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

Zekai Sun[1,2], Xiuxian Guan[1], Junming Wang[1], Yuhao Qing[1], Haoze Song[1], Dong Huang[1], Fangming Liu[3,4], Heming Cui[1,2,*]

***Abstract*—In the field of robotics, fundamental tasks such as object identification and robot control increasingly rely on Machine Learning (ML) models deployed on real-world robots. The inference of these models, which require extensive computation due to numerous parameters and complex operations (e.g., matrix multiplication, convolution), are often offloaded to GPU servers (e.g., edge devices with powerful GPUs) via the Internet of Things (IoT) for fast and energy-efficient inferences. Existing computation offloading systems are divided into transparent and non-transparent methods, with the latter necessitating modifications to the source code. While non-transparent offloading systems have proven effective with various ML models, they present significant deployment challenges when used on robots, due to the extensive coding efforts needed to modify source code for each specific application and their unsuitability for closed-source environment. Conversely, transparent offloading methods, which offload all function calls of each operator within the ML models via Remote Procedure Call (RPC) at the system layer, avoid the need for source code modifications. However, the frequent handling of RPCs on a one-by-one basis results in significant communication costs, which is particularly pronounced in robotic IoT environments where the need for high mobility often forces robots to rely on wireless connections.**

We present RRTO, the first high-performance transparent offloading system specifically designed for ML model inference on robotic IoT with a novel record/replay mechanism. Recognizing that operators in ML models for robotic tasks typically adhere a fixed sequence, RRTO automatically records and identifies these sequences during their initializing (first) inference, and replays them accurately in subsequent inferences. This mechanism enables RRTO to call all operators involved in the correct sequence through a single RPC at the start of each inference and eliminate the need to wait for the RPCs of subsequent operators during the inference process. Consequently, RRTO significantly cuts down the communication overhead caused by frequent RPCs in the traditional transparent offload mechanism, while maintaining the advantage of not requiring source code modifications, unlike non-transparent offloading systems. Evaluations show that RRTO significantly boosts the performance of robotic applications on our robots with a reduction of 90% to 98% in inference time and 89% to 98% in energy consumption per inference compared to the state-of-the-art transparent method, achieving comparable results to non-transparent methods without requiring any modifications to the source code.

***Index Terms*—Computation Offloading, Model inference, Robotic IoT, Distributed system and network**

I. Introduction

The rapid advancement of machine learning (ML) methods has led to significant achievements in fundamental robotic tasks such as object detection [1]–[3], robot control [4]–[6], and environmental perception [7]–[9], as shown by our real-world implementations (refer to Fig. 1). Deploying these ML methods onto real-world robots typically necessitates supplementary hardware due to the computationally demanding nature of ML models (e.g., the large number of parameters and complex operations). However, integrating computing accelerators (e.g., GPU [10], FPGA [11], SoC [12]) directly onto robots leads to substantial economic and energy expenditures, exemplified by a average $2.5\times$ increase in energy consumption per unit time on our robots. Therefore, many ML applications now prefer to offload the computation of ML models to GPU servers (e.g., edge devices with powerful GPUs) through the Internet of Things for robots (robotic IoT), which not only mitigates local resource limitations but also improves energy efficiency.

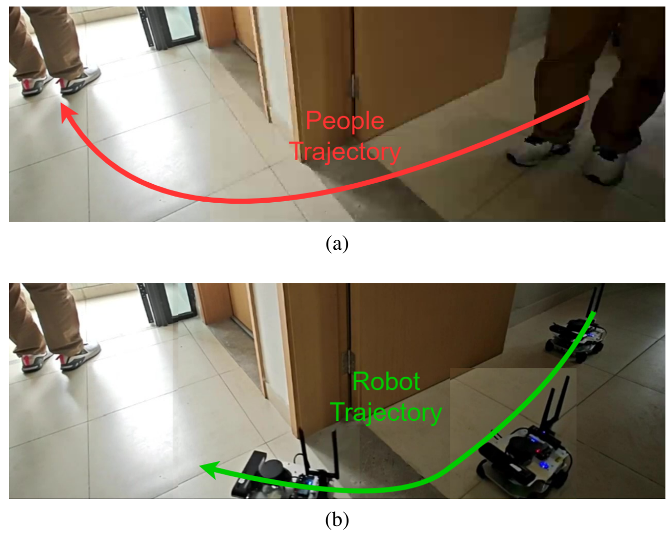


Fig. 1. A real-time people-tracking robotic application on our robot based on a well-known human pose estimation ML model, KAPAO [3].

Existing computation offloading systems are categorized into two categories: transparent and non-transparent methods, distinguished by the necessity for source code modifications to enable offloading. **Non-transparent** offloading methods require modifications to the source code, relocating the execution of model inference from local computing accelerators to GPU servers. The native method that offload all inference computations to GPU servers (places the entire ML model onto GPU servers) is prevalent in existing non-transparent offloading systems and has notably enhanced the performance of real-

*Corresponding author.
[1]The University of Hong Kong, Hong Kong SAR, China. [2]Shanghai AI Laboratory. [3]Peng Cheng Laboratory. [4]Huazhong University of Science and Technology. zksun@cs.hku.hk

world robotic applications, achieving $3.7\times$ faster inference and reducing energy consumption per unit time by 49% compared with local computation in our experiments (as indicated by "NNTO"). Furthermore, the enhanced performance of offloading facilitates various advanced scheduling optimizations, such as layer partitioning [13]–[15] and multiple inference scheduling [16]–[21], which have been incorporated into non-transparent offloading systems (see Sec. II-B for more details). However, despite their effectiveness across various ML models, non-transparent offloading systems demand 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 [22], provide a convenient way to offload computation to GPU servers, though at the cost of reduced performance. In ML model inference, which involves a series of operators (e.g., addition, convolution), transparent offloading methods intercept each operator call (e.g., *torch.nn.functional.add()*, *torch.nn.functional.conv2d()* in PyTorch [23]) to the corresponding system functions (e.g., *aten::add*, *aten::conv2d* within the "cudaLaunchKernel" function for CUDA [24]). These intercepted calls are then offloaded to GPU servers through Remote Procedure Calls (RPC), facilitating the computation process on remote servers. This approach avoids the need for source code modifications by intercepting function calls at the system layer. However, each operator's function call must be offloaded individually through RPC, adding a Round-Trip Time (RTT) to the completion time of each operator, thereby incurring additional communication costs.

In robotic IoT networks, the inherent communication costs associated with transparent offloading mechanisms become significant. ML models typically employ hundreds of operators (e.g., 522 for [3]), along with additional RPC functions during runtime across various ML frameworks (e.g., "cudaGetDevice", "cudaLaunchKernel" in PyTorch [23], as detailed in Sec. VI-B), resulting in hundreds or even thousands of RPC calls for a single inference process (e.g., 5895 for [3]). Given that robotic IoT networks often use wireless communication to support the high mobility of robots, each RPC Round-Trip Time (RTT) can take several milliseconds [25]. Consequently, even the most advanced state-of-the-art (SOTA) transparent offloading method, such as Cricket [22], significantly extends communication time (accounting for 95% of the inference time in our experiments) and may increase the total energy consumed for each inference, despite lowering the energy consumption per unit time required for computation on robots through offloading.

The primary reason existing transparent offloading methods incur significant communication costs lies in their reactive approach. These systems invoke RPCs only after an operator has been used during the inference process, instead of proactively anticipating and preparing for their use. This is because these methods are generally designed to leverage remote GPUs for a broad range of applications, not specifically for ML model inference. In such general applications, future operators used by upper-layer applications are unpredictable, making it impossible to call RPCs in advance, and thus incurring inevitable communication costs. However, in the context of ML model inference for robotic applications, we observe that these ML models are often static, with operators invoked in a fixed and predictable sequence during inference (refer to Sec. III-A for more details). This predictability provides an opportunity to develop a more efficient transparent offloading method that leverages the static nature of ML models in robotic applications to reduce communication overhead significantly.

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: automatically records the sequence of operators invoked during an ML model's inference and replays these fixed-order operators during subsequent inferences. Consequently, RRTO executes all required operators in the recorded sequence through a single RPC at the beginning of each inference, eliminating the need for additional RPCs for subsequent operators during the inference process.

However, identifying the specific operators invoked during each inference is a challenging task. The transparent offloading system operates by intercepting function calls from upper-layer applications to GPU devices at the system layer, where it can only access log records of the operators called. During the initializing (first) inference, some ML models may generate varying operator sequences, such as [3] (as shown in Sec. VI-B), and when performing multiple inferences, it produces repeated sequences in the log records. This repetition complicates the task of discerning which operators are associated with each inference and identifying the end of one inference and the start of another. RRTO must meticulously parse log records to accurately identify the sequence of operators for each inference, as reliably replicating the correct sequence is critical for ensuring accurate results; even a single operator discrepancy can compromise the correctness of the inference outcome.

To tackle this challenge, RRTO proposes a novel algorithm named *Data Dependency Search* that identifies the sequence of operators involved in each inference. This algorithm begins by constructing a relationship graph that maps the data dependencies among operators, where the output of one operator serves as the input for the next. It identifies operators with no dependencies as potential starting points and those that are not dependent on others as possible ending points. The algorithm then seeks the longest sequence of operators that spans from a starting point to an ending point and checks if this sequence can represent a complete model inference process by covering the entire log records of operators when repeated. Additionally, the requirement to send the final computation result from the GPU back to the CPU provides a crucial clue in pinpointing the correct ending operator, significantly narrowing the search space for the algorithm.

