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[2], 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[1], 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 correspondin 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[1] and also incorporated the outstanding scheduling algorithms of MCOP[2] 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[2]) and a SOTA transparent offloading method (Cricket[1]) 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.

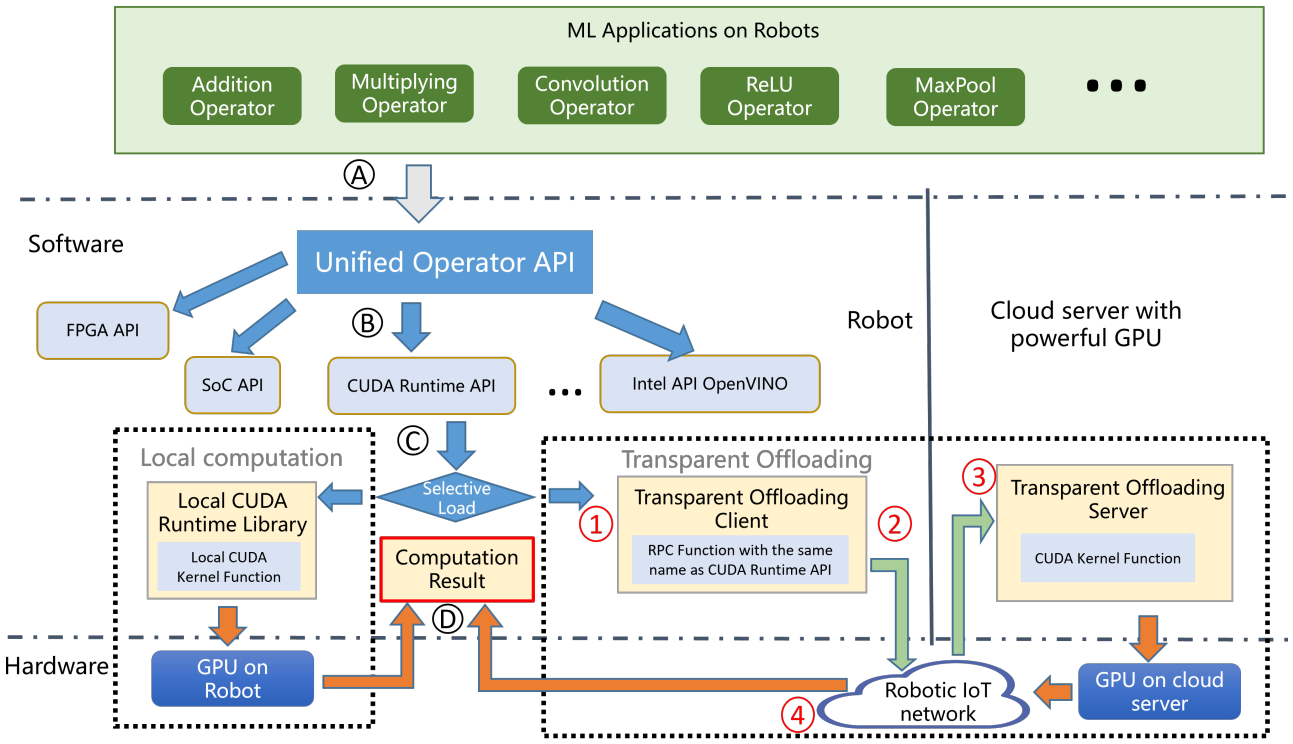

Fig. 1: Workflow of Transparent Offloading System for Model Inference on Robotic IoT

(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. And the record/replay mechanism of RRTO can be considered as a specific RPC optimization strategy designed for model inference.

III. Overview

A. Workflow of RRTO

Figure 5 presents the workflow of RRTO.

