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

Author Names Omitted for Anonymous Review. Paper-ID [add your ID here]

***Abstract*—Critical robotic tasks, such as rescue and disaster response, increasingly repy on Machine Learning (ML) models deployed on wireless robots, on which the extremely-heavy computation of ML models are offloaded to powerful GPU deviecs (e.g., cloud data centers) over Internet of Things of these robots (robotic IoT) to achieve fast and energy-efficient inferences. While offloading system have been widely successful across various ML models in untransparent ways (i.e., require modifications to the source code to enable such offloading schedules), they also pose a significant obstacle for deployment on robots: significant coding effort to modify the source code for each application and can not be used on closed-source applications. Unfortunately, existing transparent offloading methods can only offload the function call of each operator (e.g., *torch.add()*) within the ML models via Remote Procedure Call (RPC) to GPU devices, leading 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 process of calling the operators is fixed during model inference, RRTO automatically records the order of operators invoked by ML model and replays the execution of these fixed-order operators during model inference. In this way, RRTO calls the RPCs in advance to circumvent the significant communication costs caused by existing transparent offloading mechanisms and applies these successful scheduling algorithms from untransparent offloading to transparent offloading. Evaluations demonstrate that RRTO reduces inference time by xx% and saves xx% in energy consumption compared to other baselines without modifying any source code, achieving performance akin to the state-of-the-art untransparent offloading method.

I. Introduction

The rapid advancement of machine learning (ML) methods has achieved remarkable success in various robotic tasks, such as intelligent navigation[], environmental perception[], and human-robot interaction[]. Deploying these ML methods onto real-world robots typically necessitates additional hardware support due to the computationally-intensive nature of ML models (e.g., xx% computation of the whole robot task[]). However, directly integrating computing accelerators (e.g., GPU[], NPU[], FPGA[], SoC[]) onto real-world robots not only introduces additional expensive costs, but also leads to increased energy consumption (e.g., xx% for []). Consequently, these ML methods 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).

Computation offloading systems, such as MCOP[4], have demonstrated success across various ML models in **untransparent** ways, meaning that they require modifications to the source code to enable such offloading methods. 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 reduces xx%inference time and saves xx%energy consumption in our experiments. 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[2], provide a more convenient, but also worse performance approach to offload computation to the cloud servers. ML model inference consists of a series of operators (e.g., addition, convolution). Transparent offloading methods intercepts each operator's call to the corresponding system functions (e.g., *torch.add()*,*torch.convolution()*) and offloads all these calls to the cloud servers through Remote Procedure Call (RPC). In this way, they avoids modifications to the source code by intercepting at the system layer but has to offload the RPC calls to the cloud servers one by one and add once Round-Trip Time (RTT) of RPC to the completion time of each operator, bringing extra communication cost.

Moreover, such communication cost caused by the transparent offloading mechanism becomes substantial in robotic IoT networks, rendering those outstanding scheduling algorithms ineffective. ML models commonly have hundreds of operators (e.g., 451 for []), resulting in hundreds of RPC calls for a single inference. In robotic IoT networks, which typically rely on wireless communication for robots, each RPC RTT usually takes 2-5 milliseconds[]. Consequently, transparent offloading in robotic IoT networks leads to significant communication costs (e.g., 90% inference time for xxx) and energy wastage (e.g., xx% for xxx).

The key reason why existing transparent offloading methods suffer from communication cost is that they only call RPC when the operator has been used in the inference process. This is because they are designed for general applications to use remote GPUs, rather than specifically for ML model inference. Such communication costs cannot be avoided in general applications, as they do not know the future operators from the upper-layer applications and cannot call RPCs in advance. Fortunately, in the scenario of ML model inference, we find that the operators invoked by ML models are usually fixed in order and can be predicted in advance for the inference process.

Based on this observation, we present **RRTO**, a **T**ransparent

Offloading 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 calls the RPCs of the corresponding operators during inference in advance, without waiting for the operator to be called during inference. Based on this approach, RRTO achieves nearly the same communication cost as untransparent offloading methods, allowing the outstanding scheduling algorithms in untransparent offloading methods to be used in transparent offloading. However, the design of RRTO is confronted with a major challenge: how to identify the specific operators invoked during each inference throughout the log records without the help of hints from the upper-layer applications?