We implemented RRTO on the Cricket's codebase [22] and evaluated RRTO on our real-world robot equipped with Jetson Xavier NX [26], using a typical real-world robotic applications [3] and six common robotic visual models useful for three categories of fundamental robotic tasks (ResNet [27] and ConvNext [28] for object classification, FCN [29] and DeepLabv3 [30] for semantic segmentation, FasterRCNN [31] and RetainNet [32] for object detection). We compared RRTO

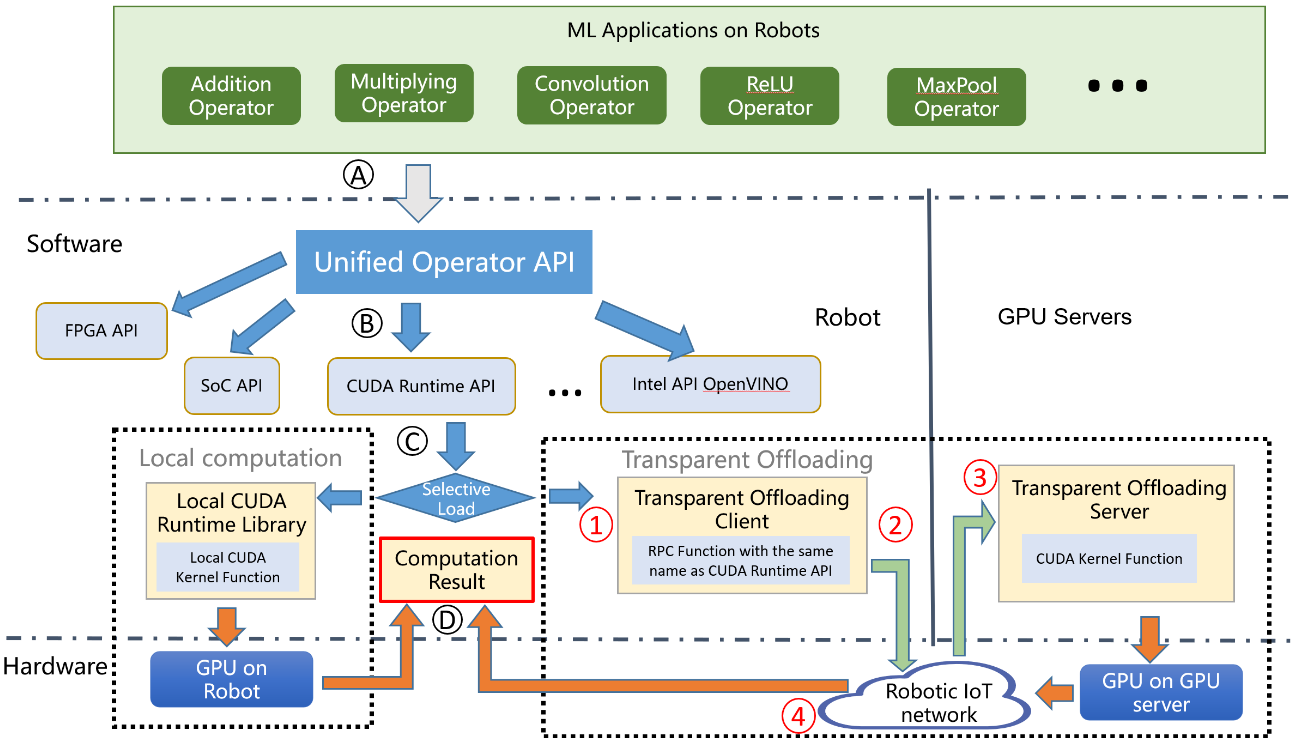


Fig. 2. Workflow of Transparent Offloading System for Model Inference on Robotic IoT.

with local computation that inference the entire model locally on the robot (referred to as "local"), the native non-transparent offloading system that offloads all inference computations to GPU servers (referred to as "NNTO"), and a SOTA transparent offloading system (Cricket [22]) under different real-world robotic IoT environments (namely indoors and outdoors). The evaluation shows that:

- RRTO is fast. It reduced inference time by 90% to 98% compared to the SOTA transparent baseline (Cricket), and by 69% to 94% compared to local computation (Local). These reductions are comparable to those achieved by non-transparent offloading systems (NNTO).
- RRTO is energy-efficient. It achieved an 89% to 98% reduction in energy consumption per inference compared to Cricket, and 83% to 97% compared to Local, mirroring the performance efficiencies of NNTO.
- RRTO is robust in various robotic IoT environments. When the robotic IoT environment changed, RRTO's superior performance remained consistent as NNTO.

Our main contribution is RRTO, the first high-performance transparent offloading system specifically designed for model inference on robotic IoT. RRTO dramatically reduces the communication costs typically incurred by frequent RPCs in the traditional transparent offloading methods through its novel record/replay mechanism. This enables RRTO to achieve performance comparable to non-transparent offloading methods, without requiring any modifications to the source code. We anticipate 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/hku-systems/RRTO.

In the rest of this paper, we provide the background in Sec. II, deliver an overview of RRTO in Sec. III, detail the design of RRTO in Sec. IV, evaluate the performance of RRTO in Sec. VI, and finally conclude in Sec. VII.

II. Background

A. Workflow of Transparent Offloading

When a robotic application utilizes the GPU on the robot for inference, the complete system call flow from top to bottom is illustrated in the left part of Fig. 2:

A The robot application sequentially invokes the corresponding operators based on the ML model's structure to complete the entire computation process.

B Each operator accesses the appropriate function library through a unified operator API tailored to the type of the device running the application; for instance, on NVIDIA GPUs, this is the CUDA runtime library ([24]).

C The operating system loads the local CUDA runtime library by default, and executes the relevant CUDA kernel functions on the robot's GPU.

D The local CUDA runtime library returns the computation results to the upper-layer application.

Transparent offloading methods [22], [33], [34] typically employ a strategy of rewriting dynamic link libraries by defining functions that share names with those in the CUDA runtime API and prioritizing the loading of these custom libraries using the *LD_PRELOAD* environment variable. This setup causes the dynamic linker to redirect calls from the original library functions to the custom library functions with the same name, effectively intercepting library functions. Subsequently, the custom library packages the function calls along with the required parameters and data, and sends them to the GPU server via RPC. It also modifies GPU memory management and the launching of CUDA kernel functions to ensure that RPCs are executed correctly on the GPU server. In this manner, these methods achieve transparent offloading by intercepting kernel functions at the system layer using similarly named functions in the dynamic link library and offloading their execution to the GPU server.

Compared to using the GPU on the robot, changes in the system call process primarily occur in step C. The detailed steps of the process (depicted in the right part in Fig. 2) are as follows:

(1) The dynamic link library is modified such that each operator prioritizes calling the RPC functions that have the same names as those in the CUDA runtime API, enabling the effective identification and interception of all CUDA kernel function calls.
(2) The transparent offloading client transmits the called CUDA runtime API and the required parameters to the GPU server via the robotic IoT network using RPC.
(3) The transparent offloading server on the GPU server launches the corresponding CUDA kernel functions and completes the computation.
(4) The computation results are sent back to the client. The transparent offloading system then returns these results to the upper-layer application.

B. Non-Transparent Offloading

Non-transparent offloading systems necessitate modifications to the source code, relocating the execution of model inference from local computing accelerators to GPU servers. By leveraging the powerful GPUs on these servers, the native method of offloading all inference computations to GPU servers (placing the entire ML model onto GPU servers, referred to as "NNTO" in this paper) not only significantly accelerates the inference process but also reduces the substantial energy consumption required for computation on robots. Additionally, various advanced scheduling optimizations have been developed to enhance offloading performance while accommodating application-specific needs such as inference speed and energy consumption. These optimizations primarily include layer partitioning [13]–[15] and multiple inference scheduling [19]–[21].

Layer partitioning aims to maximize the speed and energy efficiency of each individual inference by distributing portions of an ML model across GPU servers at a finer granularity than NNTO, specifically at the layer level. This method strategically assigns each layer to either robots or GPU servers, capitalizing on the fact that the output data in some intermediate layers of a DNN model is significantly smaller than its raw input data [35]. Effective implementation of layer partitioning demands a comprehensive understanding of the model's structure (e.g., computation times for each layer on robots and GPUs, as well as data transfer times), and making careful trade-offs between computation and transmission during the scheduling of each layer's execution, taking into account factors like network bandwidth, the computing power of GPU server, and specific application requirements.

Multiple inference scheduling optimizes numerous DNN inference tasks at a granularity broader than NNTO, aiming to enhance overall system efficiency. This approach uses various decision algorithms to strategically schedule the execution location and timing of multiple inference tasks, such as batching tasks together [16], prioritizing based on urgency [17], [18], and employing deep reinforcement learning controls [19]–[21]. Unlike layer partitioning methods that focus on optimizing each individual inference, these methods coordinate the overall offloading strategy across multiple tasks to minimize overall inference latency and energy consumption.

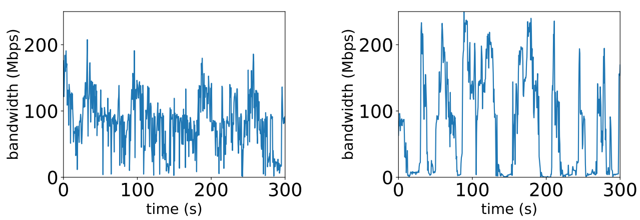
Fortunately, these advanced scheduling optimizations developed for non-transparent offloading systems can also be adapted to RRTO. For layer partitioning optimization, where a model layer often corresponds to one or several fixed operators (e.g., the convolution layer invoking a convolution operator via calling the "cudaLaunchKernel" function during inference), RRTO can leverage its Data Dependency Search's relationship graph to obtain the model's structure required by the scheduling algorithm of layer partitioning and refine the scheduling granularity from layer to operator. For multiple inference scheduling, these methods can be seamlessly integrated into RRTO by utilizing their decision algorithms to determine the execution location and start time of each inference task during the replay process of RRTO, as they do in non-transparent systems. This integration is feasible because these methods prioritize offloading system performance regardless of system transparency. Traditionally, such methods have favored non-transparent systems due to the poor performance of transparent systems caused by communication costs. However, if a transparent offloading system like RRTO achieves performance comparable to non-transparent systems, these multiple inference scheduling techniques can be successfully implemented to offer a high-performance, transparent offloading solution with sophisticated scheduling strategies.

