some $1 \leq b \leq N$, the worst-case response time of D''_{chain} in the* MAX *case is bounded by*

$$R_{MAX}^{ub}(D''_{chain}) = R_{MAX}^{ub}(D_{chain}) + e_{N+1}^{ub}. \quad (6)$$

Proof. The effective rate at which D_{chain} produces its outputs is $\mu(D_{chain}) = \mu_b$ (Definition 3). Since $\mu_b < \mu_{N+1}$, the remaining proof is analogous to that of Lemma 2. □

In Lemma 4, we use Lemmas 1 to 3 to upper-bound the worst-case response time for D_{chain}, without making any assumptions about which operator is the bottleneck.

Lemma 4. *The worst-case response time of D_{chain} in the* MAX *case is bounded by*

$$R^{ub}_{MAX}(D_{chain}) = e^{ub}_b \cdot (b+1) + \sum_{i=b+1}^{N} e^{ub}_i. \quad (7)$$

Proof. Let $D^b_{prefix} = (V^b_{prefix}, E^b_{prefix})$ denote a sub-chain of D_{chain}, with $V^b_{prefix} = \{O_1, O_2, \ldots, O_{b-1}, O_b\}$. As D^b_{prefix} is itself a chain ending in a bottleneck operator, Lemma 1 gives us the following upper bound on its response time:

$$R^{ub}_{MAX}(D_{prefix}) = e^{ub}_b \cdot (b+1). \quad (8)$$

Now consider $D^{b+1}_{prefix} = (V^b_{prefix} \cup \{O_{b+1}\}, E^b_{chain} \cup \{(O_b \rightarrow O_{b+1})\})$. From Lemma 2, and from the upper bound on the response time of D^b_{prefix} from Eq. (8), we have:

$$\begin{aligned} R^{ub}_{MAX}(D^{b+1}_{prefix}) &= R^{ub}_{MAX}(D^b_{prefix}) + e^{ub}_{b+1} \\ &= e^{ub}_b \cdot (b+1) + e^{ub}_{b+1}. \end{aligned} \quad (9)$$

Next, we consider $D^{b+2}_{prefix} = (V^b_{prefix} \cup \{O_{b+1}, O_{b+2}\}, E^b_{chain} \cup \{(O_b \rightarrow O_{b+1}), (O_{b+1} \rightarrow O_{b+2})\})$. This time, from Lemma 3 and from Eq. (9) above, we have:

$$\begin{aligned} R^{ub}_{MAX}(D^{b+2}_{prefix}) &= R^{ub}_{MAX}(D^{b+1}_{prefix}) + e^{ub}_{b+2} \\ &= e^{ub}_b \cdot (b+1) + e^{ub}_{b+1} + e^{ub}_{b+2}. \end{aligned} \quad (10)$$

Adding one operator at a time to D_{prefix} for up to $N - b$ iterations, and applying Lemma 3 every time like in Eq. (10), we can have an upper-bound on the response time of D^N_{prefix} as

$$R^{ub}_{MAX}(D^N_{prefix}) = e^{ub}_b \cdot (b+1) + \sum_{i=b+1}^{N} e^{ub}_i, \quad (11)$$

which also applies to D_{chain}, as $D^N_{prefix} = D_{chain}$. □

B. Linear Chain with OPT *Execution Times*

In the previous section, we showed that the WCRT of a linear chain D_{chain} in the MAX case depends on the bottleneck operator. In the OPT case, the bottleneck operator of the same chain may be different, and hence the WCRT may be different and needs to be separately derived.

To facilitate response-time analysis in the OPT case, we define a parameterized instance of $D_{chain} = (V_{chain}, E_{chain})$, parameterized by function λ. We use $D_{chain}(\lambda) = (V_{chain}(\lambda), E_{chain}(\lambda))$ to denote this parameterized instance. While each operator O_i in V_{chain} required e^{ub}_i time to process an input in the MAX case, each operator O_i in $V_{chain}(\lambda)$ requires $\lambda(e^{lb}_i, e^{ub}_i) \in [e^{lb}_i, e^{ub}_i]$ time to process an input in the OPT case. Note that in both cases, the operator execution time remains constant in all iterations for all inputs.

The bottleneck operator may vary for different definitions of λ. The WCRT of D_{chain} in the OPT case is thus upper-bounded by the maximum MAX-case WCRT across all possible definitions of λ, i.e.,

$$R^{ub}_{OPT}(D_{chain}) = \max_{\forall \lambda} R^{ub}_{MAX}(D_{chain}(\lambda)). \quad (12)$$

The number of possible definitions of λ is exponentially large, depending on the number of operators and their range of execution times. However, since D_{chain} contains N operators, there can be only up to N different bottleneck operators. Suppose λ_b denotes any definition for which the bottleneck is operator O_b. Thus, we can refine Eq. (12) as follows.

$$R^{ub}_{OPT}(D_{chain}) = \max_{b=1}^{n} \max_{\forall \lambda_b} R^{ub}_{MAX}(D_{chain}(\lambda_b)). \quad (13)$$

