RoSÉ: A Hardware-Software Co-Simulation Infrastructure Enabling Pre-Silicon Full-Stack <u>Ro</u>botics <u>S</u>oC <u>E</u>valuation

Dima Nikiforov
vnikiforov@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Shengjun Kris Dong
krisdong@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Chengyi Lux Zhang
iansseijelly@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Seah Kim
seah@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Borivoje Nikolić
bora@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Yakun Sophia Shao
ysshao@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

ABSTRACT

Robotic systems, such as autonomous unmanned aerial vehicles (UAVs) and self-driving cars, have been widely deployed in many scenarios and have the potential to revolutionize the future generation of computing. To improve the performance and energy efficiency of robotic platforms, significant research efforts are being devoted to developing hardware accelerators for workloads that form bottlenecks in the robotics software pipeline. Although domain-specific accelerators can offer improved efficiency over general-purpose processors on isolated robotics benchmarks, system-level constraints such as data movement and contention over shared resources can significantly impact the achievable end-to-end acceleration. In addition, the closed-loop nature of robotic systems, where there is a tight interaction across different deployed environments, software stacks, and hardware architecture, further exacerbates the difficulties of evaluating robotics SoCs.

To address this limitation, we develop RoSÉ, an open-source, hardware-software co-simulation infrastructure for full-stack, pre-silicon hardware-in-the-loop evaluation of robotics SoCs, together with the full software stack and realistic environments created to support robotics workloads. RoSÉ captures the complex interactions across hardware, algorithm, and environment, enabling new architectural research directions in hardware-software co-design for robotic systems.

KEYWORDS

Robotics, Simulation, Hardware-Software Co-Design

ACM Reference Format:
Dima Nikiforov, Shengjun Kris Dong, Chengyi Lux Zhang, Seah Kim, Borivoje Nikolić, and Yakun Sophia Shao. 2023. RoSÉ: A Hardware-Software Co-Simulation Infrastructure Enabling Pre-Silicon Full-Stack <u>Ro</u>botics <u>S</u>oC <u>E</u>valuation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589099

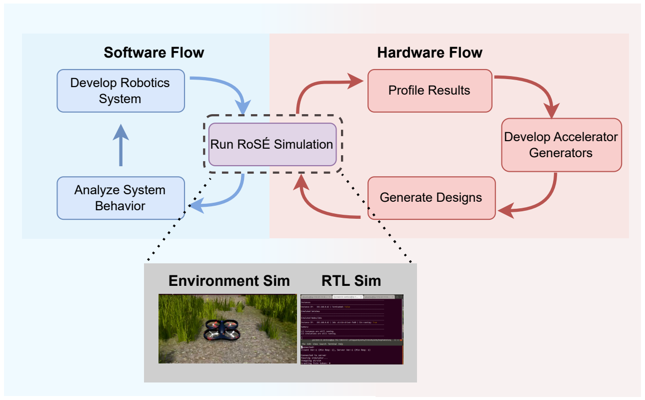


Figure 1: RoSÉ provides an integrated co-simulation environment for both hardware and software, enabling pre-silicon hardware-in-the-loop evaluation of robotics SoCs.

1 INTRODUCTION

A fruit fly can compute workloads that include trajectory planning, visual/inertial odometry (VIO), classification, and closed-loop control, all while only consuming 120 nW [50]. State-of-the-art VIO hardware consumes 2 mW when executing a standard benchmark [54]—over 10,000× more power. Such a stark difference suggests vast optimization opportunities for autonomous systems. However, closing the gap becomes increasingly challenging; with the slowdown in technology scaling, it is no longer feasible to rely on improvements in process technology and general-purpose processors to improve power efficiency and performance [25].

As a result, over the past decade, there has been a proliferation of academic research groups, startups, and industrial R&D labs that develop domain-specific accelerators (DSAs) to eke out remaining performance improvements [17, 37–39, 43, 54, 56]. Although DSAs deliver improved performance and energy efficiency for applications of interest, today's accelerators are largely evaluated in isolation, without considering the end-to-end performance of the entire robotics pipeline. This is particularly important given that robotics workloads often consist of a range of different algorithms running concurrently, where the performance of a particular accelerator is highly *system-dependent* and *data-dependent*.