In summary, while non-transparent offloading systems offer high performance, they are not user-friendly; conversely, transparent offloading systems are convenient but typically underperform. RRTO merges the best of both, providing a solution that is both high-performance and user-friendly. Additionally, the advanced scheduling optimizations that have been proven effective in non-transparent offloading systems can be adapted to RRTO, and we leave it as a future work.

C. Characteristics of Robotic IoT

In real-world scenarios, robots frequently navigate and move around to execute tasks such as search and exploration, relying on wireless networks that offer high mobility. However, these networks often have limited bandwidth, both due to the theoretical upper limit of wireless transmission technologies and the practical instability of wireless networks. For instance, the most advanced Wi-Fi technology, Wi-Fi 6, offers a maximum theoretical bandwidth of 1.2 Gbps for a single stream [36]. However, the limited hardware resources on robots often prevent them from fully utilizing the potential of Wi-Fi 6 [37]. Moreover, the actual available bandwidth of wireless networks is often reduced in practice due to various factors, such as the movement of devices [38], [39], occlusion by physical barriers [40], [41], and preemption of the wireless channel by other devices [42], [43].

To demonstrate the instability of wireless transmission in real-world situations, we conducted a robot surveillance experiment using four-wheel robots navigating around several given points at 5-40cm/s speed in our lab (indoors) and campus garden (outdoors), with hardware and wireless network settings as described in Sec. VI. We utilized iperf [44] to saturate the wireless transmission between our robot and a base station in the indoors and outdoors scenarios, thereby measuring the real-time maximum wireless bandwidth capacity and recording these values every 0.1 seconds over a period of 5 minutes.



(a) Indoors (b) Outdoors

Fig. 3. The instability of wireless transmission between our robot and a base station in robotic IoT networks.

The results in Fig. 3 show average bandwidth capacities of 93 Mbps and 73 Mbps for indoor and outdoor scenarios, respectively. The outdoor environment exhibited higher instability, with bandwidth frequently dropping to extremely low values around 0 Mbps, due to the lack of walls to reflect wireless signals and the presence of obstacles like trees between communicating robots, resulting in fewer received signals compared to indoor environments. It is important to note that the Round-Trip Time (RTT) for RPCs of each operator in traditional transparent offloading systems is relatively minimal in data center networks, where devices are equipped with high-speed networking technologies such as InfiniBand [45] or PCIe [46], offering bandwidths ranging from 40 Gbps to 500 Gbps. However, in robotic IoT networks, the bandwidth available between robots and GPU servers is typically more limited. Furthermore, when the GPU server is located in the cloud rather than at the edge, the RTT for RPCs increases significantly, impacting the overall system performance.

D. Related Work

Model Compression. Quantization and model distillation are the two most commonly used methods of ML model compression on the robots. Quantization [47]–[49] 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 [50]–[52], 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 (Remote Procedure Call) [53] is a communication protocol that allows one process to request services from another process on a remote computer, typically over a network. Common strategies to enhance RPC performance include Batching [54] (aggregating multiple RPC calls into a single request), Asynchronous RPC [55] (separating the request and response processes), and Caching [56] (storing results of prior RPC calls). However, these strategies are not effectively reducing communication costs during model inference. The inference process often requires waiting for one operator to finish before initiating the next, preventing the use of batching as each operator's RPC must be completed sequentially to ensure the correctness of the calculation logic. While Asynchronous RPC allows the client to execute other tasks (subsequent operators in model inference) without waiting for the server's response, it lacks mechanisms to determine when to halt asynchronous execution and return the final calculation to the robot, nor can it ensure that operators are executed in the correct sequence to yield accurate results. Furthermore, the unique input for each inference means that operators must be recalculated every time, rendering Caching ineffective. In contrast, RRTO further reduces the communication cost by eliminating 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 of RPC optimization strategies and transparent offloading systems for model inference.

III. Overview

A. ML Models with Fixed-Order Operators

In machine learning, many models have a fixed order in activating their layers during the inference process, referred to as static ML models. These include: 1. feed-forward neural networks (e.g., Multi-Layer Perceptrons [57], Convolutional Neural Networks [58]) with a fixed structure where neurons in each layer are activated sequentially given an input; 2. Recurrent Neural Networks (e.g., simple RNNs [59], Elman networks [60]) that have a fixed computation process at each time step, despite their ability to handle variable-length sequences; 3. Autoencoders (e.g., basic autoencoders [61], Variational Autoencoders [62]) with fixed encoder and decoder parts that activate the same components during each inference; 4. Generative Adversarial Networks (e.g., basic

GANs [63], DCGANs [64]) with fixed generator and discriminator structures; 5. shallow machine learning models (e.g., linear regression [65], logistic regression [66], Support Vector Machines [67]) that typically have a single fixed layer. These models with fixed structures and no dynamic mechanisms have relatively simple and regular computation processes and are the targeted models of our RRTO and the baselines.

On the other hand, some models may activate different layers during different inferences depending on the input, and are referred to as dynamic ML models. These include 1. models with attention mechanisms (e.g., Transformers [68], BERT [69]) that dynamically compute attention weights to focus on different parts of the input; 2. gated models (e.g., LSTM [70], GRU [71]) that control information flows through gating units, leading to different activated parts based on the input; 3. conditional computation models (e.g., Mixture of Experts [72]) that select different experts to activate based on input conditions; and 4. dynamic network models (e.g., those obtained through Neural Architecture Search [73]) that can dynamically adjust their structure based on the input. The dynamic nature of some ML models allows for better adaptability and expressiveness, but this nature also makes it difficult to profile their running statistics (time consumed, size of input and output, etc.) and thus few optimizing systems are targeted on these models.

Static ML models are widely used in robotic applications, such as Convolutional Neural Networks for computer vision tasks [3], [74], [75], and Multi-Layer Perceptrons, Recurrent Neural Networks, and Support Vector Machines for robot manipulation and automatic navigation [76]–[78]. On the other hand, dynamic ML models often require more computing resources and GPU storage, leading to their deployment mainly in data centers rather than on robots [68], [69], [72].

B. Workflow of RRTO

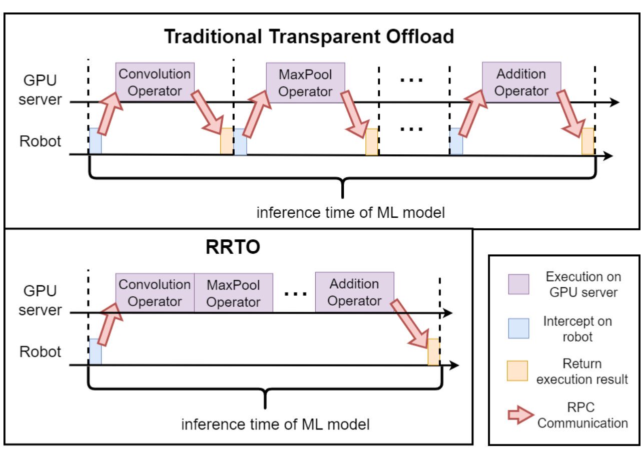


Fig. 4. Workflow of RRTO.

Fig. 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 are plagued by frequent RPC communication of operators, which incurs significant communication costs and compromises system performance, including reduced GPU utilization on GPU servers, prolonged model inference times, and heightened energy consumption per inference. These performance drawbacks are documented in both Fig. 4 and Sec. VI.

To address the communication costs in the transparent offloading systems, RRTO introduces an automatic recording and replay mechanism. Given that ML models in robotic applications often exhibit static, predictable sequences of operator invocation during inference, RRTO records the operators called during the initial inferences and replays this recorded sequence, referred to as the *inference operator sequence*, for subsequent inferences. As illustrated in Fig. 4, during the replay phase, RRTO only offloads the first operator in the sequence via RPC as done in traditional transparent offloading systems, streamlining the start of the inference task. Subsequent operators are directly invoked on the GPU server side, instead of being invoked by RPCs from the robot side as in traditional systems. Consequently, RRTO executes all required operators in the inference operator sequence with a single RPC at the beginning of each inference, eliminating the need for additional RPC communications for subsequent operators and significantly reducing communication costs.

It is important to note that while significant efforts have been made to optimize RPC communication [54]–[56], RRTO advances this by eliminating the RPC communication for Subsequent operators in the inference operator sequence altogether. Unlike existing methods that still depend on RPCs from the client to direct subsequent server actions, RRTO proactively executes these operators on the server, streamlining the process and enhancing efficiency.

IV. Design

A. Architecture of RRTO

Fig. 5 illustrates the architecture of RRTO. In comparison to Fig. 2, RRTO incorporates its record/replay mechanism into the core components of existing transparent offloading systems while maintaining transparency to upper-layer applications. This means that RRTO does not require any modifications to the source code for enabling offloading. The pseudo codes detailing the client and server sides of RRTO are provided in Sec. IV-B.