Lemma 5. *If λ^{max}_b is defined as follows:*

$$\lambda^{max}_b(e^{lb}_b, e^{ub}_b) = e^{ub}_b, \quad (14)$$

$$\forall i < b: \ \lambda^{max}_b(e^{lb}_i, e^{ub}_i) = \min(e^{ub}_b - 1, e^{ub}_i), \quad (15)$$

$$\forall j > b: \ \lambda^{max}_b(e^{lb}_j, e^{ub}_j) = \min(e^{ub}_b, e^{ub}_j), \quad (16)$$

then λ^{max}_b also yields the maximum WCRT among all possible definitions of λ_b, i.e.,

$$R^{ub}_{MAX}(D_{chain}(\lambda^{max}_b)) = \max_{\forall \lambda_b} R^{ub}_{MAX}(D_{chain}(\lambda_b)). \quad (17)$$

Proof. From Eq. (7) in Lemma 4, we know that the WCRT for a given chain is entirely dependent on the execution times of the bottleneck and of every operator following it. Thus, we can construct a maximal λ_b by adjusting the execution time of every operator in the chain. Eq. (14) maximizes the first term on the right-hand side of Eq. (7). Eq. (15) is a necessary condition, as otherwise O_b would not be the bottleneck. Eq. (16) maximizes the second term on the right-hand side of Eq. (7) while ensuring that O_b is still the bottleneck. □

If any operator's lower bound on execution time is greater than the upper bound of O_b, then no λ_b exists, as it is impossible for O_b to be the bottleneck. In this case, λ_b will not be considered at all in Eq. (13) and in Eq. (18) below.

Since λ^{max}_b is easy to compute using Eqs. (14) to (16), we can also compute $R^{ub}_{OPT}(D_{chain})$ efficiently using:

$$R^{ub}_{OPT}(D_{chain}) = \max_{b=1}^{n} R^{ub}_{MAX}(D_{chain}(\lambda^{max}_b)). \quad (18)$$

VI. DAG MODELING

Recall Example 3 from Section III. We demonstrated that using a naive approach of enumerating all paths in the DAG, deriving an upper bound on the response time of each path using a linear chain analysis, and then computing the maximum value among these upper bounds can result in an unsafe upper bound on the worst-case response time of the DAG. In other words, two or more paths in a DAG may interact through both upstream and downstream conditions to result in additional delays that do not exist in linear chains. We therefore define a different response-time analysis for DAGs.

Analogous to the worst-case execution time of an operator, we first define the *worst-case inter-processing delay* for each operator in the DAG. This differs from the worst-case execution time when two or more paths diverge from that operator. For instance, consider an operator O_f from which P paths diverge, which then converge back at operator O_g. Suppose each path i corresponds to a segment $S_i = \{O_f, O_{i_1}, O_{i_2}, \ldots, O_{i_{n_i}}, O_g\}$ consisting of n_i operators other than O_f and O_g.

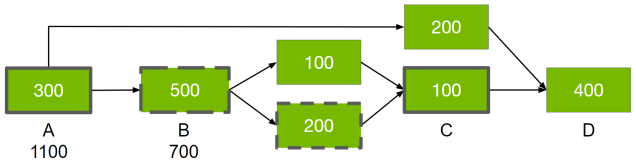

Fig. 7. An example showing an inter-processing delay calculation for the operators A and B. The number contained within each operator represents its execution time. As per Lemma 6, the inter-processing delay of B is defined as the longest path in terms of execution time from B to C, excluding the execution time of C. This comes to $\delta_B^{ub} = 500 + 200 = 700$, with the operators used in the calculation shown with dashed outlines. The longest path from A to D includes both the operators with a dashed outline and those with a bold outline. Thus, A's inter-processing delay is $\delta_A^{ub} = 300 + 500 + 200 + 100 = 1100$.

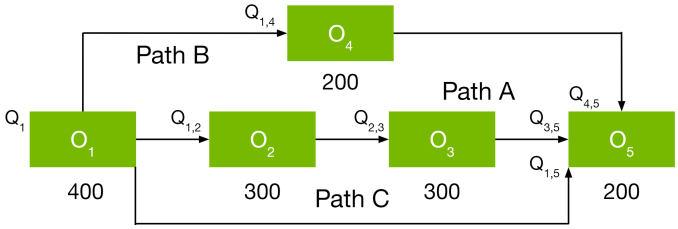

Fig. 8. A modified version of Fig. 4, with a new path from source to sink (Path C). Despite the difference in structure, the two graphs yield identical worst-case inter-processing delays as per our definition in Lemma 6.

Definition 4. Assuming operator O_f never waits for inputs, and assuming operator O_g incurs no delays due to downstream conditions, the *worst-case inter-processing delay* δ_f^{ub} of O_f is defined as the maximum delay between any two of its consecutive outputs. Formally, if O_f finishes processing the k^{th} input at time t_k, then $\delta_f^{ub} = \max_k(t_{k+1} - t_k)$.