Specifically, on the system-dependent front, the performance of each individual accelerator can be heavily impacted by system-level resource contentions where multiple general-purpose cores and accelerators are running together [29]. On the data-dependent front, taking a flying drone as an example, while higher velocity can make the drone move faster, it can also negatively impact the accuracy of pose estimation, leading to suboptimal trajectory planning [19]. The system- and data-dependent nature of closed-loop algorithms in robotics makes it infeasible to use traditional trace-driven simulation frameworks for end-to-end performance evaluation. Recently, the robotics community has adopted hardware-in-the-loop simulation to assess the performance of robotics algorithms on specific hardware [13], though it only works with off-the-shelf components, without support for pre-silicon evaluation of hardware that has not been fabricated yet.

To address the limitations, we develop RoSÉ, a hardware-software co-simulation infrastructure to enable full-stack, pre-silicon, hardware-in-the-loop evaluation of robotics SoCs, with a particular emphasis on unmanned aerial vehicles (UAVs). Specifically, RoSÉ integrates AirSim, a robotics environment simulator [51], supporting UAV models, and FireSim, an FPGA-based RTL simulator [27], to capture closed-loop interactions across environments, algorithms, and hardware, as shown in Figure 1. Traditionally, robotics software and hardware are evaluated in two separate simulation environments, where the software is evaluated using off-the-shelf hardware while the hardware is evaluated in RTL simulation using pre-recorded data traces. Such a decoupled evaluation flow will not capture the subtle interaction between hardware and software development, especially in the context of hardware-software co-design for closed-loop applications like robotics. RoSÉ addresses this challenge by providing an integrated co-simulation environment for both hardware and software to enable pre-silicon, hardware-in-the-loop evaluation for robotic SoCs. Our evaluation demonstrates that RoSÉ holistically captures the closed-loop interactions between environment, algorithms, and hardware, opening up new opportunities for systematic hardware-software co-design for robotic UAV systems.

In summary, this paper makes the following contributions:

(1) We build RoSÉ, a hardware-software co-simulation infrastructure for pre-silicon, full-stack evaluation of robotic UAV SoCs. RoSÉ captures the dynamic interactions between robotics hardware, software and environment[1].
(2) We develop software algorithms and hardware SoCs in RoSÉ to holistically evaluate the hardware-software co-design in robotic systems.
(3) We demonstrate that RoSÉ enables large design space exploration of robotics SoCs and identifies important hardware-software design trade-offs.

2 BACKGROUND AND MOTIVATION

This section discusses the complexity of robotics applications, the state-of-the-art hardware-in-the-loop evaluation methodology in robotics, and opportunities for co-simulating both hardware and software in robotic systems.

[1]RoSÉ is open-source and publicly available at https://github.com/ucb-bar/RoSE.

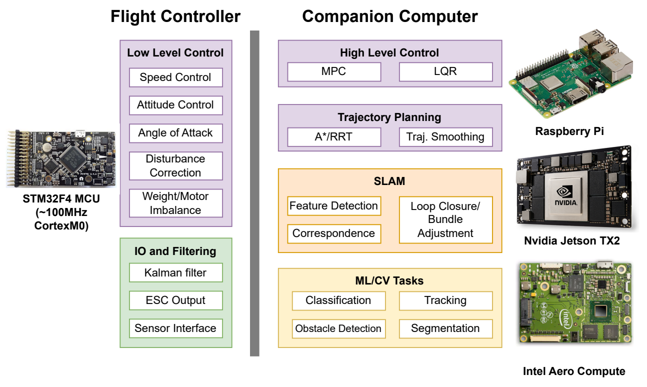


Figure 2: Typical partitioning of tasks that are performed on a UAV. The flight controller is typically a microcontroller running baremetal code or an RTOS, while the companion computer can have diverse architectures, including mobile CPUs, GPUs, or domain-specific SoCs.

