RRTO: A High Performance Transparent Offloading System for Model Inference on Robotic IoT

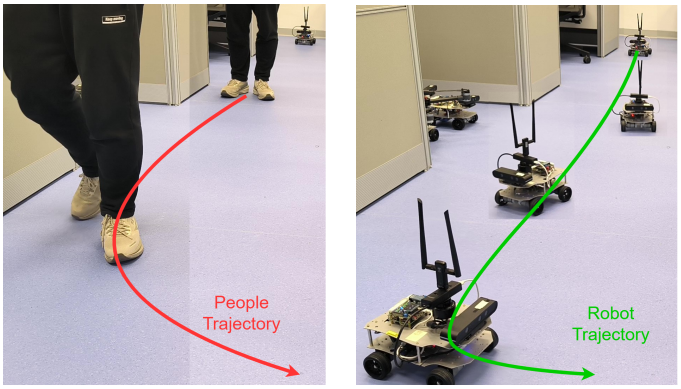
Author Names Omitted for Anonymous Review. Paper-ID [23]

***Abstract*—Fundamental robotic tasks, such as object identification and robot control, increasingly rely on Machine Learning (ML) models deployed on wireless robots, and the heavy computation of these tasks, resulting from the large number of parameters and complex operations (e.g., matrix multiplication, convolution, pooling) in ML models, is often offloaded to powerful GPU devices (e.g., cloud data centers) via the Internet of Things for these robots (robotic IoT) to achieve fast and energy-efficient inferences. Existing computation offloading systems can be categorized into two types: transparent and non-transparent methods, depending on whether source code modification is required to enable offloading. While non-transparent offloading systems have yielded success across various ML models, they also pose significant obstacles for deployment on robots, since they require considerable coding effort to modify the source code for each application and are inapplicable to closed-source applications. Meanwhile, existing transparent offloading methods can offload the function call of each operator within the ML models via Remote Procedure Call (RPC) to avoid source code modifications, but such one-by-one handling of RPCs leads to substantial communication costs in robotic IoT.**

We present RRTO, the first high performance transparent offloading system optimized for ML model inference on robotic IoT with a novel record/replay mechanism. Recognizing that the operators invoked by ML models typically follow a fixed order, RRTO automatically records the order of operators invoked by the ML model and replays the execution of these fixed-order operators during model inference. In this way, RRTO correctly calls all involved operators in the recorded order of each inference of the ML model via one RPC, circumventing the inherent communication costs caused by existing transparent offloading mechanism, without the burden of source code modification of the non-transparent offloading methods. Evaluations demonstrate that RRTO improved the performance of our real-world robotic application by 4.9x $\sim$ 48.5x and saved 8% $\sim$ 53% power consumption on our robot compared to other baselines without modifying any source code, achieving similar high performance as state-of-the-art non-transparent offloading methods.

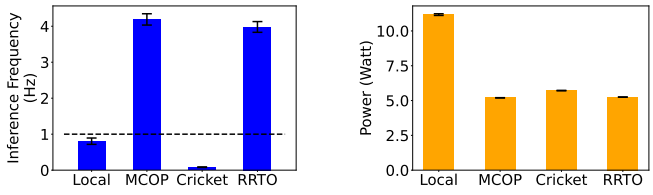
I. Introduction

The rapid advancement of machine learning (ML) methods has achieved remarkable success in various fundamental robotic tasks, such as object detection [22, 29, 30], robot control [25, 41, 46], and environmental perception [4, 26, 44]. Deploying these ML methods onto real-world robots typically requires additional hardware support due to the computationally-intensive nature of ML models (e.g., the large number of parameters, complex operations). However, directly integrating computing accelerators (e.g., GPU [36], FPGA [35], SoC [20]) onto real-world robots not only introduces additional economic costs, but also leads to increased energy consumption (e.g., 62% for KAPAO [30] on our robot), shown as "Local" in Fig 2. Consequently, instead of performing computations locally, these ML applications often offload the computation of ML models to cloud data centers or edge devices equipped with powerful GPUs through the Internet of Things for these robots (robotic IoT).



(a) (b)

Fig. 1: A real-time people-tracking robotic application on our robot based on a well-known human pose estimation ML model, KAPAO [30].



(a) Average Inference Frequency with standard deviation. (b) Average power on our robot with standard deviation.

Fig. 2: Comparison between RRTO and the baselines on the application in Fig 1. The higher frequency indicates superior performance. "Local" refers to inference executed using an integrated computing accelerator on the robot, while "MCOP" [43] is a state-of-the-art (SOTA) non-transparent offloading system, and "Cricket" [10] is a SOTA transparent offloading system.

Existing computation offloading systems can be categorized into two types: transparent and non-transparent methods, depending on whether source code modification is required to enable offloading. **Non-transparent** offloading systems, such as MCOP [43], have demonstrated success across various ML models. MCOP adaptively schedules the computation of ML models to be offloaded to cloud servers based on the

model's workload, network bandwidth, and the computing power of cloud servers. This approach successfully improved the performance of our real-world robotic application by 5.2x and saved 54% power on our robot in our experiments, shown as "MCOP" in Fig 2. Such untransparency provides them with a simple and efficient scheduling method but demands significant coding effort to modify the source code for each application and can not be used on closed-source applications.

Transparent offloading methods, such as Cricket [10], provide a more convenient, but less efficient approach to offload computation to the cloud servers. ML model inference consists of a series of operators (e.g., addition, convolution). Transparent offloading methods intercept the call of each operator to the corresponding system functions (e.g., *torch.add()*,*torch.convolution()* for PyTorch [37]) and offload all these calls to the cloud servers through Remote Procedure Call (RPC). In this way, they avoid modifications to the source code by intercepting at the system layer but have to offload the RPC calls to the cloud servers one by one and add one Round-Trip Time (RTT) of RPC to the completion time of each operator, bringing extra communication cost.

Moreover, such inherent communication cost caused by the transparent offloading mechanism becomes substantial in robotic IoT networks. ML models commonly have hundreds of operators (e.g., 418 for KAPAO [30]), and each operator usually requires several RPC functions to complete based on different ML frameworks (e.g., cudaGetDevice, cudaLaunchKernel for PyTorch [30]), resulting in hundreds or thousands of RPC calls for a single inference (e.g., 4756 in our experiments). In robotic IoT networks, which typically rely on on wireless communication for robots, each RPC RTT usually takes a few milliseconds [39] (average 2.6 milliseconds in our experiments). Consequently, transparent offloading in robotic IoT networks, shown as "Cricket" in Fig 2, leads to prolonged communication time (98% inference time in our experiments) and may even increase the final energy consumption per inference, despite reducing power on the robot by offloading.

The key reason of this problem is that existing transparent offloading methods suffer from significant communication costs because they only invoke RPCs when an operator is utilized during the inference process, rather than anticipating their use in advance. This is because these methods are designed for general applications to leverage remote GPUs, rather than specifically designed for ML model inference. Such communication costs are unavoidable in general applications, as they cannot predict future operators from the upper-layer applications and thus cannot call RPCs in advance. Fortunately, in the context of ML model inference, we observe that the operators invoked by ML models typically follow a fixed order and can be predicted in advance, allowing for a more efficient inference process.

Based on this observation, we present **RRTO**, a **T**ransparent **O**ffloading system for model inference on robotic IoT with a novel **R**ecord/**R**eplay mechanism: record the order of operators invoked by ML model automatically and replay the execution of these fixed-order operators during model inference. In this way, RRTO correctly calls all involved operators in the recorded order of each inference of the ML model via one RPC, without waiting for the operator to be called via RPCs during inference. Based on this mechanism, RRTO dramatically reduces the inherent communication cost caused by traditional transparent offloading mechanism and achieves nearly the same communication cost as non-transparent offloading methods, allowing the outstanding scheduling algorithms in non-transparent offloading methods to be utilized in transparent offloading.

However, it is non-trivial to identify the specific operators invoked during each inference, as from the system layer only the continuous sequence of operator calls is visible to RRTO. Without the hints from the upper-layer applications, RRTO must explicitly identify the start and ending operators and operators between of an inference from the sequence of operators, so that RRTO can correctly intercept the starting of inference and call all involved operators of the inference.