During the first several inferences, RRTO enters the recording phase and follows the execution pattern of traditional transparent offloading systems by offloading the execution of operators to the GPU server via RPC, as illustrated by the green lines in Fig. 5. Upon identifying and intercepting CUDA kernel function calls of operators from upper-layer ML applications, RRTO first records the function called by the operators, including the required parameters and the return value, using its recorder. Subsequently, it seeks to identify the inference operator sequence through a data dependency search, details of which are provided in Sec. IV-C.

Once the recorder identifies the inference operator sequence, RRTO transitions to the replaying phase, where the execution of the inference operator sequence is replayed for subsequent inferences using the replayer on both the robot and server,

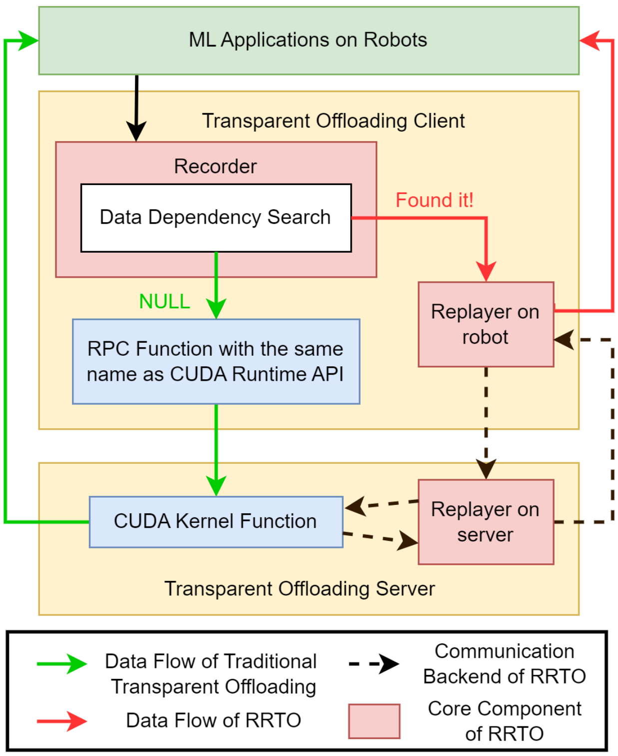


Fig. 5. Architecture of RRTO.

as illustrated by the red lines in Fig. 5. This process is akin to Caching (described in Sec. II-D) used in existing RPC optimization methods, where the replayer on the robot returns the execution results of previous RPC calls to the upper-layer applications. This allows the offloading client to continue execution until it is halted at the ending operator to receive the final computation result from the offloading server. Simultaneously, the offloading server replays the execution of the sent inference operator sequence and returns the final computation result to the offloading client. This approach enables RRTO to achieve communication costs nearly equivalent to those of non-transparent offloading methods, as it transmits almost the same input and output as these methods, depicted by the black dotted lines in Fig. 5.

The record/replay mechanism of RRTO only fails when the operator sequence changes, a scenario typical in dynamic ML models. If a prediction failure occurs (the replayer on the robot detects inconsistencies between actual operator calls and the identified inference operator sequence), RRTO temporarily suspends the record/replay mechanism, reverting to traditional transparent methods, and restarts the above process until a new operator sequence is established. However, ML models in robotic applications are often static (as discussed in Sec. III-A), making failures of RRTO's mechanism rare, thereby preventing its degradation into traditional transparent offloading systems. It is crucial to recognize that optimizing inference for dynamic ML models with changing operator sequences remains a challenge for all offloading systems, not just RRTO. Some research efforts [79], [80] explore methods to optimize inference for these dynamic models by statistically predicting which layers will activate in subsequent inferences, which is beyond the scope of this paper.

Moreover, when multiple inference tasks occur simultaneously on a robot, RRTO can effectively distinguish operators belonging to different tasks, thereby enhancing performance for each. Initially, RRTO's codebase, Cricket [22], distinguishes RPC functions from different inference processes, which enables the sharing of remote GPUs across multiple applications and ensures that results are accurately returned. Within a single inference process, RRTO's tailored record/replay mechanism identifies operators from distinct tasks. During the recording phase, its data dependency search accurately identifies the appropriate sequence, which is particularly beneficial for tasks that utilize the same model within the same process (see Sec. IV-C). During the replaying phase, the replayer on the robot not only checks for failures in the predicted operator sequence (see Sec. IV-A) but also detects new inference tasks, initiating new inferences through RRTO's record/replay mechanism. However, addressing performance issues arising from resource or network constraints due to concurrent tasks should involve multiple inference scheduling strategies, such as batching tasks (see Sec. II-D), rather than relying solely on RRTO. Future work will focus on integrating these strategies within RRTO to optimize performance further.

B. Record/Replay Mechanism

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
Parameter: inference operator sequence IOS

```
IOS ← ∅
while True do
    if IOS.empty() then
        # recorder
        SendRPCtoServer(func, args)
        IOS ← DataDependencySearch(func, args)
        ret ← GetRPCExecutionResult()
    else
        # replayer on robot
        if func is IOS.start()["func"] then
            # start a new inference
            ret ← StartRRTO(args, IOS)
        else if func is IOS.end()["func"] then
            # waiting for the final computation result
            ret ← WaitingForRRTO()
        else
            # returning the execution results of previous RPC calls
            ret ← IOS.find(func)["ret"]
        end if
    end if
    ReturnResult(ret)
end while
```

Here we describe how RRTO implements its record/replay mechanism. The transparent offloading process is outlined in two parts: the client component is detailed in Alg. 1, and the server component in Alg. 2. Further details on the data dependency search, which is central to the mechanism, are provided in the subsequent subsection IV-C.

In Alg. 1 on the offloading client side, RRTO takes a CUDA kernel function called by an operator and the requisite parameters as input. Initially, the algorithm checks whether the recorder has already identified the inference operator sequence to determine if the current phase should be recording or replaying. If the sequence has not been established, RRTO enters the recording phase (lines 4 to 6), which involves sending an RPC to the server (line 4), performing a data dependency search (line 5), and capturing and recording the RPC execution result (line 6), adhering to the execution pattern of traditional transparent offloading systems. To enhance system efficiency, RRTO overlaps the data dependency search with the RPC execution, allowing the algorithm to complete while the client awaits the RPC result. If the inference operator sequence is already identified, RRTO transitions to the replaying phase on the robot (line 8 to 14). This phase starts with initiating RRTO for a new inference at the first operator (line 9), returning the execution results of previous RPC calls for intermediate operators within the sequence (line 13), and finally, awaiting the final computation result at the last operator (line 11).

Algorithm 2 RRTO_on_Server

Input: client task $task$
Parameter: inference operator sequence IOS, the execution result ret
1: $IOS \leftarrow \emptyset$
2: **while** True **do**
3:     **if** $task$ is $SendRPCtoServer$ **then**
        # Same workflow as traditional transparent offloading systems
4:         $func, args \leftarrow GetClientInput()$
5:         $ret \leftarrow CUDARuntimeLibrary(func, args)$
6:     **else if** $task$ is $StartRRTO$ **then**
        # replayer on server
7:         $args, IOS \leftarrow GetClientInput()$
8:         **for all** $Op \in IOS$ **do**
9:             $args \leftarrow RRTOFixArgs(Op[''args''], ret, args)$
10:             $ret \leftarrow CUDARuntimeLibrary(Op[''func''], args)$
11:         **end for**
12:     **end if**
13:     $SendExecutionResultBack(ret)$
14: **end while**

In Alg. 2, the RRTO offloading server continuously awaits tasks from the client, aligning its operational phase with that of the client to ensure synchronized processing. During the recording phase, the server processes RPC requests similarly to traditional transparent offloading systems (lines 4, 5). As the client moves into the replaying phase on the robot, the offloading server concurrently transitions to its replaying phase (line 7 to 11), effectively replaying the execution of the recorded inference operator sequence. During this phase, RRTO adjusts the parameters required by the corresponding operators (line 9), which typically include the data or addresses of computation results from previous operators within the current inference task, ensuring accurate and efficient execution.

C. Algorithm of Data Dependency Search

The performance of RRTO hinges critically on its ability to accurately identify the correct inference operator sequence. Any discrepancy, even if it involves just a single operator, can prevent RRTO from achieving the correct inference result. This identification process is challenging because RRTO must retain its transparency and cannot derive any clues about the invoked operators from upper-layer applications for each inference. Consequently, RRTO relies solely on the log records of operators from the initial few inferences to determine the starting and ending operators, as well as those in between.

Algorithm 3 DataDependencySearch

Input: Cuda kernel function called by the last operator $func$ and the required parameters $args$
Output: inference operator sequence IOS
Parameter: Log records $history = \emptyset$ and relationship graph $map = \emptyset$
1: $history.add(func)$
2: $map.update(func, args)$
3: $StartPoses \leftarrow map.startposes()$
4: $EndPoses \leftarrow map.endposes()$
5: $Sequence \leftarrow FindLongestPair(map, StartPoses, EndPoses)$
6: **if** $Verify(history, Sequence)$ **then**
7:     **Return** $Sequence$
8: **else**
    # cannot find
9:     **Return** $NULL$
10: **end if**

The pseudo code for the data dependency search is outlined in Alg. 3. RRTO first records data dependencies between operators (i.e., the output of one operator serves as the input for the next) and constructs a relationship graph (line 2). These dependencies are determined by checking if input parameters and output results between operators match (i.e., share the same address), thus enabling RRTO to establish the inference operator sequence from this graph. Operators that do not rely on others are identified as potential starting points (line 3), while those not depended upon by any are considered potential ending points (line 4). RRTO then searches for the longest sequence that spans from a starting to an ending operator (line 5) and verifies if this sequence can represent a complete model inference process (line 6) by ensuring the entire log of operator records can be consistently replicated using this sequence.