2.1 Application Complexity in Robotics

Robotics is a broad field spanning varying deployment scenarios, each having its own unique target applications and constraints. In this work, we focus on UAVs as they typically have extremely tight weight, energy, and latency constraints. Naturally, UAVs have strict performance requirements to avoid collisions and quickly react to external events during flight. At the same time, to maintain long flight times, UAVs also need to satisfy battery and weight constraints, limiting the amount of compute that can be hosted onboard the UAV. These considerations lead to a diversity of deployment scenarios for compute resources in UAV applications, ranging from onboard companion computers [24], edge computers [35], and even the cloud [22].

Figure 2 illustrates a typical deployment scenario in which all computation is performed onboard the UAV. Specifically, robotics tasks are partitioned between a flight controller (typically implemented using a microcontroller) and a companion computer (typically an embedded SoC) [24]. The flight controller performs hard real-time tasks, such as low-level control, and is a well-established design point, where the STM32F4 microcontroller, running baremetal code or an RTOS such as FreeRTOS, is widely used across UAV platforms. However, the companion computer runs a diversity of algorithms, including simultaneous localization and mapping (SLAM) pipeline for UAVs to infer their position, and perception tasks using machine learning (ML) and computer vision (CV) algorithms, high-level control algorithms that control the behaviors of UAVs, trajectory planning algorithms to produce a collision-free path, and the low-level model predictive control (MPC). These algorithms can differ significantly by application and can be implemented using a range of hardware architectures, including CPUs, GPUs, and domain-specific SoCs [44, 46, 54].

2.2 Hardware-in-the-loop Evaluation of Robotics SoCs

Recognizing the complexity of closed-loop interaction between hardware and software in robotic systems, the robotics community has adopted a hardware-in-the-loop evaluation flow in which hardware platforms are integrated with robotic environment simulators to capture the interaction between hardware and software [8, 9, 21]. Recent efforts in the architecture community also highlight the importance of hardware-in-the-loop evaluation. For example, MAVBench [13] provides a benchmark suite of UAV workloads in a hardware-in-the-loop evaluation. However, MAVBench only works with off-the-shelf hardware, without support for pre-silicon, RTL-level evaluation of hardware that has not been fabricated yet. This restriction limits users to tuning post-silicon system parameters such as core count and clock frequency, without access to a wider range of microarchitectural parameters across accelerator design and SoC integration.

2.3 Hardware-Software Co-Simulation for Robotics

Computer architects have long been developing simulation infrastructures for general-purpose cores [12, 16], GPUs [28, 36], and domain-specific accelerators [33, 45, 48, 49, 52]. In addition, to better understand the behaviors of emerging applications, the architecture community has also developed software infrastructure for workload characterization, e.g., MAVBench [13] for autonomous UAVs and ILLIXR [26] for virtual and augmented reality. However, these frameworks are designed for either hardware or software, without considering the complex interactions between hardware and software in an end-to-end fashion. This is especially important for closed-loop applications like UAVs and self-driving cars, where different software and hardware components are tightly coupled with each other. For example, in a UAV, while a high sampling rate in sensors generally improves the accuracy of the perception and localization algorithms, it can also lead to more data transfer between different stages of the processing pipeline, resulting in a longer overall task time. Such complex interactions across sensing, environments, algorithms, and hardware cannot be accurately captured using traditional hardware- or software-only evaluation flows. RoSÉ is, to the best of our knowledge, the first to support end-to-end evaluation of domain-specific systems through hardware-software co-simulation for closed-loop robotics systems.