To address this challenge, RRTO proposes a novel algorithm called *Data Dependency Search* to identify which operators are invoked for each inference. This algorithm first constructs a relationship graph based on the data dependencies between operators (i.e., the calculation result of the previous operator serves as the input for the next operator). It then takes operators that do not depend on any other operators as starting points and operators that no other operators have any dependencies on as ending points. The algorithm subsequently searches for the longest covering operator sequence between starting and ending operators and verifies whether such an operator sequence can constitute a complete model inference process (i.e., the entire log records of operators can be covered by repeating this sequence). Another specific observation that the final computation result needs to be sent back from GPU to CPU aids this algorithm in identifying the correct ending operator and dramatically reduces its search space.

We implemented RRTO on the Cricket's codebase [10] and also incorporated the outstanding scheduling algorithms of MCOP [43] into RRTO. We evaluated RRTO on a real-world Jetson Xavier NX [34] robot capable of GPU accelerated computation with a robotic application that tracks people in real time [30]. We compared RRTO with local computation, a SOTA non-transparent offloading method (MCOP [43]) and a SOTA transparent offloading method (Cricket [10]) when offload computation to different GPU devices (namely edge devices with high bandwidth and cloud servers with limited bandwidth). Evaluation shows that:

- RRTO is fast. It reduced inference time by 80% $\sim$ 98% compared to other baselines, similar to the reduction achieved by the SOTA non-transparent offloading method (MCOP).
- RRTO is energy-efficient. It reduced 91% $\sim$ 98% energy consumption per inference compared to other baselines, similar to the performance of MCOP.
- RRTO is robust in various robotic IoT environments. When the robotic IoT environment (the bandwidth to the GPU devices and the computing power of the GPU de-

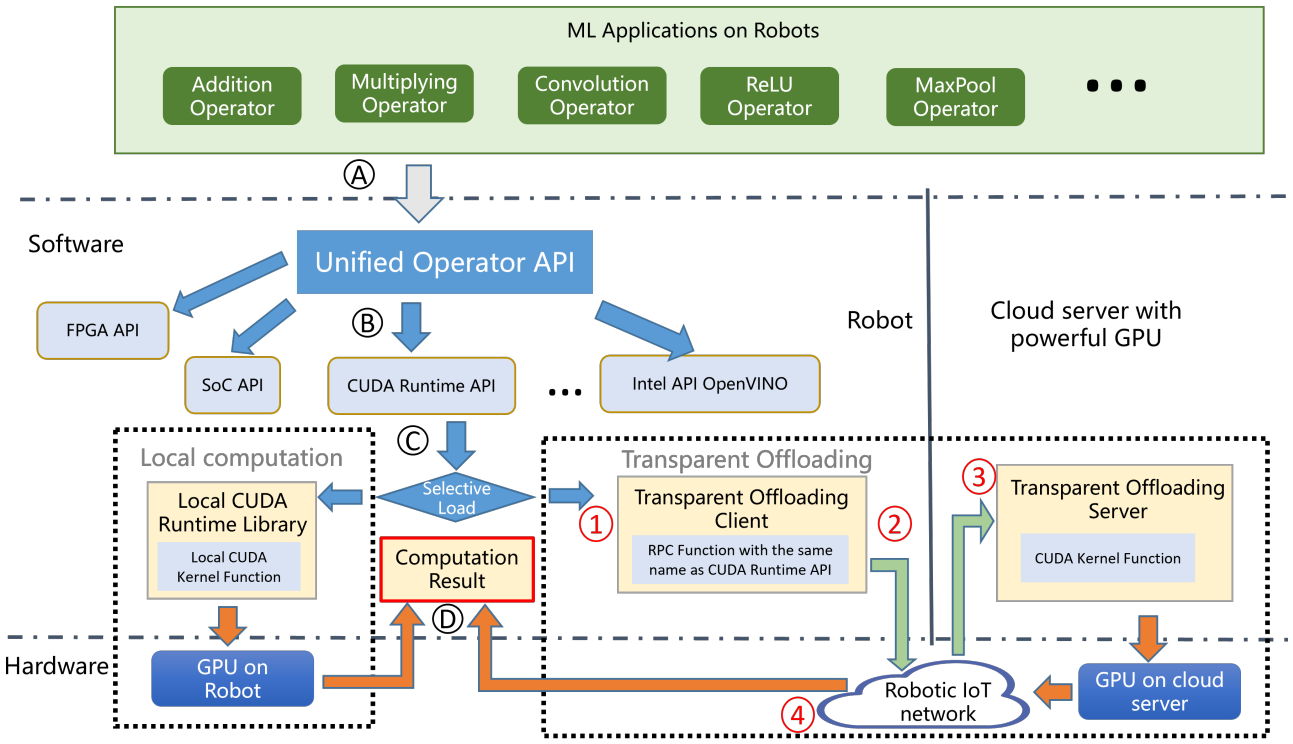


Fig. 3: Workflow of Transparent Offloading System for Model Inference on Robotic IoT

vices) changed, RRTO's superior performance remained consistent as MCOP.

Our main contribution is RRTO, the first high-performance transparent offloading system designed for model inference on robotic IoT. RRTO dramatically reduces the inherent communication cost caused by the traditional transparent offloading mechanism via the novel record/replay mechanism, achieving the same high performance as the SOTA non-transparent offloading method without modifying any source code. We envision that RRTO will foster the deployment of diverse robotic tasks on real-world robots in the field by providing fast, energy-efficient, and easy-to-use inference capabilities. RRTO's code is released on https://github.com/rss24p23/RRTO

In the rest of this paper, we introduce the background of this paper in Sec. II, give an overview of RRTO in Sec. III, present the detailed design of RRTO in Sec. IV, evaluate RRTO in Sec. VI, and finally conclude in Sec. VII.

II. Background

A. Workflow of Transparent offloading

When a robot performs GPU computations locally, the system call flow of the entire application can be depicted as the left part in Figure 3:

A The robot application completes the entire computation process for each service by sequentially calling different operators.

B Based on the application's running device (GPU), each operator passes through a unified operator API to find the local CUDA runtime library (NVIDIA GPUs provide high-performance parallel computing capabilities to applications using the CUDA runtime library [32]).

C The local CUDA runtime API is loaded by default.

D The robot's local CUDA library launches the corresponding CUDA kernel functions on the robot's GPU and returns the computation results to the upper-layer application.

Transparent offloading methods [10, 17, 28] usually takes the approach of rewriting dynamic link libraries, defining functions with the same name and using the *LD_PRELOAD* environment variable to prioritize loading the custom dynamic link library. The dynamic linker will then parse the original library function as the custom library function, thereby achieving library function interception. Subsequently, by modifying the management of GPU memory and the launch of CUDA kernel functions, the computation-related data and specific parameters of the corresponding kernel functions are sent to the remote server via RPC, realizing GPU computing transparent offloading. The primary modification occurs in step C. Similar to completing computations locally using the robot's GPU, after steps A and B, each operator's call to the corresponding kernel functions is intercepted by the functions with same name in the dynamic link library and offloaded to the cloud server to execute. The detailed steps are as follows

(depicted in the right part in Figure 3:

(1) By modifying the dynamic link library, each operator prioritizes calling the RPC functions with the same name as the CUDA runtime API in transparent offloading client, thereby identifying and intercepting all CUDA kernel function calls.
(2) The transparent offloading client transmits the called CUDA runtime API and required parameters to the cloud server through the robotic IoT network via RPC.
(3) The transparent offloading server launches the corresponding CUDA kernel functions on the cloud GPU and completes the respective computation.
(4) The transparent offloading server sends the computation results back to the client and the transparent offloading system returns the results to the upper-layer application.

B. Inconvenience of Non-transparent offloading

explain non-transparent offloading workflow (the optimizations performed in non-transparent offloading are mainly layer partition, which means placing parts of models on robots and parts on gpu servers at the granularity of layers.) and As described in Sec.II-B, the optimizations performed in non-transparent offloading are mainly layer partition, which means placing parts of models on robots and parts on gpu servers at the granularity of layers. Based on the fact that the amounts of output data in some intermediate layers of a DNN model are significantly smaller than that of its raw input data [], layer partitioning methods need to he structure of the model first (e.g. The computation time on each layer's robot and gpu and the transfer time to pass intermediate results of this layer to the gpuserver) and constitute various trade-offs between computation and transmission, taking into account application-specific inference speed requirements and energy consumption demands. As described in Sec.III, A layer of the model usually corresponds to several operators, which means that a layer produces multiple calls to rpc functions. However, a better layer partition methods to achieve fast and energy-efficient inference is beyond the scope of this paper. require much coding efforts

C. Related Work

Model Compression. Quantization and model distillation are the two most commonly used methods of ML model compression on the robots. Quantization [8, 16, 18] is a technique that reduces the numerical precision of model weights and activations, thereby minimizing the memory footprint and computational requirements of deep learning models. This process typically involves converting high-precision (e.g., 32-bit) floating-point values to lower-precision (e.g., 8-bit) floating-point representations, with minimal loss of model accuracy. Model distillation [19, 27, 42], on the other hand, is an approach that involves training a smaller, more efficient "student" model to mimic the behavior of a larger, more accurate "teacher" model by minimizing the difference between the student model's output and the teacher model's output. The distilled student model retains much of the teacher model's accuracy while requiring significantly fewer resources. These model compression methods are orthogonal to offloading methods, because they achieve faster inference speed by modifying the model and sacrificing the accuracy of the result, while offloading realizes fast inference without loss of accuracy by scheduling the calculation tasks.