During our implementation, we observed that the final computation result is transferred from the GPU to the CPU, and the ending operator can be pinpointed by comparing the address of this final result. This insight significantly assists RRTO in identifying the correct ending operator and substantially narrows 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.

some ML models, such as KAPAO [3], may display varying operator sequences during their initializing (first) inference. To address this, RRTO captures all operators across the the first several inferences, not solely the first, and ignores any variations from the initializing inference once the inference operator sequence is determined (see more details in Sec. VI-B).

V. Implementation

We implemented RRTO within Cricket's codebase [22], 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 [53], a transport-independent RPC library for Linux. We integrate RRTO's recorder and replayer into the corresponding RPC functions in Cricket, allowing for seamless integration and efficient operation of the record/replay mechanism.

VI. Evaluation

Testbed. The evaluation was conducted on a customized four-wheeled robot (Fig. 6), equipped with a Jetson Xavier NX [26] 8G onboard computer serving as the ROS master. The system runs Ubuntu 20.04 and utilizes a SanDisk 256G memory card, with ROS1 Noetic 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 the LiDAR and the depth camera facilitate environmental perception, enabling autonomous navigation, obstacle avoidance, and SLAM mapping. The onboard computer processes environmental information in ROS1 Noetic, performing path planning, navigation, and obstacle avoidance before transmitting velocity and control data to corresponding ROS topics. The controller then subscribes to these topics and executes robot control tasks.

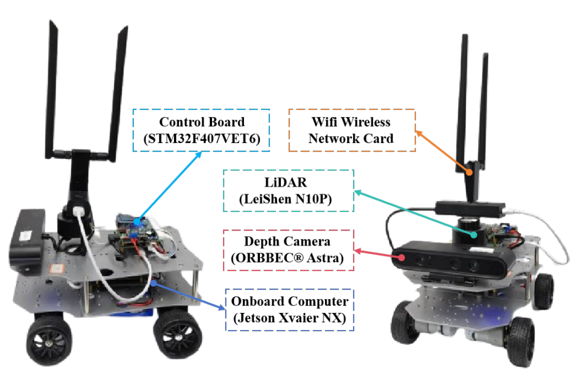


Fig. 6. The detailed composition of the robot platform.

TABLE I
Energy consumption per unit time (Watt) of our robot in different states.

	inference	communication	standby
energy consumption per unit time (W)	13.35	4.25	4.04

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. Notice that different models, due to varying numbers of parameters, exhibit distinct GPU utilization rates and energy consumption during inference.

In our experiments, the GPU server is a PC equipped with an Intel(R) i5 12400f CPU @ 4.40GHz and an NVIDIA GeForce GTX 2080 Ti 11GB GPU, connected to our robot via Wi-Fi 6 over a 160MHz channel at 5GHz frequency. We limited our evaluation to edge computing rather than cloud computing. This decision was based on the observation that the computing power of our PC's GPU is sufficiently robust for the models used in current robotic applications. while more powerful GPUs in the cloud could slightly accelerate the computation process, they would introduce additional latency due to the increased RTT of each RPC necessary to access cloud resources. The main distinction between cloud and edge scenarios lies in this increased communication overhead, which is even more pronounced than in the outdoor scenario. We believe that the robustness of RRTO under varying bandwidth conditions can be effectively demonstrated by comparing indoor and outdoor scenarios, eliminating the need for experiments in cloud settings.

Real-World Robotic Applications. We evaluated a real-time people-tracking robotic application on our robot as depicted in Fig. 7. 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 based on Convolutional Neural Networks, 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 [81] 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. KAPAO continuously performs inference to achieve the fastest possible inference speed using both baselines and RRTO.

To verify the generalization of the performance of RRTO, we also evaluated six common robotic visual models on our robot useful for three categories of real-world robotic applications: 1. ResNet [27] and ConvNext [28] for object classification; 2. FCN [29] and DeepLabv3 [30] for semantic

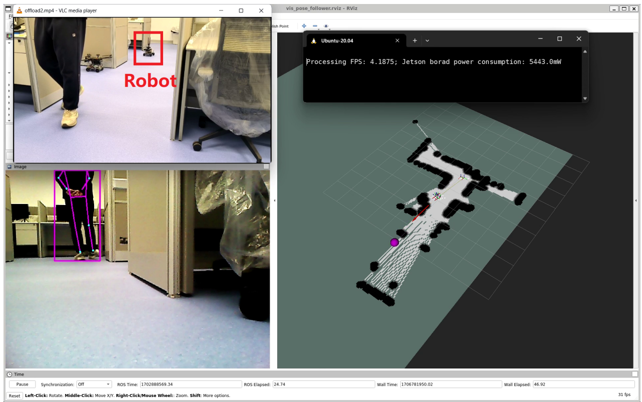


Fig. 7. A screenshot of our real-world experiment. The upper right corner displays real-time FPS and on-board energy consumption, the lower right corner shows the map created by the robot using its LiDAR, the lower left corner features the real-time view from the robot's camera, and the upper left corner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

segmentation; 3. FasterRCNN [31] and RetainNet [32] for object detection. We use the dataset of ImageNet [82] as their inference input and use their implementation from Torchvision [83].

Experiment Environments. We evaluated two real-world environments: indoors (robots move in our laboratory with desks and separators interfering with wireless signals) and outdoors (robots move in our campus garden with trees and bushes interfering with wireless signals, resulting in lower bandwidth). The corresponding bandwidths between the robot and the GPU server in indoors and outdoors scenarios are shown in Fig. 3.

Baselines. we compared RRTO with local computation that inference the entire model locally on the robot (referred to as "local"), the native non-transparent offloading system that offloads all inference computations to GPU servers (referred to as "NNTO"), and a SOTA transparent offloading system (Cricket [22]) under different real-world robotic IoT environments (namely indoors and outdoors). Although advanced scheduling optimizations proven effective in non-transparent offloading systems can be adapted to RRTO, as discussed in Sec. II-B, we have not yet incorporated these methods into RRTO and have restricted our comparisons to NNTO. This is because the performance gains from these optimizations are orthogonal to our innovations, and such optimizations are typically not implemented in transparent offloading systems due to their poor performance associated with the significant communication cost. Consequently, we have excluded these optimizations from our evaluation to ensure a fair comparison and plan to explore their integration into RRTO as part of our future work.

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 sensitive is RRTO to various robotic IoT environments (the bandwidth to the GPU server)?
- RQ3: How does RRTO's record/replay mechanism work?
- RQ4: How does RRTO perform on models common to fundamental robotic tasks on a larger scale?
- RQ5: What are the limitations and potentials of RRTO?

A. End-to-End Performance

Our evaluation results of our robotic application, KAPAO, as presented in Tab. II, demonstrate that RRTO achieved performance comparable to non-transparent offloading system (NNTO) in terms of both inference time and energy consumption on our real-world application, KAPAO.

In terms of inference time, RRTO reduced inference time by an average of 72% compared to local computation and 95% compared to Cricket in the indoors scenario; the reductions in the outdoors scenario were 69% and 94%, respectively. Local computation, which processes the entire model on the robot without any data transmission to a GPU server, exhibits consistent performance in both indoor and outdoor scenarios. In contrast, the substantial communication costs associated with Cricket's frequent RPCs in its transparent offloading mechanism considerably slowed its inference times, a point that will be elaborated on in Sec. VI-B. By implementing its innovative record/replay mechanism, RRTO effectively minimized these extensive communication costs, achieving inference times comparable to those of NNTO, with nearly identical communication expenses.

In terms of energy consumption, RRTO achieved substantial reductions, decreasing energy usage per inference by an average of 85% compared to local computation and 94% compared to Cricket in indoor scenarios; outdoor reductions were 84% and 93%, respectively. Due to the intensive computational demands of the model, local computation incurs high energy consumption per unit time, whereas other systems that offload computations to a GPU server exhibit limited energy usage. Although RRTO only reduced energy consumption per unit time by 45% compared to local computation and even showed a 13% increase in energy consumption per unit time compared to Cricket, its shorter inference times enabled significantly lower energy consumption per inference. It is important to note that the average energy consumption per unit time values listed in Table II do not correspond to those in Table I. This discrepancy arises because our application does not fully utilize the GPU capabilities of the Jetson Xavier NX, resulting in lower average energy consumption during local computation than during the inference stage. Furthermore, additional CPU computing tasks, such as robot control, cause the average energy consumption of all offloading systems to increase above levels observed during communication and standby phases.

The prolonged inference times observed in outdoor scenarios for all offloading systems can be attributed to the lower bandwidth available outdoors (see Sec. II-C), which results in extended transmission times compared to indoor scenarios. By analyzing performance in both indoor and outdoor settings, we found that RRTO is robust across various robotic IoT environments. This robustness stems from RRTO's ability to eliminate the frequent transmission requirements of RPCs in Cricket, thereby achieving communication costs nearly equivalent to those of NNTO.

TABLE II
Inference time (Second), energy consumption per unit time (Watt) and energy consumption per inference (Joule) with standard deviation (±n) of Kapao in different environments with different systems.