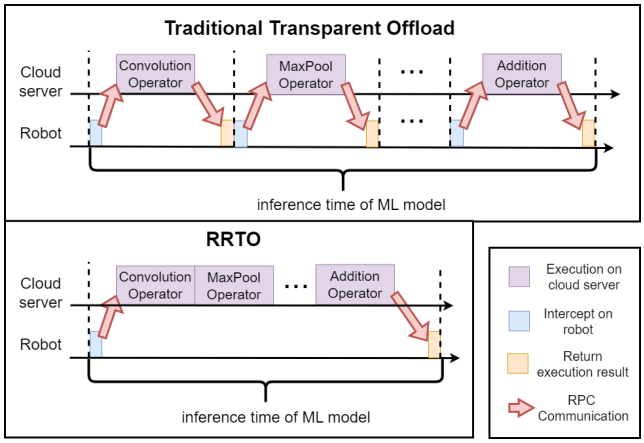

Fig. 2: Workflow of RRTO

The core problem facing this NVIDIA GPU power offloading system is how to solve the high overhead caused by wireless transmission in transparent offloading in real robot environment. As the figure below shows, in a real robot environment, a single application needs to call the CUDA kernel thousands to tens of thousands of times per minute. For example, one gesture recognition with KAPAO requires about 20 CUDA kernel calls, while smooth gesture recognition calculations require gesture recognition at a frequency of no less than 10hz, which means that KAPAO needs to call the CUDA kernel at least 12,000 times per minute. If you use the simple transparent offloading method, that is, one RPC transfer per CUDA kernel call. In the case of network fluctuations, according to the WiFi6 standard, a round trip delay (RTT) takes 2ms to calculate, and only the wireless network transmission process (that is, the RTT process) consumes 24s in 12,000 RPC transfers. Therefore, the use of simple transparent uninstall methods can introduce serious network transmission overhead, which in turn affects the uninstall computing experience. For example, for vacuum cleaners, high latency can lead to sluggish path planning, making the robot slower to complete cleaning tasks or even run into obstacles.

Therefore, this NVIDIA GPU power offloading system proposes an automatic recording and replay mechanism to reduce the network transmission overhead during the transparent offloading process of GPU power. For the same household robot application, the order of CUDA cores invoked to process different data inputs is usually fixed. For example, multi-person pose recognition requires different images to be input into the same neural network model for recognition. The automatic recording replay mechanism automatically records the CUDA kernel call sequence of the model used by the home robot application and automatically repeats the use process of the model on the GPU service area. This mechanism does not need to wait for the robot to repeat the specific CUDA kernel call during RPC transmission, which greatly reduces the number of repeated RPC transfers and thus reduces the network transmission overhead.

B. Architecture of RRTO

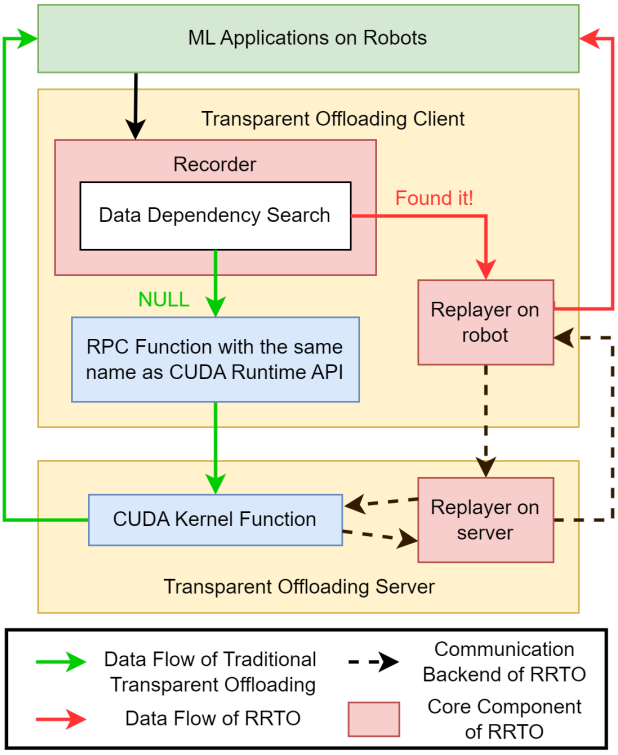

Fig. 3: Architecture of RRTO

Figure 3 presents the architecture of RRTO.

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.

B. Algorithm of Data Dependency Search

The pseudocode of data dependency search is given in Algo. 3. 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).

V. Implementation

This demo file is intended to serve as a “starter file” for the Robotics: Science and Systems conference papers produced under LaTeX using IEEEtran.cls version 1.7a and later.

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: operators for each inference $OFEI = \emptyset$

if $OFEI.empty()$ **then**
  // recorder
  $SendRPCtoServer(func, args)$
  $OFEI = DataDependencySearch(func, args)$
  $ret = GetRPCExecutionResult()$
  $RecordReturn(ret)$
end
else
  // replayer on robot
  if $func == OPEI.start()["func"]$ **then**
    $ret = StartRRTO(args, OFEI)$
    // start a new inference
  end
  else if $func == OPEI.end()["func"]$ **then**
    $ret = WaitingForRRTO()$
    // Waiting for the final computation result
  end
  else
    $ret = OPEI.find(func)["ret"]$
  end
end
return ret

Algorithm 2: RRTO_on_Server

Input: client task $task$
Data: operators for each inference $OFEI = \emptyset$, the execution result ret

if $task == SendRPCtoServer$ **then**
  $func, args = GetClientInput()$
  $ret = CUDARuntimeLibrary(func, args)$
end
else
  // replayer on server
  $args, OPEI = GetClientInput()$
  foreach $Op \in OPEI$ **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: operators for each inference $OFEI$
Data: Log records $history = \emptyset$ and relationship graph $map = \emptyset$