RPC Optimization. RPC [9] is a communication protocol that enables one process to request a service from another process located on a remote computer, typically over a network. To improve RPC performance, several optimization strategies can be employed to achieve more efficient communication between remote processes and an overall enhancement in system performance: Batching [23] (aggregate multiple RPC calls into a single request), Asynchronous RPC [13] (decouple the request and response processing), and Caching [40] (Store the results of previous RPC calls). However, these optimization strategies are not effective in reducing the communication cost during model inference in robotic IoT. During the model inference process, the next operator is typically called after the previous operator has completed its execution, which renders Batching ineffective. While Asynchronous RPC and Caching enable the client to continue executing other tasks (subsequent operators) without waiting for the server's response, they lack the ability to determine when to stop and obtain the correct computation results. Compared with the traditional RPC optimization strategies, RRTO further reduces the communication cost by avoiding most operator's corresponding RPC communication, which will be described with more details in Sec. III and can be considered as a specific co-design for RPC optimization strategy and transparent offloading system for model inference.

Multiple Inference Scheduling. Significant research efforts have been devoted to accelerating multiple DNN inference via scheduling, in which how multiple inference tasks should be performed on various devices at different network bandwidths, taking into account application-specific inference speed requirements and energy consumption demands. For instance, [2, 11] support online scheduling of offloading inference tasks based on the current network and resource status of mobile systems while meeting user-defined energy constraints. [14] focused on optimizing DNN inference workloads in cloud computing using a deep reinforcement learning based scheduler for QoS-aware scheduling of heterogeneous servers, aiming to maximize inference accuracy and minimize response delay. While these methods focus on overall optimization in multi-task scenarios involving multi-robots, they are orthogonal to transparent offloading or non-transparent offloading for a single inference, where improved performance of inference of a single DNN model on a single robot can provide faster and more energy-efficient inference for these scenarios. In this paper, we only focus on optimization of single inference task. todo: same model: batching, different models

III. Overview

A. ML models with fixed-order operations

provide groundings (e.g., stats) to support that ML models typically perform operations in a fixed order.

In machine learning, many models in machine learning have fixed activated parts during the inference process. These include feedforward neural networks (e.g., Multi-Layer Perceptrons, Convolutional Neural Networks) with a fixed structure where neurons in each layer are activated sequentially, regardless of the input; Recurrent Neural Networks (e.g., simple RNNs, Elman networks) that have a fixed computation process at each time step, despite their ability to handle variable-length sequences; Autoencoders (e.g., basic autoencoders, Variational Autoencoders) with fixed encoder and decoder parts that activate the same components during each inference; Generative Adversarial Networks (e.g., basic GANs, DCGANs) with fixed generator and discriminator structures; and shallow machine learning models (e.g., linear regression, logistic regression, Support Vector Machines) that typically have a single, fixed layer. These models with fixed structures and no dynamic mechanisms have relatively simple and regular computation processes, making them easier to deploy and optimize. However, their expressiveness and flexibility may be limited compared to models with dynamic properties, and the choice of model type should depend on the requirements of the specific task at hand.

fix order operators

In contrast to models with fixed-order operations, several types of models may activate different parts during each inference, depending on the input. These include models with attention mechanisms (e.g., Transformers, BERT) that dynamically compute attention weights to focus on different parts of the input; gated models (e.g., LSTM, GRU) that control information flow through gating units, leading to different activated parts based on the input; conditional computation models (e.g., Mixture of Experts) that select different experts to activate based on input conditions; tree-structured models (e.g., decision trees, random forests) that traverse different paths of the tree structure according to input feature values; and dynamic network models (e.g., those obtained through Neural Architecture Search) that can dynamically adjust their structure based on the input. While this dynamic nature allows models to better adapt to different inputs and enhances their expressiveness and flexibility, it also poses challenges for model deployment and acceleration, requiring specialized optimization techniques.

B. Workflow of RRTO

Figure 4 illustrates the workflow of RRTO and contrasts it with traditional transparent offloading systems during model inference in robotic IoT networks. Traditional transparent offloading systems suffer from frequent RPC communication of operators, resulting in substantial communication cost and diminished system performance, including reduced GPU utilization on cloud servers, extended model inference time, and increased energy consumption per inference. The RTT communication cost for each operator is relatively minimal in

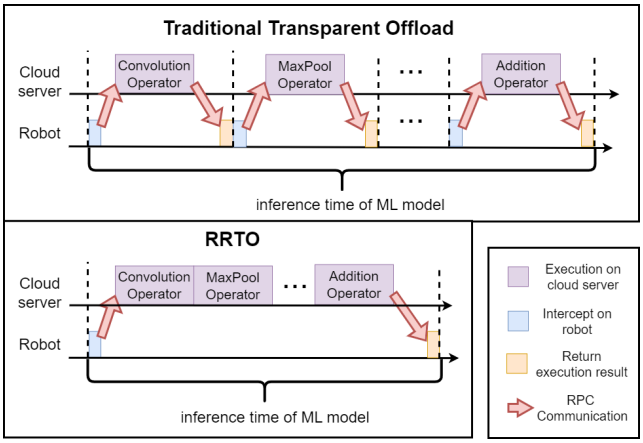


Fig. 4: Workflow of RRTO

data center networks, where devices are connected with high-speed (e.g., 40 Gbps ˜500 Gbps) networking technologies, such as InfiniBand [33] or PCIe [24]. However, in robotic IoT networks, the bandwidth between robots and powerful GPU devices is more limited and ML models commonly have hundreds of operators (e.g., 418 for [30]). For edge devices connected via Wi-Fi 6, the actual bandwidth reaches only 450 Mbps, leading to an average RTT of 2.6 milliseconds for each operator and the communication cost accounts for 98% of the total inference time in our experiments. For cloud servers, the even lower bandwidth imposed by the internet further hinders performance, as evaluated in Sec. VI.

To address the communication cost issue in transparent offloading processes of ML models, RRTO introduces an automatic recording and replay mechanism. Since ML model inference can be regarded as a complex function calculation, ML models typically invoke corresponding operators (e.g., addition, convolution, max-pool) in a fixed order to obtain accurate computation results and repeat these operators for each subsequent inference process. RRTO records the operators called during the first few inferences and replays the execution of this recorded sequence, referred to as the *inference operator sequence*, for subsequent inferences.

By employing this approach, RRTO only requires the first and last operators in the inference operator sequence to be offloaded via RPC, as in traditional transparent offloading systems, to obtain the correct input and output of ML models. For the operators in the middle of the inference operator sequence, RRTO directly calls these operators on the offloading server side without requiring any extra RPC communication from the offloading client side, thus avoiding the inherent communication cost associated with these operators.

Notice that, although there has been substantial work on optimizing RPC communication [23, 13, 40], RRTO goes a step further by directly eliminating the RPC communication of operators in the middle of the sequence. While existing RPC optimization methods still wait for RPCs from the offloading client to instruct the offloading server on the subsequent functions to be executed, RRTO preemptively calls the cor-

responding operators' functions on the offloading server side.