3 ROSÉ DESIGN

We present RoSÉ, a hardware-software co-simulation infrastructure that enables robotics developers to evaluate the end-to-end performance of robotics hardware and software in closed-loop environments. The infrastructure contains three key components, as illustrated in Figure 3. First, RoSÉ is built on top of AirSim [51] to generate environments in which to run robotic UAV simulations. Second, to simulate hardware SoC designs, RoSÉ uses FireSim, an FPGA-accelerated RTL simulator, to evaluate the pre-silicon performance of robotic hardware. In addition, RoSÉ also includes a build flow for developing robotics applications for RISC-V SoCs. The flow includes support for application development, runtime environments, and tooling to generate RISC-V images. In particular, RoSÉ integrates AirSim and FireSim together with synchronized execution to accurately capture the hardware-software interactions of robotic UAV systems. This section discusses how RoSÉ supports environment simulation, hardware simulation, software build flow, and co-simulation integration.

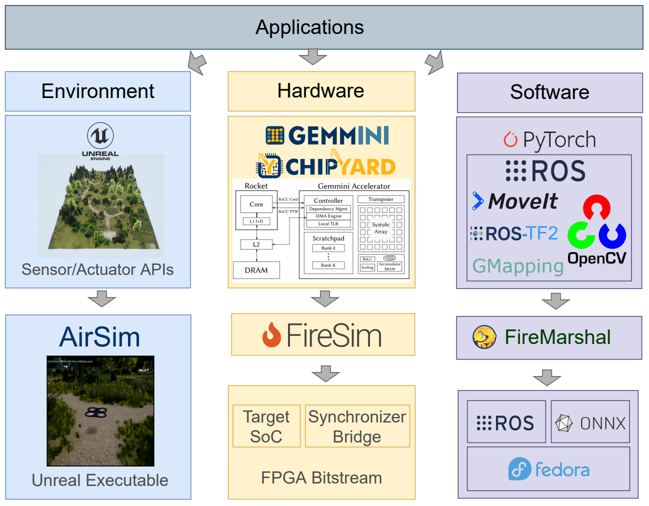


Figure 3: Overview of the RoSÉ hardware-software co-simulation infrastructure for robotics.

Robotics Environment Simulators			
Simulator Name	Domain	Simulation Mode	Interface
AirSim [51]	AV/UAV	Realtime	RPC/ROS
Gazebo [30]	General Purpose	Offline	ROS
CARLA [20]	AV	Realtime	ROS
MuJoCo [55]	Rigid Body	Offline	ROS
PyBullet [18]	Rigid Body	Realtime/Offline	ROS

Table 1: Robotics environment simulators.

3.1 Environment Simulation

To simulate an end-to-end robotics application, it is necessary to simulate the robot's environment to generate sensor data and to model the effects of actuation. Within the robotics community, there are a wide variety of robotics environment simulators that target different domains and use cases. Table 1 summarizes some of the most widely used robotics environment simulators.

In this work, to focus our evaluation on UAV applications, RoSÉ supports AirSim [51], a widely-used open-source environment simulator originally developed for real-time simulation and evaluation of machine learning and reinforcement learning applications for UAVs and autonomous vehicles. AirSim is a plugin for Unreal Engine and uses Unreal's graphics engine for rendering camera data and Unreal's physics engine to handle collisions while using its own internal physics models and inertial sensor models.

AirSim provides two APIs to interact with the simulator. First, it has a remote-procedure-call (RPC) API for sensor readings and actuation, as well as simulator commands (such as controlling simulator execution or changing time of day). Second, it exposes ROS topics and services as wrappers for API calls. RoSÉ uses the RPC API to send and receive AirSim data and serializes the data to transmit to the I/O space of the modeled SoC. To integrate with other simulators, the environment simulator needs to support discrete time-stepping. These interfaces make it feasible to integrate

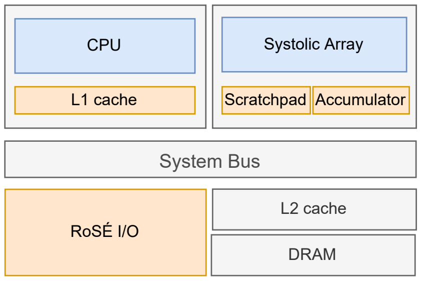


Figure 4: Example SoC integrating the RoSÉ IO module onto the system bus.

environment simulators such as AirSim with the hardware RTL simulation flow.