Lemma 6. *δ_f^{ub} is the sum of O_f's own WCET and the WCETs of all operators on the longest path up to and excluding O_g, i.e.,*

$$\text{if} \quad x = \arg\max_{i=1}^{P} \sum_{k=1}^{n_i} e_{i_k}^{ub} \quad \text{then} \quad \delta_f^{ub} = e_f^{ub} + \sum_{k=1}^{n_x} e_{x_k}^{ub}. \quad (19)$$

Proof. Consider the case where O_f has a direct edge to O_g, i.e., some path P_y consisting of no other operators (e.g., edge A in Fig. 6). This is the worst case scenario as O_f cannot start processing an input until queue $Q_{f,g}$ is empty, but O_g cannot start processing the input from $Q_{f,g}$ before inputs from all other paths are available, including from some path P_x (which is the longest path between O_f and O_g).

Hence, in the worst case, the entire path latency of P_x is added to O_f's worst-case inter-processing delay, in addition to O_f's own WCET. That is, $\delta_f^{ub} = e_f^{ub} + \sum_{k=1}^{n_x} e_{x_k}^{ub}$. □

See Fig. 7 for an example inter-processing delay calculation. For operators from which multiple paths do not diverge, i.e., they have only one following operator (like in a linear chain), their worst-case inter-processing delay is same as their worst-case execution time. Formally, as well, in the definition of δ_f^{ub} in Lemma 6, the summation term is zero as there are no paths with additional operators between O_f and O_g.

Corollary 1. *If O_f has only one outgoing edge $(O_f \to O_g)$ (like in a linear chain), then $\delta_f^{ub} = e_f^{ub}$.*

Note that Lemma 6 defines δ_f^{ub} pessimistically. It assumes there exists an edge between O_f and O_g even though there may not be such an edge in reality. For instance, when defining the worst-case inter-processing delay for O_1 in the DAG illustrated Fig. 4, Lemma 6 assumes a DAG as shown in Fig. 8 with an additional path C directly connecting O_1 and O_5.

A consequence of this pessimistic assumption is that the operator with the maximum worst-case inter-processing delay, say O_x, may not necessarily be the bottleneck, as its worst-case delay may never be realized in practice. Instead, a different operator, say O_y, with a smaller but tighter maximum worst-case inter-processing delay may become the bottleneck, resulting in a different worst-case response-time bound. This new bound may in fact be worse if O_y is positioned much later in the DAG, i.e., $y \gg x$ (assuming operators are numbered incrementally from source to sink), which forces a longer prefix of operators to process their respective inputs at O_y's rate.

For instance, recall Example 3; for the DAG illustrated in Fig. 4, Lemma 6 computes a worst-case inter-processing delay of $\delta_1^{ub} = 1000\,ms$ for operator O_1. However, the schedule in Fig. 5 suggests that the effective maximum inter-processing delay of O_1 is only $600\,ms$ (e.g., interval $[800\,ms, 1400\,ms)$ from the time when O_1 finishes processing I_2 to the time when it finishes processing I_3), and the resulting worst-case response time is $1600\,ms$ (e.g., that of I_5 from its arrival at $1600\,ms$ to its completion at $3200\,ms$). Now, suppose we add a new operator O_6 after the sink node O_5, with an edge $O_5 \to O_6$, and with the highest execution time $e_6 = 900\,ms$. The worst-case inter-processing delay for O_6 is $\delta_6^{ub} = e_6 = 900\,ms$, which is a tight bound, unlike δ_1^{ub}. Hence, although $\delta_1^{ub} > \delta_6^{ub}$, O_6 becomes the bottleneck in practice. Moreover, since it is the last operator, it limits the processing rate of all previous operators, resulting in a much higher worst-case response time of $4500\,ms$ (as analyzed using Lemma 11 later in this section). Hence, compared to our linear chain analysis, we adopt a more nuanced approach for analyzing DAGs that considers all possible operators as bottleneck candidates.

A. DAG with MAX Execution Times

We compute the maximum possible response time assuming that operator O_b with a worst-case inter-processing delay of δ_b^{ub} is the bottleneck (Lemmas 7 to 10). Once we have a formula for the worst-case response time of that DAG assuming O_b is the bottleneck (Lemma 10), we apply this bound to all operators in D, taking the maximum to be the worst-case response time of the whole DAG (Lemma 11).

Lemma 7. *Let P_{after} denote the set of all paths from O_b to O_{sink}. Each path i corresponds to a segment $S_i = \{O_b, O_{i_1}, O_{i_2}, \ldots, O_{i_{n_i}}, O_{sink}\}$ consisting of n_i operators other than O_b and O_{sink}. Let $\Delta_{after\ O_b}$ denote an upper bound on the maximum delay that any input through D incurs after*

being processed by O_b. $\Delta_{after\ O_b}$ is the sum of all execution times except that of O_b on the slowest path in P_{after}, i.e.,

$$\Delta_{after\ O_b} = \sum_{k=1}^{n_x} e^{ub}_{x_k} + e_{sink} \text{ where } x = \underset{i=1}{\overset{|P_{after}|}{\operatorname{argmax}}} \sum_{k=1}^{n_i} e^{ub}_{i_k}. \tag{20}$$

Proof. We proceed similar to the proof of Lemma 2. Since O_b is the bottleneck operator, it produces its output slower than any downstream operator, at a rate upper-bounded by its worst-case inter-processing delay. Hence, no data item has to wait in any of its downstream queues. Thus, the worst-case delay $\Delta_{after\ O_b}$ incurred by any input after O_b is simply the sum of all execution times on the slowest path in P_{after}. □