IV. Design

A. Architecture of RRTO

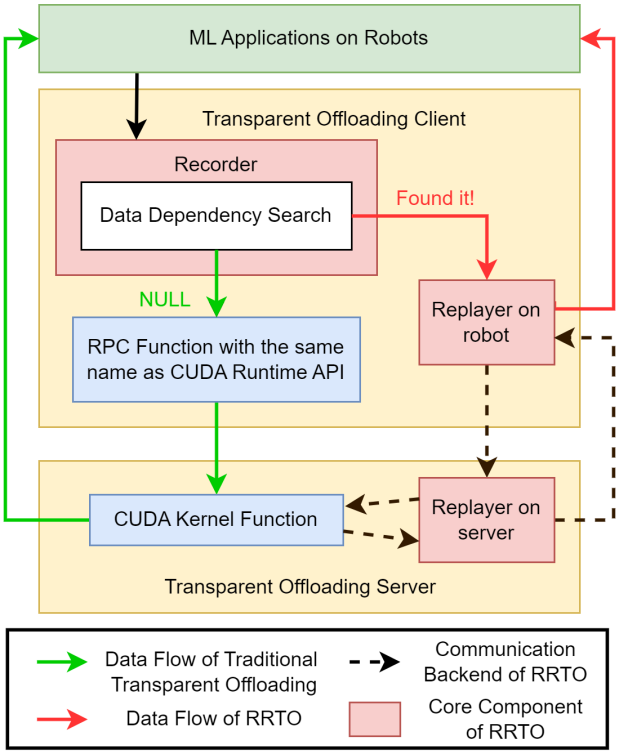


Fig. 5: Architecture of RRTO

Figure 5 presents the architecture of RRTO. In comparison with Figure 3, it is evident that RRTO implements its record/replay mechanism based on the core components of existing transparent offloading systems, and retains transparency to upper-layer applications, meaning that RRTO still does not require modifications to the source code to enable offloading.

Upon identifying and intercepting CUDA kernel function calls of operators from upper-layer ML applications, RRTO first records the function called by the operator, including the required parameters and the return value, using its recorder. Next, RRTO attempts to find the inference operator sequence through data dependency search, which will be described with more details in Sec. IV-C. If the recorder cannot find the inference operator sequence during the first few inferences, RRTO follows the same execution pattern as traditional transparent offloading systems by offloading the execution of the operator to the cloud server via RPC, as depicted by the green lines in Figure 5.

Once the recorder identifies the inference operator sequence, RRTO initiates the replaying of the execution of the inference operator sequence for subsequent inferences using the replayer on both the robot and server, as illustrated by the red lines in Figure 5. Similar to Caching (described in Sec. II-B) in existing RPC optimization methods, the replayer on the robot returns the execution results of previous RPC calls to the upper-layer applications, allowing the offloading client to continue execution until it is blocked at the ending operator to receive the final computation result from the offloading server. Meanwhile, the offloading server replays the execution of the inference operator sequence sent by the client and returns the final computation result to the offloading client. In this manner, RRTO achieves a communication cost nearly equivalent to that of non-transparent offloading methods, as it needs to transmit almost the same input and output as these methods.

Notice that, the scheduling approach discussed above offloads all computation of model inference to the cloud server. Existing computation offloading systems, such as MCOP [43], adopt more varied and flexible scheduling approaches that adapt to the dynamic environments. Depending on the model's workload, network bandwidth, and the computing power of the cloud server, these systems adaptively allocate a portion of the computation locally (compute on robots) and a portion to the cloud server. This enables faster inference by transmitting intermediate computation results with smaller transmission volumes rather than the raw input when the available network bandwidth decreases. Some works [6, 45, 31] have gone further by trading off between energy consumption and inference time. RRTO has the potential to provide these scheduling approaches with a lower-level and finer-granularity scheduling interface while maintaining nearly the same communication cost. However, adaptation to the dynamic environment is beyond the scope of traditional transparent offloading and the focus of this article. So we consider it as future work and implement the same scheduling approach in RRTO as in MCOP by adaptively choosing which operators to compute locally and which to offload to the cloud server within RRTO's recorder.

RRTO on Model with dynamic operations

RRTO when fail

B. Record/Replay Mechanism

Here we present how RRTO achieves the record/replay mechanism. The transparent offloading client part is given in Algo. 1 and the server part is given in Algo. 2. Details of data dependency search are mainly described in the next subsection IV-C.

In Algo. 1, on the offloading client side, RRTO takes as input a CUDA kernel function called by the corresponding operator and the required parameters. The algorithm first checks whether the recorder has already identified the inference operator sequence (line 1). If the sequence has not been found, RRTO proceeds with the recorder phase, which includes sending an RPC to the server (line 2), performing a data dependency search (line 3), and obtaining and recording the RPC execution result (lines 4, 5). To enhance system efficiency, RRTO overlaps the data dependency search with the execution of the RPC, allowing the DataDependencySearch algorithm calculation to be completed while the client awaits

Algorithm 1: RRTO_on_Client

Input: Cuda kernel function called by the corresponding operator $func$ and the required parameters $args$
Output: The execution result ret
Data: inference operator sequence $IOS = \emptyset$

1 **if** $IOS.empty()$ **then**
2 // recorder
3 $SendRPCtoServer(func, args)$
4 $IOS = DataDependencySearch(func, args)$
5 $ret = GetRPCExecutionResult()$
6 $RecordReturn(ret)$
7 **end**
8 **else**
9 // replayer on robot
10 **if** $func == IOS.start()["func"]$ **then**
11 $ret = StartRRTO(args, IOS)$
12 // start a new inference
13 **end**
14 **else if** $func == IOS.end()["func"]$ **then**
15 $ret = WaitingForRRTO()$
16 // Waiting for the final computation result
17 **end**
18 **else**
19 $ret = IOS.find(func)["ret"]$
20 **end**
21 **end**
22 **return** ret

Algorithm 2: RRTO_on_Server

Input: client task $task$
Data: inference operator sequence $IOS = \emptyset$, the execution result ret

1 **if** $task == SendRPCtoServer$ **then**
2 $func, args = GetClientInput()$
3 $ret = CUDARuntimeLibrary(func, args)$
4 **end**
5 **else**
6 // replayer on server
7 $args, IOS = GetClientInput()$
8 **foreach** $Op \in IOS$ **do**
9 $args = RRTOFixArgs(Op["args"], ret, args)$
10 $ret = CUDARuntimeLibrary(Op["func"], args)$
11 **end**
12 **end**
13 $SendExecutionResultBack(ret)$

the RPC execution result. If the inference operator sequence has been identified, RRTO proceeds with the replayer phase on the robot. This phase involves initiating RRTO for a new inference at the first operator (line 9), returning the execution results of previous RPC calls at the intermediate operators within the inference operator sequence (line 15), and waiting for the final computation result at the last operator (line 12).

In Algo. 2, the RRTO offloading server continuously awaits tasks from the client and returns the final execution results. If the client is still in the recorder phase, the RRTO offloading server processes RPC requests in the same manner as traditional transparent offloading systems (lines 2, 3). Once the client enters the replayer phase on the robot, the RRTO offloading server correspondingly transitions into the replayer phase on the server, replaying the execution of the inference operator sequence identified by the client (lines 8, 9). During this process, RRTO needs to adjust the parameters required by the corresponding operators, which typically consist of data or addresses of the computation results from the previous operators within the current inference.

C. Algorithm of Data Dependency Search

The performance of RRTO is heavily dependent on its ability to identify the correct inference operator sequence. If the sequence is not found accurately, even with a discrepancy of just one operator more or less, RRTO will not be able to obtain the correct inference result. Identifying the inference operator sequence is challenging, as RRTO must maintain its transparency and cannot receive any hints from upper-layer applications regarding which operators are invoked for each inference. Instead, it can only rely on log records of operators for the first few inferences to identify the starting and ending operators and operators between the two.

Algorithm 3: DataDependencySearch

Input: Cuda kernel function called by the last operator $func$ and the required parameters $args$
Output: inference operator sequence IOS
Data: Log records $history = \emptyset$ and relationship graph $map = \emptyset$

1 $history.add(func)$
2 $map.update(func, args)$
3 $StartPoses = map.startposes()$
4 $EndPoses = map.endposes()$
5 $Sequence = FindLongestPair(StartPoses, EndPoses)$
6 **if** $Verify(history, Sequence)$ **then**
7 **return** $Sequence$
8 **end**
9 **else**
10 **return** $NULL$
11 // cannot find
12 **end**

The pseudo code for data dependency search is provided in Algo. 3. To identifying the inference operator sequence, RRTO first records the data dependencies between operators (i.e., the calculation result of the previous operator serves as input for the next operator) and constructs a relationship graph (line 2). These dependencies can be established by comparing whether

parameters and calculated results between operators are the same (i.e., having the same address). Then, RRTO attempts to find the inference operator sequence based on this relationship graph.