3.2 Hardware Simulation

In addition to the environment simulator, we also need to simulate the SoC that is being evaluated. SoCs can be simulated at various levels, including at the ISA level using tools like Spike [6], at the architecture level using pre-RTL simulators such as gem5 [12], or at the RTL level with RTL-based simulators like VCS or Verilator. Tradeoffs between SoC simulators include simulation fidelity and simulation throughput. The highest fidelity simulation occurs in RTL simulation, but software RTL simulation is prohibitively slow for evaluating the system-level behavior of robotics applications: Simulating a 3-minute flight of a UAV would require more than a year of simulation time by using RTL simulation, which runs at several KHz on a high-end compute server. To mitigate this cost, RoSÉ uses hardware-accelerated RTL simulation. Multiple commercial solutions exist for accelerated RTL emulation, such as Synopsys' ZeBu or Cadence's Palladium. However, such setups are typically expensive for smaller research teams. To achieve cycle-accurate accelerated RTL simulation, RoSÉ uses FireSim [27], an open-source platform with FPGA-accelerated RTL simulation.

To support deterministic integration of external models (such as for memories and IO devices), FireSim provides an interface for writing bridges, which enable arbitrating and synchronizing between software tasks running on the simulation host, and registers on the FPGA simulation target. Bridges consist of RTL wrappers around modules in the simulated SoC, as well as bridge driver software that runs on the host and interfaces with registers in the bridges.

RoSÉ builds on top of the FireSim infrastructure with the RoSÉ Bridge, which synchronously models I/O between a companion computer and a flight controller. The RoSÉ Bridge is exposed to the target SoC as memory-mapped I/O registers on the system bus, as depicted in Figure 4. The bridge itself consists of hardware queues that buffer data being sent to and from the SoC, as well as a control unit that can throttle the execution of the RTL simulation. Section 3.4 details the implementation of the RoSÉ Bridge.

3.3 Software Build Flow

To build applications that will be evaluated on robotics SoCs, RoSÉ provides a software build flow that targets RISC-V SoCs. There are two supported flows, one for generating DNN-based workloads and one for classical control workloads. To build DNN-based workloads, the infrastructure uses PyTorch and exports trained models in ONNX format [3]. The ONNX models can then be executed using ONNX-Runtime [4] either directly on CPUs or systolic-array-based matrix accelerators like Gemmini [23].

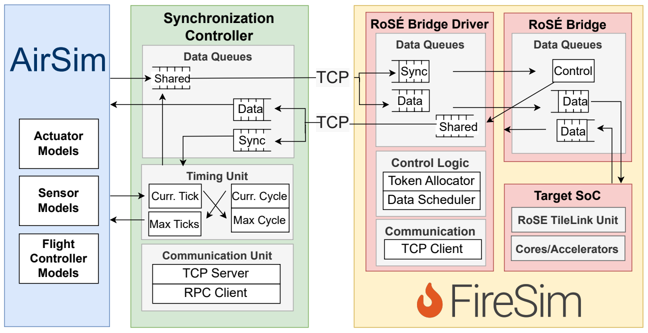


Figure 5: Functional block diagram of RoSÉ infrastructure depicting both synchronizer and RoSÉ Bridge designs. The synchronization controller and the RoSÉ bridge driver use software queues to schedule and buffer packets between synchronization steps, and RoSÉ Bridge contains hardware queues to stage packets being transmitted over the modeled IO interface, exposed as memory-mapped registers on the SoC's system bus.

To support conventional robotics applications, the infrastructure provides a port of the Robot Operating System (ROS) [47] for RISC-V. RoSÉ provides ROS Noetic Ninjemys, and is built from source for Fedora 32. Both the `roscpp` and `rospy` interfaces are supported. Other commonly used libraries such as `OpenCV` [15], `tf2` [7], `GMapping` [5], `Hector-Mapping` [31], `MoveIt` [2] and other ROS libraries are also supported.