To address this challenge, RRTO propose a novel algorithm, *Data Dependency Search*, to find out which operators are invoked for each inference. This algorithm first build a relationship graph according to the data dependencies between operators (i.e., the calculation result of the previous operator serves as the input for the next operator). Then, take operators that does not depend on any other operators as starting operators and takes operators that does not have any operators dependent on them as ending operators. Finally, this algorithm searchs for the longest covering operator sequence between starting operators and ending operators, and verifys whether such operator sequence can constitute a complete model inference process (i.e., the entire log records of operators can be covered by repeating this sequence).

We implemented RRTO on the Cricket's codebase[2] and also incorporated the outstanding scheduling algorithms of MCOP[4] into RRTO. We evaluated RRTO on a real-world Jetson Xavier NX robot capable of GPU accelerated computation with a robotic application that tracks people in real time[]. We compared RRTO with local computation, a state-of-the-art (SOTA) untransparent offloading method (MCOP[4]) and a SOTA transparent offloading method (Cricket[2]) 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 reduces inference time by xx% compared to other baselines, similar to the xx% reduction achieved by the SOTA untransparent offloading method.
- RRTO is energy-efficient. It saves xx% in energy consumption compared to other baselines, similar to the xx% savings of the SOTA untransparent offloading method.
- 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 devices) changes, RRTO's superior performance remains consistent.

Our main contribution is RRTO, the first high-performance transparent offloading system designed for model inference on robotic IoT. RRTO dramatically reduces the communication cost caused by the traditional transparent offloading mechanism via the novel record/replay mechanism, achieving the same high performance as the SOTA untransparent 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/xxxx/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 diagram of the entire application is shown in the left part of Figure 1:

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**??**).

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 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 shown in the right part if Figure 1:

(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.