Lemma 8. *Let P_{before} denote the set of all paths from O_{src} to O_b. Each path k corresponds to a segment $S_k = \{O_{src}, O_{k_1}, O_{k_2}, \ldots, O_{k_{m_k-2}}, O_b\}$ consisting of m_k operators including O_{src} and O_b. Let $\Delta_{before\ O_b}$ denote an upper bound on the maximum delay that any input through D incurs before being processed by O_b. $\Delta_{before\ O_b}$ depends on the length of the shortest path in P_{before} and is defined as follows:*

$$\Delta_{before\ O_b} = \delta^{ub}_b \cdot m_y \text{ where } y = \underset{k=1}{\overset{|P_{before}|}{\operatorname{argmin}}} m_k. \tag{21}$$

Proof. In the worst case, all operators in all paths in P_{before} are either directly or transitively waiting on a downstream queue just before O_b, which is the bottleneck. When this happens, operators in the paths in P_{before} are not processing any inputs but simply holding an input in their respective input queues.

It must be noted that all paths in P_{before} derive their inputs simultaneously from O_{src}. Therefore, all paths in P_{before} must always hold the same number of inputs at any point of time. For example, if the lengths of two paths in P_{before} are 4 and 5, then maximum 4 inputs will be processed and held in each of the two paths. Hence, the number of pending inputs in P_{before} is determined by the shortest path, i.e., the path with the fewest number of operators (or the fewest number of queues).

Furthermore, as we show in the proof of Lemma 1, the effective service rate of all operators on this path decreases to that of O_b. Hence, the longest time it takes these inputs to move through this path is upper-bounded by $\delta^{ub}_b \cdot m_y$, where m_y is the length of the shortest path before and including O_b. □

Lemma 9. *Let P denote the set of all paths from O_{src} to O_{sink}. Each path j corresponds to a segment $S_j = \{O_{src}, O_{j_1}, O_{j_2}, \ldots, O_{j_{l_j}}, O_{sink}\}$ consisting of l_j operators other than O_{src} and O_{sink}. Let z be the longest path in P in terms of total execution time. Let q be the longest path in P of which O_b is a part of, i.e., $O_b \in S_q$. The maximum delay that an input through DAG D can incur increases by the difference in execution time between z and q. That is,*

$$\Delta_{longest\ O_b} = \sum_{k=1}^{l_z} e^{ub}_{z_k} - \sum_{k=1}^{l_q} e^{ub}_{q_k} \text{ where} \tag{22}$$

$$z = \underset{1\le j\le |P|}{\operatorname{argmax}} \sum_{k=1}^{l_j} e^{ub}_{j_k} \text{ and } q = \underset{1\le j\le |P| \wedge O_b \in S_j}{\operatorname{argmax}} \sum_{k=1}^{l_j} e^{ub}_{j_k}.$$

Proof. If O_b is not on the path z, then at some point, an item moving through O_b's path may need to wait for an input belonging to z to arrive to the operator where both paths join. The worst-case increase in delay due to this occurrence is equal to the difference in the execution time between the two paths, or the amount of time the item moving through q must wait for the item moving through z to arrive. If $z = q$, then the difference between their total execution times is 0. □

Lemma 10. *If y is the shortest path from O_{src} to O_b with length m_y, including O_b, and x is the slowest path from O_b to O_{sink}, with length n_x, the worst-case response time of D in the* MAX *case, assuming that O_b is the bottleneck, is bounded by*

$$R^{ub}_{MAX}(D)_b = \Delta_{before\ O_b} + e_b + \Delta_{after\ O_b} + \Delta_{longest\ O_b} \tag{23}$$

Proof. Follows from Lemmas 7 to 9. □

Lemma 11. *If D is an arbitrary DAG with r operators, the worst-case response time of D in the* MAX *case is bounded:*

$$R^{ub}_{MAX}(D) = \max_{\forall r} R^{ub}_{MAX}(D)_r \tag{24}$$

Proof. Lemma 10 bounds the worst-case response time an input can suffer in a DAG for which we know O_b is the bottleneck. Assuming each operator in the DAG to be the bottleneck, and taking the maximum over all possible WCRTs bounds the worst-case response time through the DAG. □

Using these results requires identifying, for each operator where paths diverge, the first operator at which those paths come back together (i.e., O_f and O_g in Lemma 6, respectively). This can be done by constructing the *postdominator* tree corresponding to D. The algorithm to identify dominance is quadratic in theory, but much faster in practice [16].