$history.add(func)$
$map.update(func, args)$
$StartPoses = map.startposes()$
$EndPoses = map.endposes()$
$Sequence = FindLongestPair(StartPoses, EndPoses)$
if $Verify(history, Sequence)$ **then**
  return $Sequence$
end
else
  return $NULL$
  // cannot find
end

VI. Evaluation

Testbed. The evaluation was conducted on a custom four-

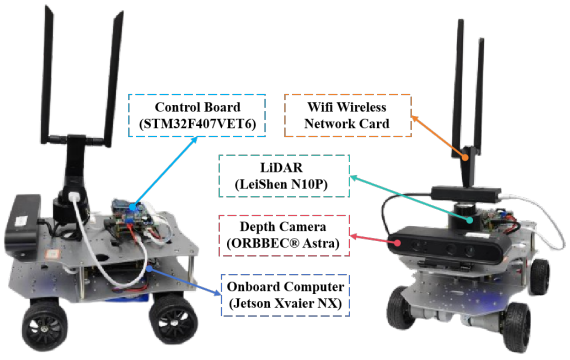

Fig. 4: The detailed composition of the robot platform

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.

Real-world Robotic application.
a robotic application that tracks people in real time[].

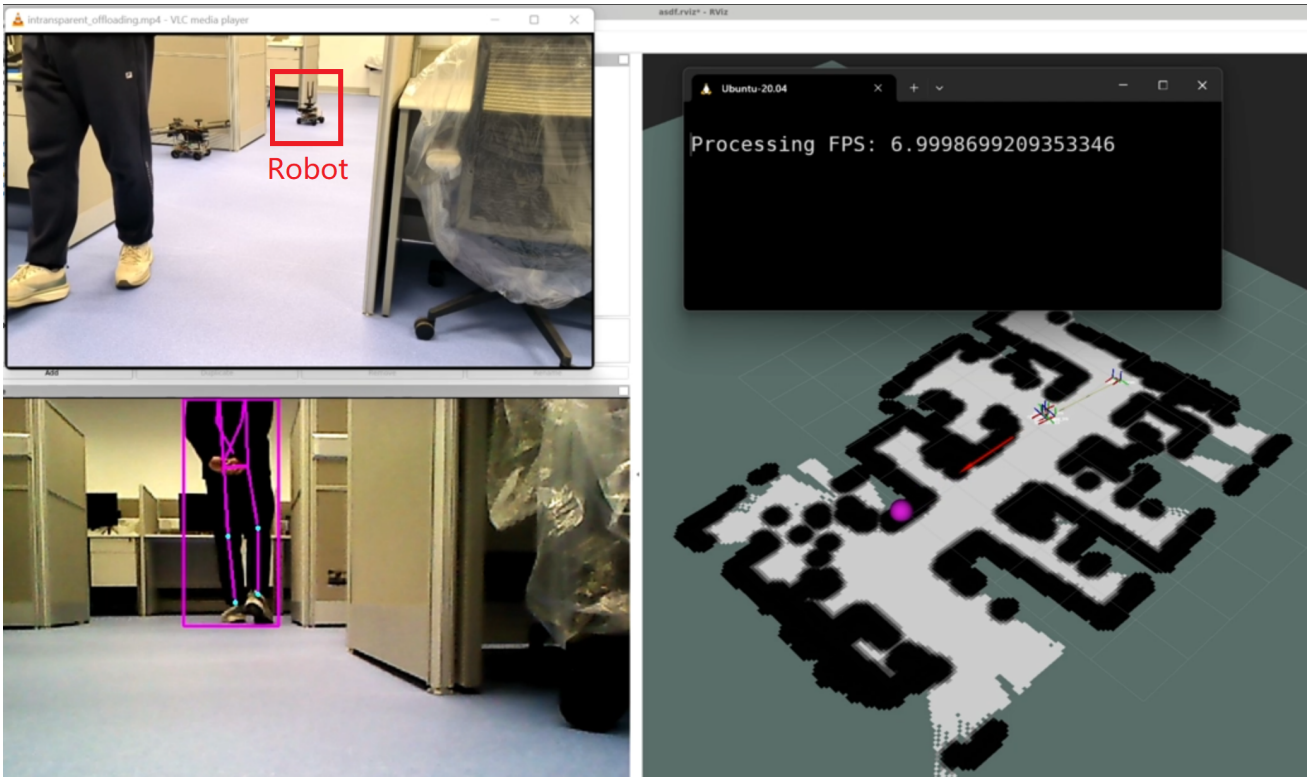

Fig. 5: A robotic application that tracks people in real time[].

Baselines.

We compared RRTO with local computation, a state-of-the-art (SOTA) untransparent offloading method (MCOP[2]) and a SOTA transparent offloading method (Cricket[1]) 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

C. Sensitivity Studies

D. Lessons learned

VII. Conclusion

The conclusion goes here.

Acknowledgments

References

[1] 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.

[2] 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.