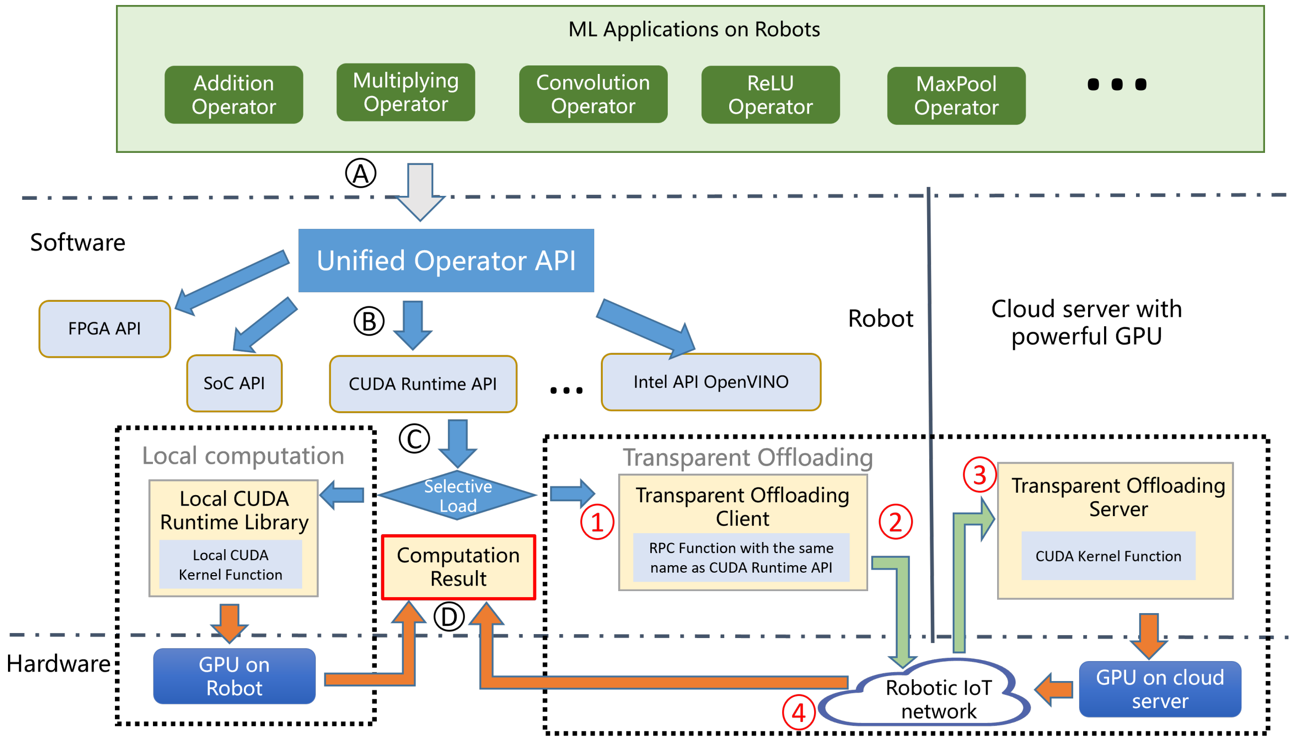

Fig. 1: Workflow of Transparent Offloading System for Model Inference on Robotic IoT

(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 returns the results to the upper-layer application.

B. Related Work

Model Compression. Quantization and model distillation are the two most commonly used methods of ML model compression on the robots. Quantization[] 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) fixed-point representations, with minimal loss of model accuracy. Model distillation[], 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 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 (aggregate multiple RPC calls into a single request), Asynchronous RPC (decouple the request and response processing), and Caching (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 strategies and transparent offloading system for model inference.

III. Overview

A. Workflow of RRTO

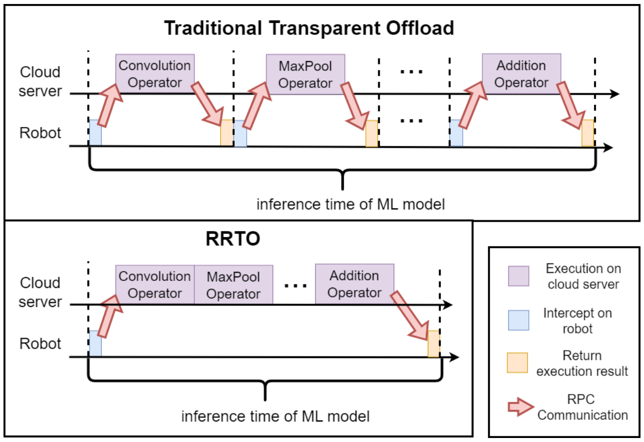

Fig. 2: Workflow of RRTO

Figure 2 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 data center networks, where devices connected with high-speed (e.g., 40 Gbps ~500 Gbps) networking technologies, such as InfiniBand[] or PCIe[]. 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., 451 for []). For edge devices connected via Wi-Fi 6, the actual bandwidth reaches only 1.2 Gbps, leading to an average RTT of xx milliseconds for each operator and the communication cost accounting for xx% of the total inference time in our experiments. For cloud servers, the 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 replay 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 RPC communication from the offloading client side, thus avoiding the communication cost associated with these operators.

Notice that, although there has been substantial work on optimizing RPC communication[], 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 corresponding operators' functions on the offloading server side.