B. DAG with OPT *Execution Times*

Like in Section V-B, to facilitate response-time analysis in the OPT case, we define a parameterized instance $D(\lambda) = (V(\lambda), E(\lambda))$. Each operator O_i in $V(\lambda)$ requires $\lambda(e^{lb}_i, e^{ub}_i) \in [e^{lb}_i, e^{ub}_i]$ time to process an input in the OPT case. The WCRT of D in the OPT case is upper-bounded by the maximum MAX-case WCRT across all possible definitions of λ, i.e.,

$$R^{ub}_{OPT}(D) = \max_{\forall \lambda} R^{ub}_{MAX}(D(\lambda)). \tag{25}$$

Section V-B proposed an efficient method to compute the OPT-case WCRT for linear chains. We reuse this methodology for arbitrary DAGs. First, we refine Eq. (25) as follows,

$$R^{ub}_{OPT}(D) = \max_{b=1}^{n} \max_{\forall \lambda_b} R^{ub}_{MAX}(D(\lambda_b)), \tag{26}$$

where λ_b denotes any definition for which the bottleneck is operator O_b. More formally, this means any definition where:

$$R^{ub}_{MAX}(D(\lambda_b)) = \max_{\forall r} R^{ub}_{MAX}(D(\lambda_b))_r \tag{27}$$

Lemma 12. *If λ_b^{max} is defined as follows:*

$$\lambda_b^{max}(e_b^{lb}, e_b^{ub}) = e_b^{ub}, \tag{28}$$

$$\forall j > b: \ \lambda_b^{max}(e_j^{lb}, e_j^{ub}) = e_j^{ub}, \tag{29}$$

$$\forall i < b: \ \lambda_b^{max}(e_i^{lb}, e_i^{ub}) = e_i^{lb}, \tag{30}$$

$$\forall z_k: \ \lambda_b^{max}(e_{z_k}^{lb}, e_{z_k}^{ub}) = e_{z_k}^{ub}, \tag{31}$$

then λ_b^{max} also yields the maximum WCRT among all possible definitions of λ_b, i.e.,

$$R_{MAX}^{ub}(D(\lambda_b^{max})) = \max_{\forall \lambda_b} R_{MAX}^{ub}(D(\lambda_b)). \tag{32}$$

Proof. The greatest difference between this scenario and the linear chain is the presence of $\Delta_{longest\ O_b}$. This term subtracts an amount equal to $\sum_{k=1}^{l_q} e_{q_k}^{ub}$ from the WCRT, where q is defined in Lemma 9. Notice that all the execution times after O_b are added in $\Delta_{after\ O_b}$. These terms always cancel out, so we do not need to consider that part of the subtraction in the optimization. This means the only part of the subtraction remaining corresponds to the operators before O_b, the execution times of which we can minimize without affecting other parts of the equation.

We construct a maximum λ_b^{max} by adjusting the execution times of every relevant operator in the DAG.

Eq. (28) is the first part of maximizing $\Delta_{before\ O_b}$, as e_b is a component of δ_b^{ub} and is also itself part of Eq. (23).

Eq. (29) is the second part of maximizing $\Delta_{before\ O_b}$, as well as maximizing $\Delta_{after\ O_b}$. $\Delta_{before\ O_b}$ is maximized because all operators from O_b to its postdominator are a component of δ_b^{ub}, as shown in Lemma 6.

Eq. (30) is the first part of maximizing $\Delta_{longest\ O_b}$, as these are the negative execution times within that expression that do not cancel with another term.

Eq. (31) is the second part of maximizing $\Delta_{longest\ O_b}$. This maximizes the execution times along z, the path from source to sink with the greatest total execution time. Note that if $z = q$, this condition contradicts with Eq. (30) because we would be trying to maximize execution times in z while minimizing them in q. This does not matter, as if we ignore the two contradictory conditions, we are left with Eq. (28) and Eq. (29), which maximize all parts of Eq. (23) aside from $\Delta_{longest\ O_b}$, which is 0 if $z = q$. □

We can thus compute $R_{OPT}^{ub}(D)$ efficiently using:

$$R_{OPT}^{ub}(D) = \max_{b=1}^{n} R_{MAX}^{ub}(D(\lambda_b^{max})). \tag{33}$$

VII. Evaluation

We evaluate the pessimism and scalability of our analytically derived response-time upper bounds using real-world Holoscan applications and larger synthetic DAGs. We use results from discrete event simulations as well as measurements from an NVIDIA embedded platform as baselines. Specifically, we use the NVIDIA Jetson AGX Orin Developer Kit featuring a 12-core Arm Cortex-A78AE CPU and 64GB RAM for experiments. It runs an Ubuntu-based Linux for Tegra (L4T) OS, with kernel 5.15. The CPU frequency is fixed at 2.2 GHz (maximum). We run all experiments with Holoscan version 0.6.0 (we have also reproduced all results with Holoscan version 2.2.0).

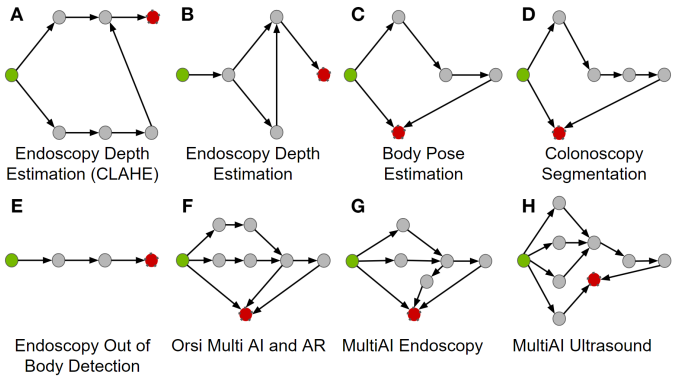

Fig. 9. Holoscan application DAGs used for evaluation. Green nodes are source operators while red nodes with dashed outlines are sink operators.