3.4 Co-Simulation Architecture

RoSÉ executes a robotics UAV simulation by co-simulating an AirSim environment with a FireSim-accelerated RTL simulation of an SoC. An architectural diagram of the simulator is depicted in Figure 5. To accurately simulate a trajectory of a robot, the simulator must support synchronized execution between simulators, and present a clear abstraction between the simulator infrastructure and the modeled world, enabling realistic models for sensing and actuation.

3.4.1 Synchronization. The two simulators must adhere to the same simulation time. Although the simulators execute at different rates in real-time, events occurring in one environment must be observed at corresponding simulation times by the other simulator. Both AirSim and FireSim step through time by using discrete time steps. In AirSim, the minimum time period is a single frame, which corresponds to a physics and rendering step. The amount of simulation time corresponding to a frame can be defined at runtime, but a typical rate is 60-120Hz. On the other hand, as a digital RTL simulation, the minimum unit of time is a clock cycle. Given the disparity of the minimum time interval, typically many clock ticks correspond to one AirSim frame.

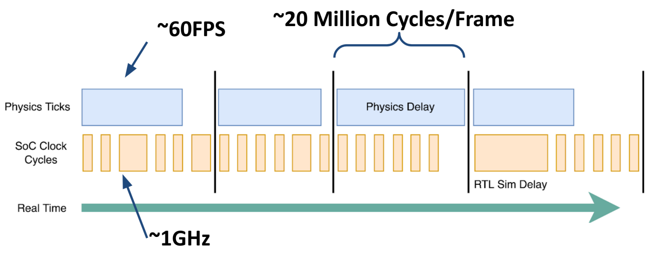


Figure 6: A representation of the lockstep synchronization implemented in RoSÉ. As an example of a typical configuration, if modeling a 1GHz SoC and updating AirSim 60 frames per simulated second, synchronization occurs every 16 million cycles.

RoSÉ implements a lockstep synchronization method, as depicted in Figure 6. A synchronization period is defined between both simulators in terms of AirSim frames and SoC clock cycles. The ratio between the two is defined in Equation 1, and is therefore constrained by the target frequency of the SoC, a property of the physical SoC being designed, and by the frequency of AirSim's physics updates, which are a tunable parameter.

$$\frac{\text{airsim\_steps}}{\text{firesim\_steps}} = \frac{\text{soc\_clock\_freq}}{\text{airsim\_frame\_freq}} \quad (1)$$

The main synchronization loop of RoSÉ is depicted in Algorithm 1. The synchronizer runs as a separate process, communicating with the AirSim server by using its RPC interface, and with FireSim by using a TCP listener.

Communication with FireSim. TCP Packets are used to transmit serialized synchronization and data packets. Packets consist of a header, containing the packet type and number of bytes, as well as a payload containing the serialized contents of the message. RoSÉ transmits both synchronization packets and data packets. Synchronization packets are used to communicate information about the simulation state, such as the number of cycles FireSim can advance every synchronization, and communicate with RoSÉ Bridge but not the modeled SoC. Data packets encode sensor and actuator data. Data packets are the only packets that are visible to the simulated SoC and are accessible through queues pointed to by memory-mapped registers on the system bus, as shown in the FireSim side of Figure 5.

3.4.2 Simulation Abstraction. The simulated SoC must be oblivious to the fact that it is in a simulated environment. This means that the SoC receives sensor data and performs actuation as it would in a deployed environment by communicating through I/O devices. The SoC does not have access to any simulation-level APIs, and runs the same application as it would when deployed to facilitate sim-to-real transfer. To achieve this, as shown in Algorithm 1, the synchronizer receives all data communication from FireSim as serialized packets which must then be translated into API calls within AirSim. As an example, if the SoC requests IMU data it sends an `IMU_REQ` packet over the RoSÉ I/O, which gets transmitted through queues in the RoSÉ Bridge as depicted in Figure 5. The synchronizer receives the packet, decodes it, and then makes an IMU request over RPC to