B. Architecture of RRTO

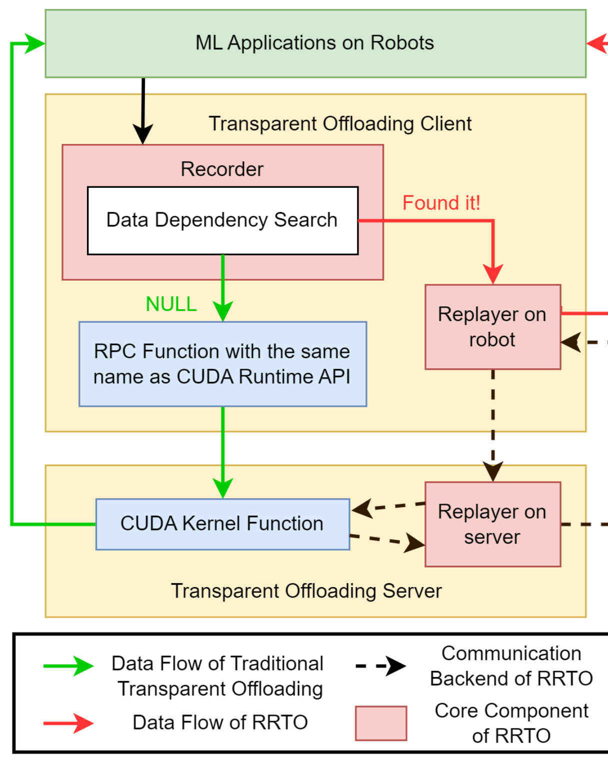

Fig. 3: Architecture of RRTO

Figure 3 presents the architecture of RRTO. In comparison with Figure 1, 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-B. 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 3.

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 3. Similar to Caching 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 untransparent offloading methods, as it needs to transmit almost the same input and output as these methods.

Notice that, the scheduling approach utilized above offloads all computation of model inference to the cloud server. Existing computation offloading systems, such as MCOP[4], adopt more varied and flexible scheduling approaches. 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. Some works[] have gone further by achieving a tradeoff between energy consumption and inference times. RRTO provides these scheduling approaches with a lower-level and finer-granularity scheduling method while maintaining nearly the same communication cost. However, this is a completely new field and is not 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.

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

IV. Design

A. Record/Replay Mechanism

Here we present how RRTO to achieve 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-B.

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 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.

Algorithm 2: RRTO_on_Server

```
Input: client task task
Data: inference operator sequence IOS = ∅, the execution
      result ret
if task == SendRPCtoServer then
    func, args = GetClientInput()
    ret = CUDARuntimeLibrary(func, args)
end
else
    // replayer on server
    args, IOS = GetClientInput()
    foreach Op ∈ IOS do
        args =
         RRTOFixArgs(Op["args"], ret, args)
        ret =
         CUDARuntimeLibrary(Op["func"], args)
    end
end
SendExecutionResultBack(ret)
```