Model(number of parameters)	System	Inference time (s)		Energy consumption per unit time (W)		Energy consumption per inference (J)	
		indoors	outdoors	indoors	outdoors	indoors	outdoors
Kapao(77M)	Local	1.42(±0.04)	1.42(±0.04)	10.02(±0.46)	10.02(±0.46)	14.26(±0.4)	14.26(±0.4)
	Cricket	7.37(±0.42)	7.39(±0.33)	4.84(±0.27)	5.05(±0.28)	35.68(±1.95)	37.3(±2.06)
	NNTO	0.38(±0.18)	0.42(±0.16)	5.07(±0.26)	4.89(±0.09)	1.92(±0.1)	2.07(±0.04)
	RRTO	0.40(±0.20)	0.44(±0.19)	5.47(±0.37)	5.27(±0.41)	2.17(±0.15)	2.34(±0.18)

TABLE III
Composition of RPC function calls during different stages of KAPAO inference.

CUDA Runtime API	Composition during loading model	Composition during initializing inference	Composition during the following inference loop
cudaGetDevice	46858 (82.32%)	4789 (80.12%)	4735 (80.32%)
cudaGetLastError	4244 (7.46%)	616 (10.31%)	607 (10.30%)
cudaLaunchKernel	2752 (4.83%)	523 (8.75%)	522 (8.85%)
cudaMalloc	65 (0.11%)	4 (0.07%)	0 (0.00%)
cudaStreamIsCapturing	68 (0.12%)	4 (0.07%)	0 (0.00%)
cudaStreamSynchronize	1118 (1.96%)	16 (0.27%)	11 (0.19%)
cudaMemcpyHtoD	1117 (1.96%)	7 (0.12%)	3 (0.05%)
cudaMemcpyDtoH	1 (0.002%)	9 (0.15%)	8 (0.14%)
cudaMemcpyDtoD	701 (1.23%)	9 (0.15%)	9 (0.15%)

TABLE IV
Semi-RRTO: only applying Caching [56] specifically to the RPCs of cudaGetDevice and cudaGetLastError in RRTO, effectively eliminating their transmission requirements.

Model	System	Inference time (s)		Energy consumption per unit time (W)		Energy consumption per inference (J)	
		indoors	outdoors	indoors	outdoors	indoors	outdoors
Kapao(77M)	Semi-RRTO	1.56(±0.24)	1.99(±2.76)	5.47(±0.37)	5.27(±0.41)	8.54(±0.58)	10.5(±0.81)

B. Micro-Event Analysis

To gain a deeper insight into the performance improvements facilitated by RRTO, we analyzed the RPC function calls made by Cricket during various stages of KAPAO inference, illustrating the characteristics of traditional transparent offloading mechanisms. A detailed breakdown of these calls is provided in Table III.

By comparing the different stages of function calls in Table III, we can see that KAPAO undergoes an initialization stage of inference different from subsequent inference loops. This is because the working process of KAPAO [3] follows the default detection model in Yolo v5 [84]: 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. RRTO recorded all involved operators during the first few inferences, not just the initial process, and ignored the different operator sequences from the initializing inference when the correct operator sequence was found.

In the loop inference detailed in Table III, we observed that a significant portion, specifically 90.62%, of RPC function calls consisted of cudaGetDevice and cudaGetLastError. These calls, generated by PyTorch [23] due to our application's reliance on this framework, are crucial for determining the data's location, facilitating computations across multiple GPUs and parallel tasks. Despite restricting PyTorch to use only a single GPU sequentially and employing Caching (referred to as ”semi-RRTO” in Tab. IV) to reduce RPC functions that require one round-trip time (RTT) to access the GPU server, semi-RRTO achieved inference times comparable only to local computation in our experiments and did not reach the speeds observed with NNTO. This is evident from the fact that cudaLaunchKernel still represents 8.85% of total RPC function calls, which are essential for notifying the server about subsequent computing tasks like additional convolution or maxpool operations. While traditional RPC optimization methods depend on waiting for cudaLaunchKernel RPCs from the client to direct the server's subsequent computing tasks, RRTO records these cudaLaunchKernel function calls and directly executes the subsequent computing tasks on the server, thereby eliminating the need for ongoing communication with the client.

Regarding the remaining RPC functions, namely cudaMalloc, cudaStreamIsCapturing, cudaStreamSynchronize, and cudaMemcpyDtoD, which collectively account for 0.34% 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 0.19% 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 further illustrate the performance gains achieved by

TABLE V
Inference time (Second), energy consumption per unit time (Watt), and energy consumption per inference (Joule) with standard deviation (±n) of torchvision models in different environments with different systems.

Model(number of parameters)	System	Inference time (s)		Energy consumption per unit time (W)		Energy consumption per inference (J)	
		indoors	outdoors	indoors	outdoors	indoors	outdoors
ResNet101(44M)	Local	0.10(±0.02)	0.10(±0.02)	10.61(±0.25)	10.61(±0.25)	1.02(±0.21)	1.02(±0.21)
	Cricket	7.41(±0.37)	7.86(±0.70)	4.98(±0.43)	4.74(±0.23)	36.91(±3.21)	37.26(±1.83)
	NNTO	0.09(±0.09)	0.09(±0.11)	5.32(±0.32)	5.16(±0.3)	0.45(±0.03)	0.47(±0.03)
	RRTO	0.09(±0.11)	0.10(±0.11)	5.05(±0.3)	5.16(±0.37)	0.45(±0.03)	0.5(±0.04)
ConvNext(197M)	Local	0.34(±0.02)	0.34(±0.02)	10.92(±0.45)	10.92(±0.45)	3.69(±0.24)	3.69(±0.24)
	Cricket	4.53(±0.33)	4.78(±0.57)	4.85(±0.22)	4.84(±0.22)	21.96(±0.98)	23.1(±1.04)
	NNTO	0.41(±0.15)	0.42(±0.11)	4.94(±0.27)	4.66(±0.25)	2.04(±0.11)	1.97(±0.11)
	RRTO	0.43(±0.17)	0.44(±0.16)	5.11(±0.3)	5.07(±0.33)	2.22(±0.13)	2.25(±0.14)
FCN(35M)	Local	1.44(±0.33)	1.44(±0.33)	6.0(±2.69)	6.0(±2.69)	8.62(±1.96)	8.62(±1.96)
	Cricket	4.44(±0.28)	4.46(±0.33)	4.84(±0.25)	4.73(±0.25)	21.01(±1.12)	21.56(±1.13)
	NNTO	0.17(±0.12)	0.26(±2.17)	4.97(±0.32)	4.91(±0.41)	0.83(±0.05)	1.27(±0.11)
	RRTO	0.17(±0.14)	0.28(±2.45)	5.04(±0.34)	5.0(±0.31)	0.88(±0.06)	1.4(±0.09)
DeepLabv3(42M)	Local	1.65(±0.49)	1.65(±0.49)	6.01(±2.53)	6.01(±2.53)	9.93(±2.96)	9.93(±2.96)
	Cricket	4.87(±0.31)	4.76(±0.53)	4.8(±0.23)	4.7(±0.26)	23.38(±1.11)	22.39(±1.22)
	NNTO	0.17(±0.10)	0.18(±0.12)	5.03(±0.27)	4.81(±0.29)	0.86(±0.05)	0.88(±0.05)
	RRTO	0.18(±0.13)	0.19(±0.13)	5.0(±0.32)	5.02(±0.34)	0.9(±0.06)	0.97(±0.07)
FasterRCNN(43M)	Local	2.96(±0.51)	2.96(±0.51)	10.96(±0.84)	10.96(±0.84)	32.42(±5.56)	32.42(±5.56)
	Cricket	7.86(±0.33)	7.91(±1.25)	4.7(±0.27)	4.62(±0.28)	36.91(±2.1)	36.57(±2.19)
	NNTO	0.16(±0.91)	0.18(±1.90)	4.93(±0.29)	4.65(±0.22)	0.78(±0.05)	0.85(±0.04)
	RRTO	0.17(±1.00)	0.20(±2.33)	4.66(±0.33)	5.4(±0.8)	0.78(±0.06)	1.06(±0.16)
RetinaNet(38M)	Local	1.59(±0.18)	1.59(±0.18)	10.32(±0.7)	10.32(±0.7)	16.37(±1.9)	16.37(±1.9)
	Cricket	9.99(±0.34)	9.81(±0.42)	4.84(±0.23)	4.7(±0.25)	48.34(±2.31)	46.15(±2.46)
	NNTO	0.15(±0.90)	0.21(±4.06)	4.97(±0.32)	4.67(±0.19)	0.73(±0.05)	0.99(±0.04)
	RRTO	0.15(±1.01)	0.23(±4.73)	5.13(±0.36)	5.04(±0.35)	0.79(±0.06)	1.17(±0.08)

TABLE VI
Comparison between RRTO and the baselines about numbers of RPC calls and average GPU utilization on the edge device

	NNTO	Cricket	RRTO
RPCs for each inference	NA	5895	11
Average GPU utilization on the edge device	29.0%	1.1%	27.5%

RRTO, we compared it with baseline systems in terms of the number of RPC calls and the resulting average GPU utilization on the GPU server during robotic application execution, as measured using pynvml [85] and presented in Table VI. Unlike RRTO and Cricket, NNTO bypasses RPC by directly synchronizing input and output data of the ML model between the CPU and GPU via cudaMemcpyHtoD and cudaMemcpyDtoH at the application layer, requiring modifications to the source code. Cricket, on the other hand, experiences higher communication costs, which contribute to lower GPU utilization on the GPU server. Although RRTO also manages cudaMemcpyDtoH and cudaMemcpyHtoD like Cricket, resulting in 11 RPCs per inference, the performance improvements offered by RRTO are clearly advantageous.