Algorithm 1 Synchronization Loop

```
t ← 0
▷ % Write clock cycles per sync to RoSÉ Bridge HW%
set_firesim_steps(firesim_steps)
while t < max_time do

    ▷ % Poll simulators for new data, checking if
    ▷ FireSim communicated over IO, or if AirSim
    ▷ sent sensor data %
    env_data ← req_airsim_data()
    rtl_data ← req_rtl_data()
    ▷ % Encode AirSim data to packets sent to FireSim %
    if env_data then
        serialized_data ← encode(env_data)
        transmit_to_rtl(serialized_data)
    end if
    ▷ % Translate IO packets into AirSim APIs %
    if rtl_data then
        for datum in rtl_data do
            cmd ← decode(datum)
            call_airsim_api(cmd)
        end for
    end if
    ▷ % Allocate tokens to start AirSim and FireSim %
    allocate_airsim_frames()
    allocate_rtl_frames()
    airsim_running ← True
    rtl_running ← True
    ▷ % Poll simulators until both finish,
    ▷  and advance by one synchronization step %
    while airsim_running and rtl_running do
        airsim_running ← poll_airsim_state()
        rtl_running ← poll_rtl_state()
    end while
    t ← t + sync_period
end while
```

AirSim. Finally, the IMU data is encoded as a packet and transmitted back over the SoC's I/O.

Additionally, commands sent to/from the SoC must model realistic interfaces. In a typical UAV system, the companion computer does not directly interface with motors and low-level sensors, and instead communicates through a flight controller, enabling the use of intermediate-level controls such as position or velocity targets. However, to ensure accurate modeling of the system it is important to describe and partition the hardware and software being modeled in RoSÉ, which is described in the following sections.

4 EVALUATION METHODOLOGY

This section describes the UAV system we modeled and the setup of hardware, software, and environment used to evaluate RoSÉ.

4.1 Modeled System

RoSÉ models a full UAV system, including sensing, actuation, and compute. A high-level block diagram of a UAV hardware platform is depicted in Figure 7. The sensor models directly use measurement

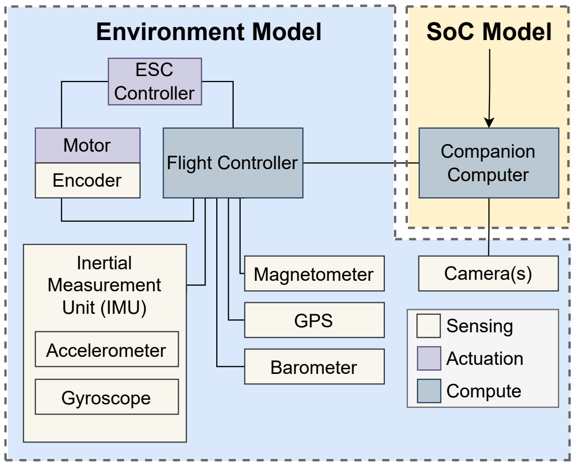


Figure 7: The block diagram of a UAV system modeled by RoSÉ, representative of a typical UAV with a flight controller and an onboard companion computer.

data from the AirSim APIs through sensors such as cameras or Inertial Measurement Units (IMUs). However, modeling actuation and compute is a more nuanced task in robotics UAV systems, given the hierarchy of hardware and software controllers present in typical UAVs. As an example, motor angular velocity is typically controlled by mixed-signal circuitry in electronic speed controllers (ESCs), which take in velocity targets from a flight controller.

Likewise, low-level control itself is handled by the flight controller, which interfaces with low-bandwidth sensors such as IMUs. The flight controller sends targets via pulse-width modulation (PWM) to the ESCs, computed by using low-latency PID algorithms. Moving up the hierarchy, targets for the PID controllers are generated by the companion computer running trajectory planners or high-level control algorithms such as model predictive control (MPC). The companion computer also does not directly interface with sensors and actuators, instead communicating through the flight controller over a protocol such as MAVLink [1]. Given that flight controller behavior is well understood, in our experimental setup, the flight controller is modeled using AirSim's software-in-the-loop controller. However, the companion SoC is simulated using RTL simulation to accurately capture its compute and data movement behaviors. This partitioning is depicted in Figure 7.