A. Workloads

Our discrete event simulator is designed to mimic the behavior of the Holoscan framework. Each application is run for 30 minutes of simulated time. These simulations are also used in the scalability study to efficiently find WCRTs of large synthetic DAGs to compare against our analysis. For measurements on the NVIDIA hardware, we encode the graph structures into Holoscan applications using placeholder operators that busy-wait for their designated execution time upon each invocation. We run each graph variation for 30 minutes and measure the performance using Holoscan's Data Flow Tracking feature. We assume that new input is always available to the source operator immediately after it consumes its previous input. For our theoretical bounds, we use the analyses described in the previous sections. The average time to compute a response-time bound (outside of the scalability study) is $0.4\,ms$.

As part of evaluation on the NVIDIA hardware, we also characterize the overheads associated with running a Holoscan application, which correspond to the closed-source part of the GXF backend engine, Linux scheduling, and related system software underneath – components we have not modeled as part of our response-time analysis (Section VII-C). Our overhead analysis reveals a fairly straightforward relationship with the number of Holoscan operators in an application, but more in-depth analysis is reserved for future work.

The HoloHub repository by NVIDIA provides a number of realistic example applications in medical devices, radio signal processing and other industries, which are used as prototypes for production-quality software by commercial vendors [10]. For evaluation, we use the DAG structures extracted from these existing 50 HoloHub applications. We exclude simplistic applications (such as with only two or three operators) and eliminate isomorphic and unconnected examples, leaving only unique graphs in our evaluation set. Fig. 9 shows the final eight types of DAG structures evaluated in this work.

For each graph, we create n variations, where n is the number of operators in the graph. The variations differ in terms of the

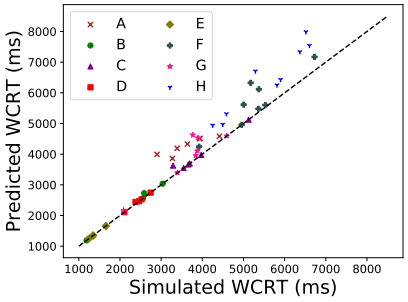

Fig. 10. Analysis versus simulation

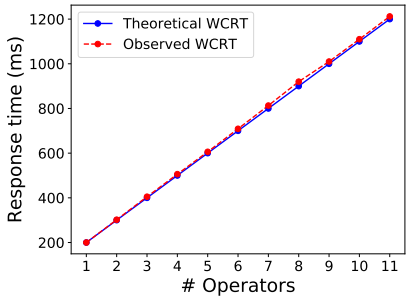

Fig. 11. Overhead measurement for linear chains

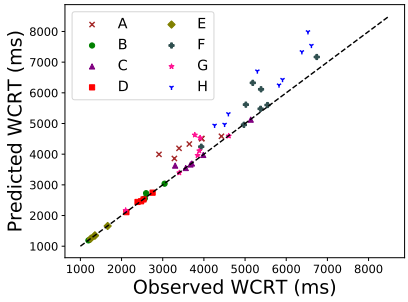

Fig. 12. Analysis versus measurements

WCET assigned to each operator, which we randomly select from the interval $[100\,ms, n * 100\,ms]$. We increase the upper bound based on the number of total operators in an application to allow a wider execution time range. Additionally, we enforce a condition that for each operator in the graph, there is one variation in which it has the greatest WCET across all operators. This is to help ensure that our randomly generated variations do not overly favor certain configurations (e.g., the source operator having the greatest WCET in every variation).

B. Evaluation in Simulation

Fig. 10 plots simulated worst-case response times against our predicted worst-case response times for different graph applications and variations. Our proposed algorithm provides a close upper bound on the WCRT for every graph variation. Most applications and variations are on or near the identity line, signifying a close match between predicted and simulated worst-case response times. We also note that no simulated WCRT is greater than our predicted upper bound.

The results show that most graph structures have a similar relationship between their corresponding predicted and simulated WCRT across all variations. Our analysis is conservative for all variations of certain graph structures, particularly Endoscopy Depth Estimation (CLAHE) (A) and MultiAI Ultrasound (H). Their simulated WCRT are several hundred milliseconds below our derived theoretical upper bounds. Our Section VI analysis makes a worst-case assumption about path length in Lemma 6, where we assume the existence of a direct edge between two operators which may not exist. This sometimes leads to an overly conservative, although safe, upper bound.

C. Evaluation on an Embedded Platform

We also evaluated HoloHub applications on an AGX Orin developer kit. The high-level difference between our simulated WCRTs and observed WCRT on AGX Orin is the presence of overheads. Here, overhead is defined as any part of the overall response time of a data item which is not the result of queuing and operator execution, and thus not part of our system model. As our response-time upper bounds can be very close to simulated WCRTs, as demonstrated in the previous section, even a minimal amount of execution overhead on real hardware may lead to underpredicting the WCRT of an application. Therefore, we perform a preliminary analysis of Holoscan system overheads to facilitate comparison of observed WCRTs with our theoretically derived bounds.