RRTO considers operators that do not depend on any other operators as starting operators (line 3), which are candidates for the first operator in the inference operator sequence. It also considers operators that no other operators depend on as ending operators (line 4), which are candidates for the last operator in the inference operator sequence. RRTO searches for the longest covering operator sequence between starting and ending operators (line 5). Finally, RRTO verifies whether such an operator sequence can constitute a complete model inference process (line 6) by checking if the entire log records of operators can be covered by repeating this sequence.

During our implementation, we discovered that the final computation result needs to be sent back from GPU to CPU, and the ending operator can be identified by comparing the address of the final computation result. This specific observation aids RRTO in identifying the correct ending operator and dramatically reduces its search space. However, this does not entirely eliminate the need for establishing the relationship graph, because when the model has multiple outputs, an inference will involve multiple data copies from GPU to CPU and the Data Dependency Search algorithm is still needed to find the correct ending operator.

integrating Algorithm 3 with a pictorial representation of the relationship graph defined in lines 2 to 10, differentiating (maybe using different colours) operator sequences that constitute a complete model inference process from those that do not.

V. Implementation

We implemented RRTO within Cricket's codebase [10], a transparent offloading system that provides a virtualization layer for CUDA applications, enabling remote execution without the need for recompiling applications. RRTO employs the same Remote Procedure Call (RPC) for communication operations as Cricket: Libtirpc [9], a transport-independent RPC library for Linux. RRTO integrates its recorder and replayer into the corresponding RPC functions in Cricket and adapts MCOP's scheduling approach [43] by refining the scheduling granularity from MCOP's layers of ML models to the more fine-grained operators.

VI. Evaluation

Testbed. The evaluation was conducted on a customed four-wheeled robot (Fig 6), equipped with a Jetson Xavier NX [34] 8G onboard computer serving as the ROS master. The system runs Ubuntu 18.04 and utilizes a SanDisk 256G memory card, with ROS2 Galactic installed for application development and a dual-band USB network card (MediaTek MT76x2U) for wireless connectivity. The Jetson Xavier NX interfaces with a Leishen N10P LiDAR, ORBBEC Astra depth camera, and an STM32F407VET6 controller via USB serial ports. Both LiDAR and depth cameras facilitate environmental perception, enabling autonomous navigation, obstacle avoidance, and SLAM mapping. The host computer processes environmental information in ROS2 Galactic, performing path planning, navigation, and obstacle avoidance before transmitting velocity and control data to corresponding ROS topics. The controller then subscribes to these topics, executing robot control tasks.

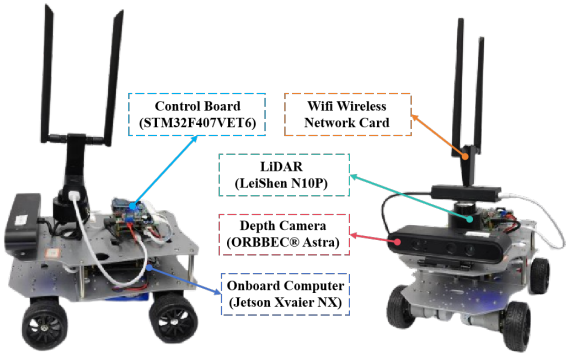


Fig. 6: The detailed composition of the robot platform

	inference	communication	standby
Power (W)	13.35	4.25	4.04

TABLE I: Power (Watt) of our robot in different states.

We documented the overall on-board energy consumption (excluding motor energy consumption for robot movement) of the robot in various states, as presented in Table I. These states include: inference, which refers to model inference with the full utilization of GPU and encompasses the energy consumption of both the CPU and GPU; communication, which involves communication with the server and includes the energy consumption of the wireless network card; and standby, during which the robot has no tasks to execute.

We evaluated two prevalent offloading scenarios for ML applications on robots, referred to as edge and cloud scenarios. In the edge scenario, computation is offloaded to an edge device, which is a PC equipped with 8xIntel(R) Core(TM) i7-7700K CPU @ 4.20GHz and NVIDIA GeForce GTX 1080 Ti 11GB, connected to our robot via Wi-Fi 6 with an average bandwidth of 450 Mbps over 160MHz channel at 5GHz frequency in our experiments. In the cloud scenario, computation is offloaded to a cloud server, which is a GPU server equipped with 48xIntel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz and 4xNVIDIA GeForce RTX 2080 Ti 11GB, connected to our robot via the Internet with an average bandwidth of 160 Mbps in our experiments.

Real-World Robotic Application. We evaluated a real-time people-tracking robotic application on our robot as depicted in Fig 7. To achieve seamless tracking of individuals, a minimum detection frequency of 1 Hz for the target person is required, illustrated by the black dotted line in Fig 2a and Fig 9a. The detailed workflow is described as follows: The ORBBEC Astra

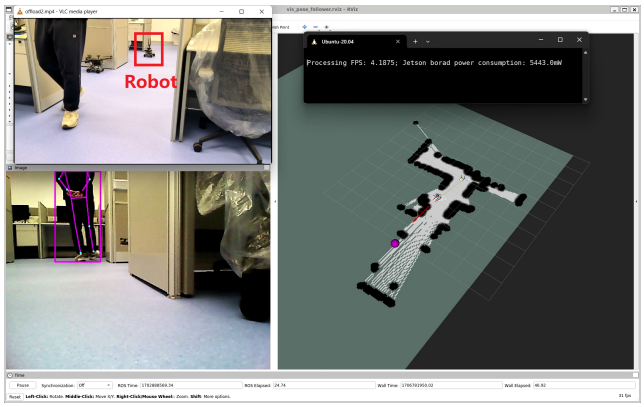
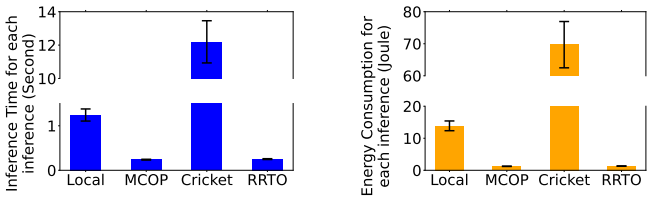


Fig. 7: A screenshot of our real-world experiment. The upper right corner displays real-time FPS and on-board energy consumption, the lower right corner shows the map created by the robot using its LiDAR, the lower left corner features the real-time view from the robot's camera, and the upper left corner provides a third-angle observation of the entire experimental process.



(a) Average Inference Time with standard deviation. (b) Average Energy Consumption with standard deviation.

Fig. 8: Comparison between RRTO and the baselines per inference in the edge scenario.

depth camera on our robot generates both RGB images and corresponding depth images. First, we obtain a person's key points in the RGB image using a well-known human pose estimation model, KAPAO [30]. Then, by utilizing the depth values corresponding to these key points in the depth image, the points are mapped to a three-dimensional map constructed by the robot's LiDAR. A Kalman filter is applied to filter out noise and obtain a more accurate position of the person. Finally, the STM32F407VET6 controller directs the robot to the target position, enabling real-time tracking of the person.

Baselines. We compared RRTO with local computation, MCOP [43] (a SOTA non-transparent offloading system) and Cricket [10] (a SOTA transparent offloading system) when offloading computation to different GPU devices (edge devices with high bandwidth and cloud servers with limited bandwidth).

The evaluation questions are as follows:

- RQ1: How does RRTO benefit real-world robotic applications compared to baseline systems in terms of inference time and energy consumption?
- RQ2: How does RRTO's record/replay mechanism work?
- RQ3: How sensitive is RRTO to various robotic IoT environments (the bandwidth to the GPU devices and the computing power of the GPU devices)?
- RQ4: What are the limitations and potentials of RRTO?

A. End-to-End Performance

Our evaluation results for the edge scenario, as presented in Fig 2, demonstrate that RRTO provides performance comparable to MCOP for our robotic application. We further compared RRTO with baseline methods in terms of inference time and energy consumption per inference, as illustrated in Fig 8.

In terms of inference time, RRTO achieved an 80% reduction compared to local computation and a 98% reduction compared to Cricket, as shown in Fig 8a, leading to a higher frequency in Fig 2a. Despite the additional data transfers required, the powerful GPU's shorter computation time allowed RRTO to perform inferences faster than local computation. The significant communication cost incurred by Cricket's transparent offloading mechanism dramatically slowed down its inference time, which will be discussed in more detail in Sec. VI-B. RRTO successfully reduced this extremely heavy communication cost using its record/replay mechanism, achieving a similar inference time to MCOP with nearly the same communication cost.