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 = ∅ and relationship graph
         map = ∅
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
       // cannot find
11 end
```

B. 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, where there is no direct way to determine which inference the operator is invoked by.

The pseudocode for data dependency search is provided in Algo. 3. Due to data dependencies between operators (i.e., the calculation result of the previous operator serves as input for the next operator), RRTO first records the data dependencies between operators and constructs a relationship graph (line 2). These dependencies can be established by comparing whether parameters and calculated results between operators are the same (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 do not have any operators dependent on them 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.

V. IMPLEMENTATION

We implemented RRTO within Cricket's codebase[2], 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[1], a transport-independent RPC library for Linux. RRTO integrates its recorder and replayers into the corresponding RPC functions in Cricket and adapts MCOP's scheduling approach[4] 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 custom four-wheeled robot (Fig 4), equipped with a Jetson Xavier NX 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 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.

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 and encompasses the energy consumption of both the CPU and

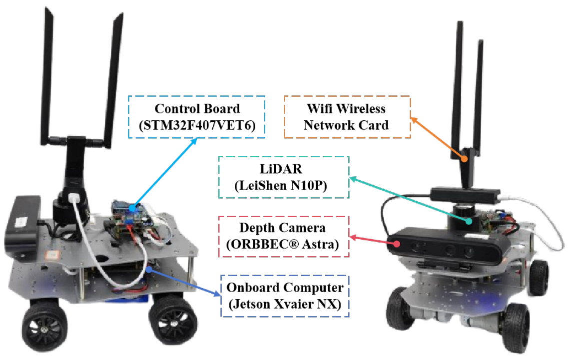

Fig. 4: The detailed composition of the robot platform

	inference	communication	standby
Power (W)	13.35	4.25	4.04

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

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 a edge device, which is a PC equipped with a xxx CPU and xxx GPU, connected to our robot via Wi-Fi 6 with an average bandwidth of 1 Gbps in our experiments. In the cloud scenario, computation is offloaded to a cloud server, which is a GPU server equipped with a xxx CPU and xxx GPU, 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. The detailed workflow is described as follows: The ORBBEC Astra 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[3]. 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.

Fig 5 is a screenshot of our real-world experiment, also presented in our video demonstration. In the screenshot, 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.

Baselines. We compared RRTO with local computation, a state-of-the-art (SOTA) untransparent offloading method (MCOP[4]) and a SOTA transparent offloading method (Cricket[2]) when offloading computation to different GPU devices (namely 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*

B. *Micro-Event Analysis*

To further understand the performance gain of RRTO, we recorded how RRTO and Cricket handle the operators in KAPAO, as shown in Fig. **??**.

C. *Sensitivity Studies*

D. *Lessons learned*

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 finer granularity offloading schedules as described in Sec. III-B. We believe such investigation will enable even faster, more energy-efficient and more robust inference capabilities for ML applications in real-world Robotic IoT Networks.

VII. Conclusion

The conclusion goes here.

Acknowledgments

References

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

[2] 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.

[3] 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.

[4] 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.

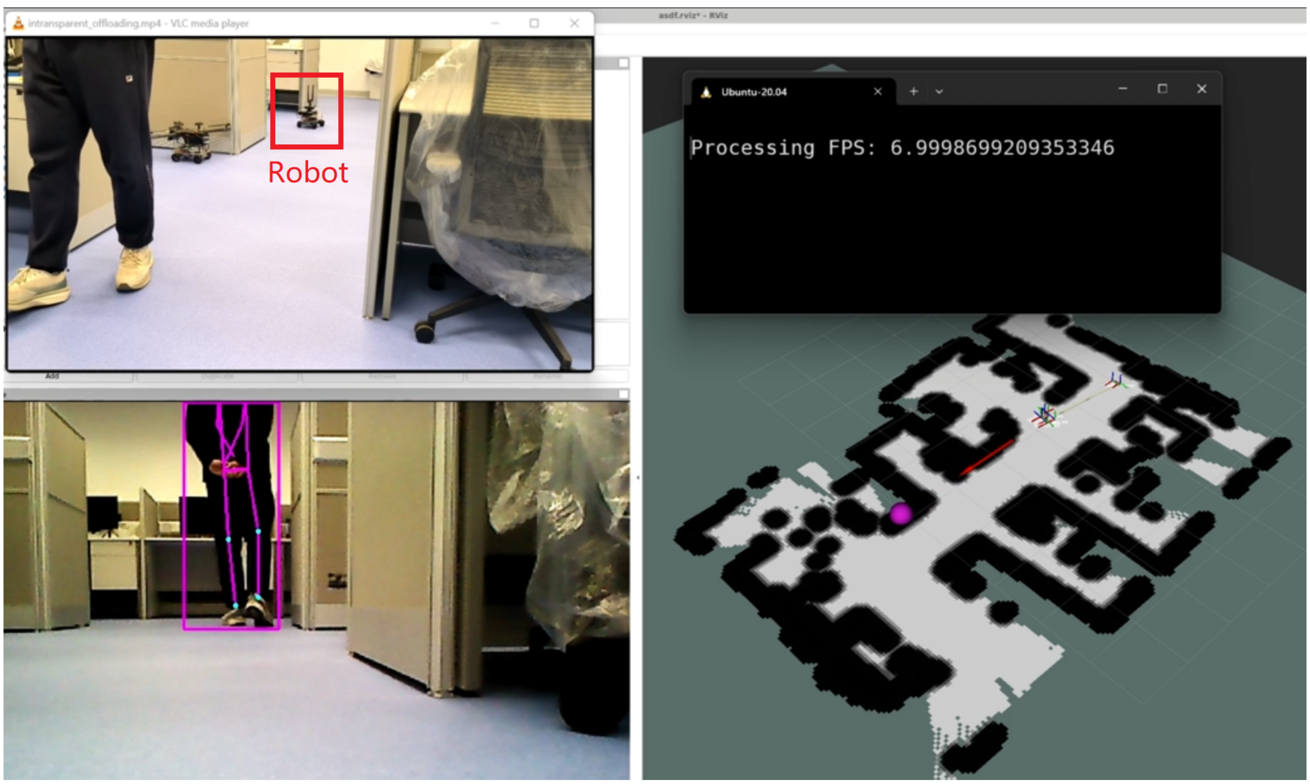

Fig. 5: A real-time people-tracking robotic application on our robot.