C. Validation on a larger range of models

We conducted a comprehensive evaluation of RRTO and other baseline systems across a diverse set of models commonly used in mobile devices, varying in parameter counts as detailed in Table V. We selected the two most prevalent models for each of three fundamental robotic tasks (object classification, semantic segmentation, and object detection) to assess the generalizability of RRTO's performance. Our findings confirm that RRTO achieves performance comparable to NNTO without necessitating any modifications to the source code. Cricket sometimes exhibits slower performance indoors compared to outdoors, primarily due to its exceptionally prolonged inference times and the unstable network fluctuations as shown in Sec. 3. While RRTO consistently outperforms across various models, we noted that the performance gains are relatively smaller for models with fewer parameters. This observation can be attributed to the fact that models with larger computational demands benefit more significantly from the robust computing power of the GPU server. Additionally, the substantial energy consumption incurred by extensive computations on the robot suggests that models with a larger number of parameters are more suited for offloading, thereby deriving greater benefits from offloading.

D. Lessons Learned

Fixed Calculation Logic. RRTO leverages the characteristic that operators in the inference of a DNN model often follow a fixed order, allowing its record/replay mechanism to support other computational tasks [86], [87], not solely static ML models, provided they exhibit fixed computational logic. However, RRTO is unable to support tasks with unfixed computational logic, such as dynamic ML models with changing operator sequences or tasks involving complex logic and branching, due to its inability to replay varying operator sequences. Moreover, tasks that entail complex logic and branching are generally more suited for CPU rather than GPU execution [88], and optimizing inference for dynamic ML models with changing operator sequences remains a pervasive challenge across all offloading systems (see Sec. III-A).

Adapting existing optimizations from non-transparent offloading systems to RRTO. As discussed in Sec. II-B, advanced scheduling optimizations effective in non-transparent offloading systems are adaptable to RRTO. For example, akin to the layer partitioning in non-transparent systems [13]–[15], RRTO employs an operator partition strategy where certain operators are executed on robots, while others are processed on GPU servers through its communication backend. This methodology not only allows RRTO to leverage the layer partition scheduling algorithm from non-transparent systems to boost its performance but also provides a more granular and flexible offloading schedule, which is particularly beneficial for accommodating fluctuations in wireless network bandwidth within robotic networks. Additionally, by adopting the decision algorithms from multiple inference scheduling, RRTO can efficiently support multiple inference tasks simultaneously, maintaining high performance as detailed in Sec. IV-A.

Future Work. In the future, we aim to apply and evaluate RRTO on a wider range of real-world applications across various robotic platforms, such as unmanned aerial vehicles and legged robots. As the first transparent offloading system to match the high performance of non-transparent systems, RRTO opens new avenues for enhancements, such as developing an edge computing network that offloads computations to other idle robots or edge devices [89] via fine-grained operator partition scheduling to maximize GPU resource utilization. Moreover, due to the limited bandwidth in robotic IoT environments, transmission time significantly impacts performance even in SOTA layer partitioning algorithms [13] (often accounting for nearly half of the inference time on our robots). This highlights transmission still as a bottleneck in robotic IoT and underscores the need for innovative distributed inference methods to address this challenge.

VII. Conclusion

In this paper, we introduce RRTO, a high-performance transparent offloading system optimized for ML model inference on robotic IoT. RRTO addresses the substantial communication costs typically associated with frequent RPCs in traditional transparent offloading systems by implementing a novel record/replay mechanism. This approach achieves comparable high performance to existing non-transparent offloading methods without necessitating any modifications to the source code. We anticipate that RRTO will significantly enhance the deployment of various ML applications on mobile robots in real-world environments. By providing fast and energy-efficient inference capabilities, RRTO enables these robots to perform complex tasks with increased efficiency and effectiveness, all while maintaining a user-friendly interface.

References

[1] K. J. Joseph, S. Khan, F. S. Khan, and V. N. Balasubramanian, "Towards open world object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 5830–5840.

[2] S. Liu, X. Li, H. Lu, and Y. He, "Multi-object tracking meets moving uav," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 8876–8885.

[3] W. McNally, K. Vats, A. Wong, and J. McPhee, "Rethinking keypoint representations: Modeling keypoints and poses as objects for multi-person human pose estimation," *arXiv preprint arXiv:2111.08557*, 2021.

[4] Q. Li, F. Gama, A. Ribeiro, and A. Prorok, "Graph neural networks for decentralized multi-robot path planning," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 11 785–11 792.

[5] B. Wang, Z. Liu, Q. Li, and A. Prorok, "Mobile robot path planning in dynamic environments through globally guided reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6932–6939, 2020.

[6] Y. Yang, L. Juntao, and P. Lingling, "Multi-robot path planning based on a deep reinforcement learning dqn algorithm," *CAAI Transactions on Intelligence Technology*, vol. 5, no. 3, pp. 177–183, 2020.

[7] A.-Q. Cao and R. de Charette, "Monoscene: Monocular 3d semantic scene completion," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 3991–4001.

[8] Y. Li, Z. Yu, C. Choy, C. Xiao, J. M. Alvarez, S. Fidler, C. Feng, and A. 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*, 2023, pp. 9087–9098.

[9] Z. Xia, Y. Liu, X. Li, X. Zhu, Y. Ma, Y. Li, Y. Hou, and Y. Qiao, "Scpnet: Semantic scene completion on point cloud," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 17 642–17 651.

[10] B. Plancher, S. M. Neuman, T. Bourgeat, S. Kuindersma, S. Devadas, and V. J. Reddi, "Accelerating robot dynamics gradients on a cpu, gpu, and fpga," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2335–2342, 2021.

[11] T. Ohkawa, K. Yamashina, H. Kimura, K. Ootsu, and T. Yokota, "Fpga components for integrating fpgas into robot systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 2, pp. 363–375, 2018.

[12] V. Honkote, D. Kurian, S. Muthukumar, D. Ghosh, S. Yada, K. Jain, B. Jackson, I. Klotchkov, M. R. Nimmagadda, S. 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)*. IEEE, 2019, pp. 48–50.

[13] H. Wu, W. J. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1464–1480, 2019.

[14] X. Chen, J. Zhang, B. Lin, Z. Chen, K. Wolter, and G. Min, "Energy-efficient offloading for dnn-based smart iot systems in cloud-edge environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 683–697, 2021.

[15] H. Liang, Q. Sang, C. Hu, D. Cheng, X. Zhou, D. Wang, W. Bao, and Y. Wang, "Dnn surgery: Accelerating dnn inference on the edge through layer partitioning," *IEEE transactions on Cloud Computing*, 2023.

[16] D. Zhang, N. Vance, Y. Zhang, M. T. Rashid, and D. Wang, "Edge-batch: Towards ai-empowered optimal task batching in intelligent edge systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 366–379.

[17] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1584–1607, 2019.

[18] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[19] M. Altamimi, A. Abdrabou, K. Naik, and A. Nayak, "Energy cost models of smartphones for task offloading to the cloud," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 3, pp. 384–398, 2015.

[20] K. Elgazzar, P. Martin, and H. S. Hassanein, "Cloud-assisted computation offloading to support mobile services," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 279–292, 2014.

[21] Z. Fang, T. Yu, O. J. Mengshoel, and R. 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*, 2017, pp. 2067–2070.

[22] N. Eiling, J. Baude, S. Lankes, and A. Monti, "Cricket: A virtualization layer for distributed execution of cuda applications with checkpoint/restart support," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 14, 2022.

[23] pytorch, "pytorch," https://www.pytorch.org, 2024.

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

[25] M. Schneider, F. Haag, A. K. Khalil, and D. A. Breunig, "Evaluation of communication technologies for distributed industrial control systems: Concept and evaluation of 5g and wifi 6," *Procedia CIRP*, vol.

107, pp. 588–593, 2022, leading manufacturing systems transformation – Proceedings of the 55th CIRP Conference on Manufacturing Systems 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827122003146

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

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[28] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," 2022. [Online]. Available: https://arxiv.org/abs/2201.03545

[29] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," 2015. [Online]. Available: https://arxiv.org/abs/1411.4038

[30] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," 2017. [Online]. Available: https://arxiv.org/abs/1706.05587

[31] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016. [Online]. Available: https://arxiv.org/abs/1506.01497

[32] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," 2018. [Online]. Available: https://arxiv.org/abs/1708.02002

[33] G. Giunta, R. Montella, G. Agrillo, and G. 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*. Springer, 2010, pp. 379–391.

[34] Y.-S. Lin, C.-Y. Lin, C.-R. Lee, and Y.-C. Chung, "qcuda: Gpgpu virtualization for high bandwidth efficiency," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2019, pp. 95–102.

[35] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1423–1431.

[36] R. Liu and N. Choi, "A first look at wi-fi 6 in action: Throughput, latency, energy efficiency, and security," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 1, pp. 1–25, 2023.

[37] X. Yang, H. Lin, Z. Li, F. Qian, X. Li, Z. He, X. Wu, X. Wang, Y. Liu, Z. Liao *et al.*, "Mobile access bandwidth in practice: Measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 114–128.

[38] A. Masiukiewicz, "Throughput comparison between the new hew 802.11 ax standard and 802.11 n/ac standards in selected distance windows," *International Journal of Electronics and Telecommunications*, vol. 65, no. 1, pp. 79–84, 2019.

[39] Y. Pei, M. W. Mutka, and N. Xi, "Connectivity and bandwidth-aware real-time exploration in mobile robot networks," *Wireless Communications and Mobile Computing*, vol. 13, no. 9, pp. 847–863, 2013.

[40] M. Ding, P. Wang, D. López-Pérez, G. Mao, and Z. Lin, "Performance impact of los and nlos transmissions in dense cellular networks," *IEEE Transactions on Wireless Communications*, vol. 15, no. 3, pp. 2365–2380, 2015.