NVIDIA's Graph Execution Framework (GXF), which Holoscan uses as a backend execution engine, is partially open-source (only header files are available), restricting us from analyzing the source code to accurately model the overhead of scheduling operators belonging to a specific application in Linux. Instead, we empirically predict the overhead by carrying out experiments over different DAG structures. We experiment with increasing the length of linear chains of operators, like those shown in Fig. 3. We fix each operator's execution time at $100\,ms$ and use the default conditions at the input and output ports. Fig. 11 plots the theoretical WCRT using our response-time analysis (simple to calculate as there should be no queuing or blocking) against the observed WCRT. It shows minimal overheads due to scheduling of operators by the GXF engine on Linux. We also observed that multiple paths in a DAG do not add any extra overheads compared to the longest linear chain of operators. Based on this empirical observation, we add $3\,ms$ as overhead per operator in the longest linear chain in an application DAG, to achieve a safe and reliable upper bound for the NVIDIA hardware.

The theoretical WCRT including the heuristic overheads and the observed WCRT on AGX Orin for HoloHub application DAGs are shown in Fig. 12. Our response-time analysis again provides a close upper bound for most graphs, with no observed WCRTs exceeding our upper bounds. Although system overheads play an important role in achieving close upper bounds, they are orthogonal to our response-time analysis of the Holoscan programming framework.

D. Scalability Study

Current Holoscan applications have only a small number of operators, but this is not a limitation inherent to our system model. To evaluate how our analysis fares for larger graphs, we randomly generate 50 synthetic series-parallel DAGs of increasing size using the Open Graph Drawing Framework (specifically the 'randomSeriesParallelDAG' function) [17]. We consider series-parallel DAGs because all the HoloHub application graphs we have analyzed are series-parallel. The set of 50 graphs we consider consists of 10 graphs with 20, 40, 60, 80, and 100 edges each. We generate 10 sets of random execution times for these graphs for a total of 500 variations. We

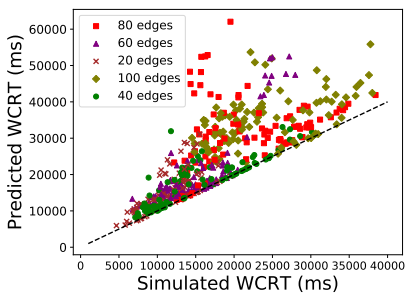

Fig. 13. Analysis vs. sim. for synthetic DAGs

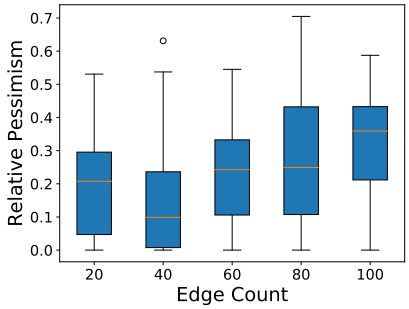

Fig. 14. Relative pessimism versus edge count

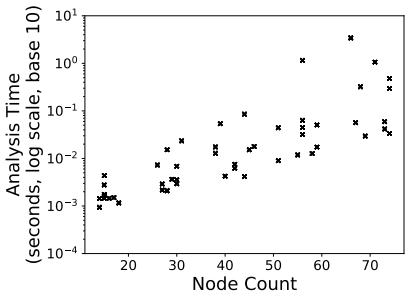

Fig. 15. Analysis time versus node count

then simulate the execution of each graph for approximately an hour of simulated time. Fig. 13 compares the WCRTs derived from simulation to those predicted by our algorithm. We observe a higher, more variable degree of pessimism.

For a clearer understanding of the pessimism incurred, Fig. 14 compares the number of edges in each graph to its *relative* pessimism. The relative pessimism of the different graphs has a weak positive correlation with edge count. Pessimism in our analysis does not strictly scale linearly with the size of the graph, but is rather inherent to the specific structure of the graph. A graph with 70 nodes may suffer less pessimism that a graph with 30, depending on the unique structure of each. Fig. 15 shows the relationship between node count and analysis time. While the analysis time grows exponentially with the node count, it remains manageable even for larger DAGs (even though we do not expect to encounter such large DAGs for NVIDIA Holoscan applications).

VIII. Related Work

Many real-time scheduling works build upon Graham's classic worst-case response time bound for a DAG task described in [18]. For example, He et al. [19] dominate the classic bound by optimizing the order in which vertices are executed, and many other works such as [20] or [21] use the bound as a pessimistic starting point to improve on. These works on traditional DAG tasks only consider precedence constraints between vertices, as their main focus is on scheduling. In contrast, we model both upstream and downstream constraints between subtasks, which makes the Graham bound *optimistic* if applied to our problem.

More similar to the specifics of our work is the literature on synchronous data flow (SDF), which was first formalized by Lee et al. [22]. SDF deals with data flow between tasks whose rates of data consumption and production are known a priori.

We believe the SDF model is general enough to allow for the modeling of Holoscan applications, but have not found a WCRT analysis for SDF that aligns with our objectives. A common approach for SDF analysis is to transform the graphs into a set of independent tasks and use existing hard-real-time scheduling algorithms [6, 7]. These techniques rely on assumptions incompatible with our system model. They require that tasks are periodic with period T_i and a relative deadline D_i such that $D_i < T_i$. This precludes the possibility of queuing, as a task will always have either finished processing or have violated its deadline by the time a new task arrives. As much of our analysis depends on delays introduced by queuing and blocking, this approach is incompatible with our model.