In terms of energy consumption, RRTO saved 90% energy compared to local computation and 98% energy compared to Cricket to finish the inference of one frame, as shown in Fig 8b. Although RRTO only saved 53% power consumption compared to local computation and 8% power compared to Cricket in Fig 2b, the shorter inference time in Fig 8a allowed RRTO to consume significantly less energy per inference.

Notice that, the average power values in Fig 2b are not equal to those in Table I. Our application cannot fully utilize the GPU of the Jetson Xavier NX, resulting in the average power of local computation being lower than during the inference stage. Additional CPU computing tasks, such as robot control, cause the average power of offloading methods to be higher than during the communication and standby stages. Furthermore, the extremely frequent RPC function calls of Cricket generated 8% more power consumption on the CPU, and the significant communication cost caused Cricket to spend excessive time in the communication stage, wasting 98% of energy for each inference.

B. Micro-Event Analysis

The working process of kapao [30] follows the default detection model in yolo v5 [21]: the inference pipeline is first initialized by generating a mesh grid of a certain size that fits the input image size, which serves as the storage of intermediates; then in the following loop iterations through the inference pipeline the mesh grid is reused and the operator call sequence is fixed.

To obtain a deeper understanding of the performance improvement achieved by RRTO, we performed an analysis of the RPC function calls generated by Cricket during a single inference to present the features of the traditional transparent

offloading mechanism. The detailed breakdown of these calls is presented in Table II.

CUDA Runtime API	Numbers
cudaGetDevice	3677 (80.35%)
cudaGetLastError	458 (10.00%)
cudaLaunchKernel	418 (9.13%)
cudaStreamSynchronize	6 (0.13%)
cudaMalloc	5 (0.11%)
cudaStreamIsCapturing	5 (0.11%)
cudaMemcpyHtoD	5 (0.11%)
cudaMemcpyDtoH	1 (0.02%)
cudaMemcpyDtoD	1 (0.02%)

TABLE II: Composition of RPC function calls during each inference of KAPAO

In Table II, we observed that a significant portion, specifically 90.35%, of the RPC function calls were attributed to cudaGetDevice and cudaGetLastError. These calls were generated by PyTorch [37] due to our application's reliance on this framework and served the purpose of determining the data's location, which are essential for executing computations across multiple GPUs and parallel tasks. Even when restricting PyTorch to utilize only a single GPU sequentially and employing Caching (described in Sec. II-B) to avoid invoking these RPC functions, transparent offloading systems can only achieve inference times similar to local computation accodring to our experiments. However, they still cannot match the inference times achieved by MCOP or RRTO. This is evident from the fact that cudaLaunchKernel still accounts for 9.13% of the total RPC function calls, as it informs the server about subsequent computing tasks, such as additional convolution or maxpool operations. While existing RPC optimization methods rely on waiting for RPCs of cudaLaunchKernel from the client to instruct the server to fullfill the subsequent computing tasks, RRTO records these cudaLaunchKernel function calls and directly executes the subsequent computing tasks on the server without the need for communication with the client.

Regarding the remaining RPC functions, namely cudaStreamSynchronize, cudaMalloc, cudaStreamIsCapturing, and cudaMemcpyDtoD, which collectively account for 0.37% of the total RPC calls, they primarily handle data transmission and synchronization within the GPU and can also be replayed by RRTO on the server. However, cudaMemcpyHtoD and cudaMemcpyDtoH, accounting for 0.13% of the total RPC calls, are primarily used for data transmission between the CPU and GPU, which are mainly employed for the input and output of the ML model and cannot be replayed by RRTO.

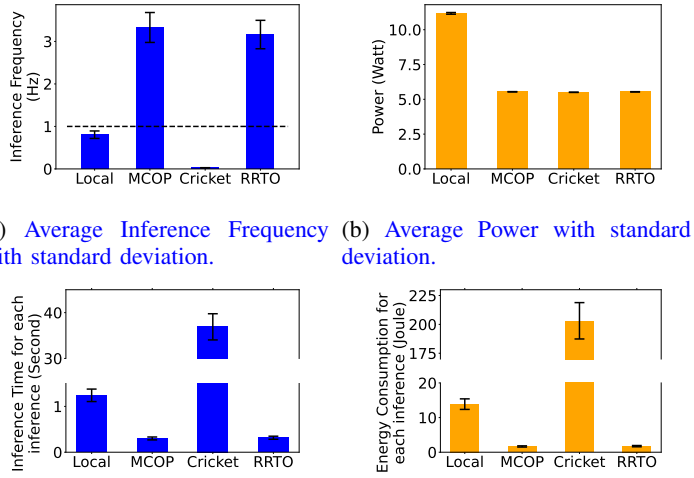
To provide an additional perspective on the performance gain achieved by RRTO, we compared it with the baselines using third-party explicit features, as shown in Table III. MCOP synchronizes data (input and output of the ML model via cudaMemcpyHtoD and cudaMemcpyDtoH in Cricket) at the application layer by modifying the source code instead of using RPC. Cricket incurred a higher communication cost,

	MCOP	Cricket	RRTO
RPCs for each inference	N\A	4576	6
Average GPU utilization on edge device	29.0%	1.1%	27.5%

TABLE III: Comparison between RRTO and the baselines on the third-party explicit features

leading to lower GPU utilization on the edge device. Although RRTO still needs to handle cudaMemcpyDtoH and cudaMemcpyHtoD as Cricket does, resulting in 6 RPCs per inference, the benefits of RRTO in terms of performance improvement are evident.

C. Sensitivity Studies



(a) Average Inference Frequency with standard deviation.
(b) Average Power with standard deviation.
(c) Average Inference Time with standard deviation.
(d) Average Energy consumption with standard deviation.

Fig. 9: Comparison between RRTO and the baselines in the cloud scenario.

We conducted an empirical evaluation of RRTO's performance across various robotic IoT environments, the cloud scenario as illustrated in Fig 9. While cloud servers are equipped more powerful GPUs, their limited bandwidth causes offloading methods to incur greater communication time, resulting in slower inference times. In the cloud scenario, Cricket experiences additional performance degradation due to the significant proportion of the communication cost caused by its transparent offloading mechanism. However, with a transmission cost similar to that of MCOP, RRTO still manages to achieve nearly same high performance as MCOP in the cloud scenario.

D. Lessons learned

Changing network bandwidth. During the implementation and evaluation of RRTO, we discovered that the performance of computation offloading heavily rely on the network bandwidth of robotic IoT networks. Unlike GPU clusters equipped

with fast interconnects such as InfiniBand [33], robotic IoT often lack fast and stable network connections for computation offloading. Many factors may cause network fluctuations, including wireless signal interference for Wi-Fi (due to obstacles [1], robot movement [15], channel preemption by other wireless devices [12], etc.) and network congestion for Internet connections (caused by multiple users accessing simultaneously). The fine-grained offloading schedules for operators in RRTO, as described in Sec. III-B, have the potential to handle such random and violent fluctuations in network bandwidth, and we leave it as future work.

ML models with dynamic operation sequences. First, as described in Sec.III, dynamic require more computation resource, and commonly located in the cloud, but not on robot. Secondly, even placing these dynamic model onto robots, the existing method still face the same challenge to perform with as RRTO does. For non-transparent offloading methods, they offload , For transparent offloading methods, We provide more detailed background knowledge about non-transparent offloading methods in Sec.II-B. In other words, perform with ML models with

Fixed calculation logic. The record/replay mechanism of RRTO is based on the fact that operators involved in inference of a DNN model are typically invoked in fixed order. On one hand, this mechanism allows RRTO to support other computation tasks with fixed calculation logic, even if they are not ML applications but rather computation-intensive tasks [5, 7]. On the other hand, RRTO cannot support computation tasks with unfixed calculation logic. In such tasks, the logic of each calculation differs, making it impossible for RRTO to replay them. However, computation tasks with complex logic and branching are better suited for execution in CPU rather than GPU [38] beyond the scope of RRTO.