[41] N. I. Sarkar and O. Mussa, "The effect of people movement on wi-fi link throughput in indoor propagation environments," in *IEEE 2013 Tencon-Spring*. IEEE, 2013, pp. 562–566.

[42] T. Adame, M. Carrascosa-Zamacois, and B. Bellalta, "Time-sensitive networking in ieee 802.11 be: On the way to low-latency wifi 7," *Sensors*, vol. 21, no. 15, p. 4954, 2021.

[43] Y. Ren, C.-W. Tung, J.-C. Chen, and F. Y. Li, "Proportional and preemption-enabled traffic offloading for ip flow mobility: Algorithms and performance evaluation," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 12, pp. 12 095–12 108, 2018.

[44] "iPerf - Download iPerf3 and original iPerf pre-compiled binaries." [Online]. Available: https://iperf.fr/iperf-download.php

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

[46] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[47] A. Défossez, Y. Adi, and G. Synnaeve, "Differentiable model compression via pseudo quantization noise," *arXiv preprint arXiv:2104.09987*, 2021.

[48] t. Gheorghe and M. Ivanovici, "Model-based weight quantization for convolutional neural network compression," in *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE, 2021, pp. 1–4.

[49] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "Vecq: Minimal loss dnn model compression with vectorized weight quantization," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 696–710, 2020.

[50] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 2021.

[51] T. Lin, L. Kong, S. U. Stich, and M. Jaggi, "Ensemble distillation for robust model fusion in federated learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 2351–2363, 2020.

[52] L. Wang and K.-J. Yoon, "Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 6, pp. 3048–3068, 2021.

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

[54] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. 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*, 2021, pp. 36–51.

[55] S. Eyerman and I. Hur, "Efficient asynchronous rpc calls for microservices: Deathstarbench study," *arXiv preprint arXiv:2209.13265*, 2022.

[56] A. Singhvi, A. Akella, M. Anderson, R. Cauble, H. Deshmukh, D. Gibson, M. M. Martin, A. Strominger, T. F. Wenisch, and A. Vahdat, "Cliquemap: Productionizing an rma-based distributed caching system," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 93–105.

[57] R. Kruse, S. Mostaghim, C. Borgelt, C. Braune, and M. Steinbrecher, "Multi-layer perceptrons," in *Computational intelligence: a methodological introduction*. Springer, 2022, pp. 53–124.

[58] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 12, pp. 6999–7019, 2021.

[59] J. Koutnik, K. Greff, F. Gomez, and J. Schmidhuber, "A clockwork rnn," in *International conference on machine learning*. PMLR, 2014, pp. 1863–1871.

[60] Y. Wang, L. Wang, F. Yang, W. Di, and Q. Chang, "Advantages of direct input-to-output connections in neural networks: The elman network for stock index forecasting," *Information Sciences*, vol. 547, pp. 1066–1079, 2021.

[61] D. Meyer, "Introduction to autoencoders," 2015.

[62] D. P. Kingma, M. Welling *et al.*, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[63] Z. Pan, W. Yu, X. Yi, A. Khan, F. Yuan, and Y. Zheng, "Recent progress on generative adversarial networks (gans): A survey," *IEEE access*, vol. 7, pp. 36 322–36 333, 2019.

[64] F. Gao, Y. Yang, J. Wang, J. Sun, E. Yang, and H. Zhou, "A deep convolutional generative adversarial networks (dcgans)-based semi-supervised method for object recognition in synthetic aperture radar (sar) images," *Remote Sensing*, vol. 10, no. 6, p. 846, 2018.

[65] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[66] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008.

[67] I. Steinwart and A. Christmann, *Support vector machines*. Springer Science & Business Media, 2008.

[68] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[69] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[70] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[71] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE, 2017, pp. 1597–1600.

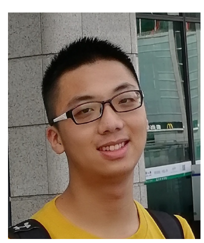
[72] S. Masoudnia and R. Ebrahimpour, "Mixture of experts: a literature survey," *Artificial Intelligence Review*, vol. 42, pp. 275–293, 2014.

[73] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, “A comprehensive survey of neural architecture search: Challenges and solutions,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–34, 2021.
[74] L.-E. Montoya-Cavero, R. D. de León Torres, A. Gómez-Espinosa, and J. A. E. Cabello, “Vision systems for harvesting robots: Produce detection and localization,” *Computers and electronics in agriculture*, vol. 192, p. 106562, 2022.
[75] S. Wan and S. Goudos, “Faster r-cnn for multi-class fruit detection using a robotic vision system,” *Computer Networks*, vol. 168, p. 107036, 2020.
[76] S. M. J. Jalali, S. Ahmadian, A. Khosravi, S. Mirjalili, M. R. Mahmoudi, and S. Nahavandi, “Neuroevolution-based autonomous robot navigation: A comparative study,” *Cognitive Systems Research*, vol. 62, pp. 35–43, 2020.
[77] Z. Xie, L. Jin, X. Luo, Z. Sun, and M. Liu, “Rnn for repetitive motion generation of redundant robot manipulators: An orthogonal projection-based scheme,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 615–628, 2020.
[78] F. Piltan, A. E. Prosvirin, M. Sohaib, B. Saldivar, and J.-M. Kim, “An svm-based neural adaptive variable structure observer for fault diagnosis and fault-tolerant control of a robot manipulator,” *Applied Sciences*, vol. 10, no. 4, p. 1344, 2020.
[79] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, “Efficient algorithms for device placement of dnn graph operators,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 451–15 463, 2020.
[80] Y. He, J. Fang, F. R. Yu, and V. C. Leung, “Large language models (llms) inference offloading and resource allocation in cloud-edge computing: An active inference approach,” *IEEE Transactions on Mobile Computing*, 2024.
[81] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.
[82] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
[83] S. Marcel and Y. Rodriguez, “Torchvision the machine-vision package of torch,” in *Proceedings of the 18th ACM International Conference on Multimedia*, ser. MM ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1485–1488. [Online]. Available: https://doi.org/10.1145/1873951.1874254
[84] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, K. Michael, TaoXie, J. Fang, imyhxy, Lorna, Z. Yifu, C. Wong, A. V, D. Montes, Z. Wang, C. Fati, J. Nadar, Laughing, UnglvKitDe, V. Sonck, tkianai, yxNONG, P. Skalski, A. Hogan, D. Nair, M. Strobel, and M. Jain, “ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation,” Nov. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7347926
[85] N. Corporation, “pynvml: Python Bindings for the NVIDIA Management Library.” [Online]. Available: http://www.nvidia.com/
[86] N. Cautaerts, P. Crout, H. W. Ånes, E. Prestat, J. Jeong, G. Dehm, and C. 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*, vol. 237, p. 113517, 2022.
[87] R. Chowdhury and D. Subramani, “Optimal path planning of autonomous marine vehicles in stochastic dynamic ocean flows using a gpu-accelerated algorithm,” *IEEE Journal of Oceanic Engineering*, vol. 47, no. 4, pp. 864–879, 2022.
[88] V. Rosenfeld, S. Breß, and V. Markl, “Query processing on heterogeneous cpu/gpu systems,” *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–38, 2022.
[89] Y. Bao, Y. Peng, C. Wu, and Z. Li, “Online job scheduling in distributed machine learning clusters,” in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 495–503.

Zekai Sun is currently working toward a PhD degree in the Systems and Networks Laboratory at the University of Hong Kong under the guidance of Prof.Heming Cui. He received Bachelor’s Degree at University of Science and Technology of China (USTC) in 2020. His primary research areas include distributed systems, edge computing and systems for machine learning.

Xiuxian Guan is currently a HKU-SusTech Joint PhD student (2021Fall–Now) in Department of Computer Science, HKU and Department of Electrical & Electronic Engineering, SusTech. He received Bachelor’s Degree in Department of Computer Science and Technology in University of Science and Technology of China (USTC) in 2020. His research interest includes distributed systems, edge computing and systems for machine learning.

Junming Wang is currently pursuing his MPhil in the Systems and Networks Laboratory at the University of Hong Kong under the guidance of Prof. Heming Cui. He received his Bachelor’s degree at Lanzhou Jiaotong University. His primary research areas include embodied intelligence, autonomous navigation, and 3D computer vision.

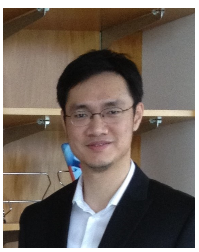


Yuhao Qing is currently a Ph.D. student in the department of computer science at the University of Hong Kong, under the supervision of Prof. Heming Cui. He received his bachelor’s degree from the City University of Hong Kong. His research interests include machine learning systems and cloud computing.

Haoze Song is a Ph.D. student in the department of computer science at the University of Hong Kong. He received his bachelor’s degree in Computer Science from the University of Science and Technology of China with honors. His current research interests are in the areas of distributed computing, data management in cloud computing environments, and scalable real-time systems.

Dong Huang is a Ph.D. student in the department of computer science at the University of Hong Kong. He received his bachelor's degree in Material Processing and Control Engineering from the Huazhong University of Science and Technology with honors. His current research interests are broadly in code generation such as correctness, efficiency, and fairness.

Fangming Liu received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor at the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, data center and green computing, SDN/NFV/5G, and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of the Ministry of Education in China, as well as the Second Class Prize of National Natural Science Award in China.



Heming Cui is an Associate Professor in Computer Science of HKU. His research interests include operating systems, programming languages, distributed systems, and cloud computing, with a particular focus on building software infrastructures and tools to improve reliability and security of real-world software. He is a member of IEEE.