The authors of [8] do conduct analysis on an SDF to find a WCRT bound, but this setting also has the aforementioned periodic activations, in this case of a job consisting of multiple tasks rather than independent tasks. The SDF analysis in [9] is more similar to our own in terms of assumptions, but it focuses on throughput, while our objective is a WCRT bound.

Also similar to our work are response-time analyses of frameworks comparable to Holoscan, such as ROS 2. In particular, the relation between ROS 2 components has been modeled as a DAG in [23] and [24].

IX. Conclusion and Future Work

Our novel static analysis bounds worst-case end-to-end response times for Holoscan applications. The analysis relies on a DAG of Holoscan operators and their scheduling conditions dependent on communication data buffers. We derived an upper bound for the WCRT of a linear chain of operators and extended this approach to a DAG of operators. Our theoretical analysis was validated against real-world applications from the NVIDIA HoloHub, demonstrating that our WCRT bounds closely approximate actual performance, achieving within 99% accuracy of empirical measurements on NVIDIA hardware.

This analysis marks the first response-time analysis of Holoscan applications, helping the development of reliable medical devices. However, there is still pessimism inherent to these bounds, which could be mitigated through further refinement that relaxes some of our worst-case assumptions. While this initial evaluation was conducted on a single embedded platform, future work will assess the adaptability of our theoretical framework across various SoCs. Additionally, we plan to explore more scheduling policies that incorporate both Linux thread and GPU task management.

X. Acknowledgements

We would like to thank our anonymous shepherd and reviewers for their comments. This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC as well as the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] NVIDIA, "Holoscan SDK - GitHub," https://github.com/nvidia-holoscan/holoscan-sdk, 2024.

[2] Activ Surgical, "Activ Surgical aims to bring real-time AI to their surgery platform using NVIDIA Clara Holoscan on newly-launched NVIDIA IGX," https://www.activsurgical.com/news/nvidia-launches-igx-edge-ai-computing-platform-for-safe-secure-autonomous-systems%ef%bf%bc, 2022.

[3] FIERCE Biotech, "Medtronic adds Nvidia's AI to colorectal polyp-spotting software," https://www.fiercebiotech.com/medtech/medtronic-adds-nvidias-ai-colorectal-polyp-spotting-software, 2022.

[4] NVIDIA, "NVIDIA Jetson Orin," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2024.

[5] ——, "Data Flow Tracking," https://docs.nvidia.com/holoscan/sdk-user-guide/flow_tracking.html, 2024.

[6] M. A. Bamakhrama and T. P. Stefanov, "On the hard-real-time scheduling of embedded streaming applications," *Design Automation for Embedded Systems*, vol. 17, no. 2, p. 221–249, 2013.

[7] J. Spasic, D. Liu, E. Cannella, and T. Stefanov, "Improved hard real-time scheduling of CSDF-modeled streaming applications," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2015.

[8] J. Choi and S. Ha, "Worst-Case Response Time Analysis of a Synchronous Dataflow Graph in a Multiprocessor System with Real-Time Tasks," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 2, 2017.

[9] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," in *Sixth International Conference on Application of Concurrency to System Design*, 2006.

[10] NVIDIA, "HoloHub - GitHub," https://github.com/nvidia-holoscan/holohub, 2024.

[11] ——, "GXF Core Concepts," https://docs.nvidia.com/holoscan/archive/0.6/gxf/gxf_core_concepts.html, 2023.

[12] ——, "HoloHub - Endoscopy Tool Tracking," https://github.com/nvidia-holoscan/holohub/tree/main/applications/endoscopy_tool_tracking, 2024.

[13] ——, "Double Buffer Receiver C++ Class Definition," https://docs.nvidia.com/holoscan/sdk-user-guide/api/cpp/classholoscan_1_1doublebufferreceiver.html#exhale-class-classholoscan-1-1doublebufferreceiver, 2024.

[14] ——, "Greedy Scheduler C++ Class Definition," https://docs.nvidia.com/holoscan/sdk-user-guide/api/cpp/classholoscan_1_1greedyscheduler.html#exhale-class-classholoscan-1-1greedyscheduler, 2024.

[15] ——, "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, 2024.

[16] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, no. 1-10, pp. 1–8, 2001.

[17] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel, "The Open Graph Drawing Framework (OGDF)," in *Handbook of Graph Drawing and Visualization*, R. Tamassia, Ed. CRC Press, 2014, ch. 17.

[18] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[19] Q. He, X. Jiang, N. Guan, and Z. Guo, "Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 2283–2295, 2019.

[20] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, "Timing characterization of OpenMP4 tasking model," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2015.

[21] J. Sun, N. Guan, J. Sun, and Y. Chi, "Calculating Response-Time Bounds for OpenMP Task Systems with Conditional Branches," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019.

[22] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[23] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems*, 2019.

[24] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *IEEE Real-Time Systems Symposium*, 2021.