Future work. We would like to apply and evaluate RRTO in a wider variety of real-world applications on robots in the future. Also, it is of interest to explore further improvements of RRTO such as building an edge computing power network that offloads computation to other idle robots or edge devices [3] through fine-grained job scheduling at the operator level to ensure full utilization of GPU resources. We believe that such investigation will enable even faster, more energy-efficient, and more robust inference capabilities for ML applications in real-world robotic IoT networks. for Multiple Inference

VII. Conclusion

In this paper, we introduce RRTO, a high-performance transparent offloading system designed for model inference on robotic IoT. RRTO tackles the inherent communication cost associated with traditional transparent offloading mechanisms by introducing a novel record/replay mechanism, achieving the same high performance as the SOTA non-transparent offloading method without modifying any source code. RRTO tackles the inherent communication cost associated with traditional transparent offloading mechanisms by introducing a novel record/replay mechanism, achieving the same high performance as the SOTA non-transparent offloading method without modifying any source code. We envision that RRTO will significantly streamline the deployment of a diverse array of ML applications on mobile robots in real-world settings. By offering fast and energy-efficient inference capabilities, RRTO enables these robots to execute complex tasks with heightened efficiency and effectiveness.

Appendix

The revised parts of our manuscript are highlighted in blue. We have comprehensively addressed the reviewers' valuable comments by enhancing the overall writing quality, including but not limited to rectifying typographical errors, updating figures in Sec.VI with standard deviations, and expanding the theoretical background in SecII and Sec.III. Additionally, we have provided point-by-point responses to each reviewer's comments, with corresponding hyperlinks directing to the revised sections for easy reference.

Q1. (Chair qHQW) The presented application scenario is quite simplistic and the theoretical background is quite poor.

A1: In Sec.II, we have included more comprehensive background information on non-transparent offloading methods and other related scenarios as suggested by the reviewers. Sec.III provides detailed observations about RRTO and the case study, supporting the notion that ML models typically perform operations in a fixed order. Furthermore, we have updated our evaluation based on the reviewers' comments, and we believe that our revised evaluation fully represents the scenarios described in the case study. Regarding other scenarios mentioned by the reviewers that fall outside the scope of this paper, we will address them individually in the following questions.

Q2. (Reviewer sZZS) This paper lacks scientific findings and groundings to support that ML models typically perform operations in a fixed order.

A2: We provide detailed observations about RRTO and the case study to support that ML models typically perform operations in a fixed order in Sec.III.

Q3. (Reviewer sZZS) How the existing method and RRTO can perform with ML models with dynamic operation sequences or fixed order operation with a different initial sequence?

A3: For ML models with dynamic operation sequences, both the existing method and RRTO face the same challenge to perform with, as described in Sec.VI-D, which means that ML models with dynamic operation sequences is still a open problem for all existing offloading methods, not just for RRTO and beyond the scope of this paper. For ML models with fixed order operations with a different initial sequence, note that RRTO records all involved operators during the first few inferences, not just during the initial process as described in Sec.IV. In the first several inferences, RRTO is the same as the traditional non-transparent method, the only difference is that the corresponding operators are also recorded, then when the correct operators sequence is found, RRTO activates the record/replay mechanism, as described in Sec.IV. In this

way, RRTO only requires that the order of the operators during inferences is fixed, independent of the operators in the initialization process. We also supplement with more complete analysis at different stages of model inference on system call in Sec.VI-B, including initial process of KAPAO[30], to help readers better understand the entire workflow of RRTO at system call level.

Q4. (Reviewer 6MCn) What would happen if the inference model triggers a new sequence of inference operations that start and end with similar GPU calls following the result of a previous inference?

A4: For a new inference triggered by a new inference model, the code base of RRTO, Cricket, has already provided the ability to distinguish RPC functions from different models, because it is designed for general applications to leverage remote GPUs, so by distinguishing RPC functions from different models, the calculation results in the GPU servers can be returned to the correct model. As for a new inference triggered by the same inference model, it is indeed difficult to distinguish which operators are the first inference and which are the second inference When the operator records of two inferences are mixed together. This is because at the level of system function calls, without the help of the upper application, RRTO cannot distinguish whether these operators that appear twice are called twice in the same inference process Or are called once in each of the two inferences. However, note that a inference process usually starts with 'cudaMemcpyHtoD' (put input data from cpu to the local or remote GPU) and ends with 'cudaMemcpyDtoH' (return the final computation result from GPU to CPU). By marking these two functions, RRTO can know how many inference tasks are running at the same time, so that, as long as the sequence of operators called by each of the two inference sequences is fixed, RRTO can know how many times the operator in the correct operator sequence of the single inference process will be repeated in the corresponding log record, so as to help RRTO propose redundant repeating operator records and find the correct operators sequence. Moreover, a new inference should be the handled by multiple inference scheduling methods first, as described in Sec.II-C, such as batching several inference tasks together, but not directly handled by the above method.

Q5. (Reviewer 6MCn) Could the optimizations performed in frameworks like MCOP be adapted to RRTO, as this would significantly improve the performance?

A5: Yes, the optimizations performed in MCOP, which are layer partition (placing parts of models on robots and parts on gpu servers at the granularity of layers) has been adapted to RRTO as described in Sec.II-B.

Q6. (Reviewer 6MCn) Would RRTO adapt to false predictions on the list of GPU operations to be performed?

A6: When the prediction fails, the record/replay mechanism of RRTO cannot find the correct sequence of operators in an inference process, so it cannot replay the inference process correctly. As described in Sec.IV, RRTO will restart a inference via the traditional non-transparent method when predictions false. However, false prediction happens only when the operator sequence of the model inference is dynamic, thus making RRTO degenerate into the traditional non-transparent method. Our case study in Sec.III shows that the operator sequence of model inference on robots are typically in fixed order, while the model with dynamic operations is usually deployed in data centers because of its larger parameter count, and how to handle ML models with dynamic operation sequences is still a open problem for all existing offloading methods, not just for RRTO, as described in Sec.VI-D and beyond the scope of this paper.

Q7. (Reviewer 6MCn) How about the scenario where robots using edge computing utilize multiple inference models simultaneously

A7: As described in Sec.II-C, the scheduling of multiple inference is required by all offloading methods because they are orthogonal to all offloading methods, which can provide faster and more energy-efficient inference for them. Although offloading some small models may lead to negative optimization compared with local computation due to the limited bandwidth and there are more and more end-to-end

Q8. (Reviewer 6MCn) What is the benefit of such an offloading when the images from the robot can be transmitted at a similar or higher rate to perform inference on an edge device?

A8: Similar to Question 7, are beyond the scope of this paper.

Q9. (Reviewer De3N) The case study of RRTO.

A9: we provide the case study of RRTO in Sec.III

Q10. (Reviewer De3N) Stressing the proposed framework to more complex robotic systems (like UAVs or legged robots) requiring more computationally demanding machine learning performances.

A10: The more computation-intensive models do need to be offloaded to GPU servers, and RRTO can provide them with a convenient and high-performance offloading deployment. However, complex robotic systems (like UAVs or legged robots) require additional hardware support and extra coding efforts to setup the corroding robotic systems. we leave it as a future work to evaluate RRTO on various robotic systems due to the limited time of rebuttal. we believe the high performance of RRTO will remain in various robotic systems.

Q11. (Reviewer De3N) Proposed writing comments to help readers deeply understand your work

A11: Thanks to the reviewer's suggestions, we have revised the corresponding part of the writing according to the reviewer's suggestions.

REFERENCES

[1] Badrul Alam, Giovanna Calò, Gaetano Bellanca, Jacopo Nanni, Ali Emre Kaplan, Marina Barbiroli, Franco Fuschini, Paolo Bassi, Jinous Shafiei Dehkordi, Velio Tralli, et al. Numerical and experimental analysis of on-chip optical wireless links in presence of obstacles. *IEEE Photonics Journal*, 13(1):1–11, 2020.

[2] Majid Altamimi, Atef Abdrabou, Kshirasagar Naik, and Amiya Nayak. Energy cost models of smartphones for

task offloading to the cloud. *IEEE Transactions on Emerging Topics in Computing*, 3(3):384–398, 2015.

[3] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online job scheduling in distributed machine learning clusters. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 495–503. IEEE, 2018.

[4] Anh-Quan Cao and Raoul de Charette. Monoscene: Monocular 3d semantic scene completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3991–4001, 2022.

[5] Niels Cautaerts, Phillip Crout, Håkon W Ånes, Eric Prestat, Jiwon Jeong, Gerhard Dehm, and Christian H Liebscher. Free, flexible and fast: Orientation mapping using the multi-core and gpu-accelerated template matching capabilities in the python-based open source 4d-stem analysis toolbox pyxem. *Ultramicroscopy*, 237:113517, 2022.

[6] Xing Chen, Jianshan Zhang, Bing Lin, Zheyi Chen, Katinka Wolter, and Geyong Min. Energy-efficient offloading for dnn-based smart iot systems in cloud-edge environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):683–697, 2021.

[7] Rohit Chowdhury and Deepak Subramani. Optimal path planning of autonomous marine vehicles in stochastic dynamic ocean flows using a gpu-accelerated algorithm. *IEEE Journal of Oceanic Engineering*, 47(4):864–879, 2022.

[8] Alexandre Défossez, Yossi Adi, and Gabriel Synnaeve. Differentiable model compression via pseudo quantization noise. *arXiv preprint arXiv:2104.09987*, 2021.

[9] Steve Dickson. libtirpc: Transport independent rpc library. https://git.linux-nfs.org/?p=steved/libtirpc.git, 2024. Git repository.

[10] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. Cricket: A virtualization layer for distributed execution of cuda applications with checkpoint/restart support. *Concurrency and Computation: Practice and Experience*, 34(14), 2022. doi: 10.1002/cpe.6474.

[11] Khalid Elgazzar, Patrick Martin, and Hossam S Hassanein. Cloud-assisted computation offloading to support mobile services. *IEEE Transactions on Cloud Computing*, 4(3):279–292, 2014.

[12] Khaled Eshteiwi, Georges Kaddoum, Bassant Selim, and François Gagnon. Impact of co-channel interference and vehicles as obstacles on full-duplex v2v cooperative wireless network. *IEEE Transactions on Vehicular Technology*, 69(7):7503–7517, 2020.

[13] Stijn Eyerman and Ibrahim Hur. Efficient asynchronous rpc calls for microservices: Deathstarbench study. *arXiv preprint arXiv:2209.13265*, 2022.

[14] Zhou Fang, Tong Yu, Ole J Mengshoel, and Rajesh K Gupta. Qos-aware scheduling of heterogeneous servers for inference in deep neural networks. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 2067–2070, 2017.

[15] Yange Feng, Enrico Benassi, Liqiang Zhang, Xiaojuan Li, Daoai Wang, Feng Zhou, and Weimin Liu. Concealed wireless warning sensor based on triboelectrification and human-plant interactive induction. *Research*, 2021.

[16] Ștefan Gheorghe and Mihai Ivanovici. Model-based weight quantization for convolutional neural network compression. In *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–4. IEEE, 2021.

[17] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31-September 3, 2010, Proceedings, Part I 16*, pages 379–391. Springer, 2010.

[18] Cheng Gong, Yao Chen, Ye Lu, Tao Li, Cong Hao, and Deming Chen. Vecq: Minimal loss dnn model compression with vectorized weight quantization. *IEEE Transactions on Computers*, 70(5):696–710, 2020.

[19] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.

[20] Vinayak Honkote, Dileep Kurian, Sriram Muthukumar, Dibyendu Ghosh, Satish Yada, Kartik Jain, Bradley Jackson, Ilya Klotchkov, Mallikarjuna Rao Nimmagadda, Shreela Dattawadkar, et al. 2.4 a distributed autonomous and collaborative multi-robot system featuring a low-power robot soc in 22nm cmos for integrated battery-powered minibots. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 48–50. IEEE, 2019.

[21] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, NanoCode012, Yonghye Kwon, Kalen Michael, TaoXie, Jiacong Fang, imyhxy, Lorna, (Zeng Yifu), Colin Wong, Abhiram V, Diego Montes, Zhiqiang Wang, Cristi Fati, Jebastin Nadar, Laughing, UnglvKitDe, Victor Sonck, tkianai, yxNONG, Piotr Skalski, Adam Hogan, Dhruv Nair, Max Strobel, and Mrinal Jain. ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation, November 2022. URL https://doi.org/10.5281/zenodo.7347926.

[22] K J Joseph, Salman Khan, Fahad Shahbaz Khan, and Vineeth N Balasubramanian. Towards open world object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5830–5840, June 2021.

[23] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 36–51, 2021.

[24] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating

modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[25] Qingbiao Li, Fernando Gama, Alejandro Ribeiro, and Amanda Prorok. Graph neural networks for decentralized multi-robot path planning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11785–11792. IEEE, 2020.

[26] Yiming Li, Zhiding Yu, Christopher Choy, Chaowei Xiao, Jose M Alvarez, Sanja Fidler, Chen Feng, and Anima Anandkumar. Voxformer: Sparse voxel transformer for camera-based 3d semantic scene completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9087–9098, 2023.

[27] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. Ensemble distillation for robust model fusion in federated learning. *Advances in Neural Information Processing Systems*, 33:2351–2363, 2020.

[28] Yu-Shiang Lin, Chun-Yuan Lin, Che-Rung Lee, and Yeh-Ching Chung. qcuda: Gpgpu virtualization for high bandwidth efficiency. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 95–102. IEEE, 2019.

[29] Shuai Liu, Xin Li, Huchuan Lu, and You He. Multi-object tracking meets moving uav. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8876–8885, June 2022.

[30] William McNally, Kanav Vats, Alexander Wong, and John McPhee. Rethinking keypoint representations: Modeling keypoints and poses as objects for multi-person human pose estimation. *arXiv preprint arXiv:2111.08557*, 2021.

[31] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. Distributed inference acceleration with adaptive dnn partitioning and offloading. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 854–863. IEEE, 2020.

[32] NVIDIA. cuda runtime apis. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html, 2024.

[33] NVIDIA. Infiniband networking solutions. https://www.nvidia.com/en-us/networking/products/infiniband/, 2024.

[34] NVIDIA. The world's smallest ai supercomputer. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[35] Takeshi Ohkawa, Kazushi Yamashina, Hitomi Kimura, Kanemitsu Ootsu, and Takashi Yokota. Fpga components for integrating fpgas into robot systems. *IEICE TRANSACTIONS on Information and Systems*, 101(2):363–375, 2018.

[36] Brian Plancher, Sabrina M. Neuman, Thomas Bourgeat, Scott Kuindersma, Srinivas Devadas, and Vijay Janapa Reddi. Accelerating robot dynamics gradients on a cpu, gpu, and fpga. *IEEE Robotics and Automation Letters*, 6 (2):2335–2342, 2021. doi: 10.1109/LRA.2021.3057845.

[37] pytorch. pytorch. https://www.pytorch.org, 2024.

[38] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Computing Surveys (CSUR)*, 55(1):1–38, 2022.

[39] Matthias Schneider, Fabian Haag, Abdul Kader Khalil, and David Albert Breunig. Evaluation of communication technologies for distributed industrial control systems: Concept and evaluation of 5g and wifi 6. *Procedia CIRP*, 107:588–593, 2022. ISSN 2212-8271. doi: https://doi.org/10.1016/j.procir.2022.05.030. URL https://www.sciencedirect.com/science/article/pii/S2212827122003146. Leading manufacturing systems transformation – Proceedings of the 55th CIRP Conference on Manufacturing Systems 2022.

[40] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. Cliquemap: Productionizing an rma-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 93–105, 2021.

[41] Binyu Wang, Zhe Liu, Qingbiao Li, and Amanda Prorok. Mobile robot path planning in dynamic environments through globally guided reinforcement learning. *IEEE Robotics and Automation Letters*, 5(4):6932–6939, 2020. doi: 10.1109/LRA.2020.3026638.

[42] Lin Wang and Kuk-Jin Yoon. Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks. *IEEE transactions on pattern analysis and machine intelligence*, 44(6):3048–3068, 2021.

[43] Huaming Wu, William J. Knottenbelt, and Katinka Wolter. An efficient application partitioning algorithm in mobile environments. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1464–1480, 2019. doi: 10.1109/TPDS.2019.2891695.

[44] Zhaoyang Xia, Youquan Liu, Xin Li, Xinge Zhu, Yuexin Ma, Yikang Li, Yuenan Hou, and Yu Qiao. Scpnet: Semantic scene completion on point cloud. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17642–17651, 2023.

[45] Min Xue, Huaming Wu, Guang Peng, and Katinka Wolter. Ddpqn: An efficient dnn offloading strategy in local-edge-cloud collaborative environments. *IEEE Transactions on Services Computing*, 15(2):640–655, 2021.

[46] Yang Yang, Li Juntao, and Peng Lingling. Multi-robot path planning based on a deep reinforcement learning dqn algorithm. *CAAI Transactions on Intelligence Technology*, 5(3):177–183, 2020.