RoSÉ supports any sensor data and commands that can be serialized through the synchronizer. For the evaluations in this paper, the drone is equipped with a first-person view (FPV) camera with a field-of-view (FOV) of 90 degrees. Additionally, the onboard flight controller has access to an IMU. The companion computer sends commands to the flight controller containing angular and linear velocity targets.

4.2 Experimental Setup

4.2.1 Hardware. To evaluate the effect of the SoC architecture on robot performance, we considered several different hardware configurations. We use Chipyard [10], a framework for designing and evaluating full-system hardware. The generated designs were then compiled to FPGA bitstreams in FireSim. We generate two types of CPU cores. For our in-order core, we use a Rocket CPU, a 5-stage in-order scalar processor core generator[11], and for the superscalar out-of-order CPU we use SonicBOOM, an open-source RTL implementation of a RISC-V out-of-order core [57]. To add DNN acceleration capacity to both cores, we generate a systolic-array accelerator using Gemmini, a full-stack DNN accelerator generator[23]. A summary of the generated designs is shown in Table 2. As the DNNs being evaluated in this paper use floating-point datatypes, Gemmini was configured to use FP32 data. For this configuration, we generate a 4×4 FP32 mesh to match Gemmini's 128-bit maximum memory bus width. The systolic array uses a weight-stationary dataflow to match the workload, and we use a 256KB scratchpad with a 64KB accumulator.

Configuration	A	B	C
CPU	3-wide BOOM	Rocket	3-wide BOOM
Accelerator	Gemmini	Gemmini	None

Table 2: Hardware configurations evaluated using RoSÉ.

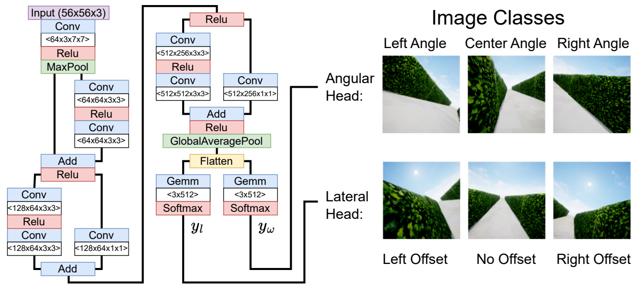


Figure 8: A multi-headed DNN is used as a controller. The DNN heads classify images as left, center, or right angular and lateral views of a trail.

4.2.2 Software. Software running on our simulated UAV is partitioned between a flight controller and a companion computer, as illustrated in Figure 7. The following paragraphs discuss the software that runs on both systems.

Flight Controller. The flight controller used in evaluations is based on the `SimpleFlight` controller provided by AirSim. `Simple Flight` contains a hierarchy of PID controllers that manage the position, velocity, and angle of attack targets. The flight controller takes in angular and velocity control targets from the companion computer, and uses the control hierarchy to track the most recent target recieved.

Companion Computer. The companion computer runs a full-stack setup, running robotics UAV applications on top of a RISC-V Fedora image running `Linux 5.7.0-rc3`. In particular, we evaluate DNN-based high-level end-to-end controllers that have been widely used for UAVs. Examples include ResNet-based controllers based on classifiers such as DroNet [34, 41] and TrailNet [53]. DroNet uses a dual-headed architecture, with one head outputting a target angle and one head outputting a collision probability. TrailNet, on the other hand, has a dual-headed classifier with one head outputting a classification of the UAV's angle relative to a trail and the other head

Model	ResNet6	ResNet11	ResNet14	ResNet18	ResNet34
Latency (BOOM with Gemmini)	77ms	83ms	85ms	130ms	225ms
Latency (Rocket with Gemmini)	101ms	108ms	125ms	185ms	300ms
Validation Accuracy	72%	78%	82%	83%	86%

Table 3: Latency and accuracy of trained DNN controllers.

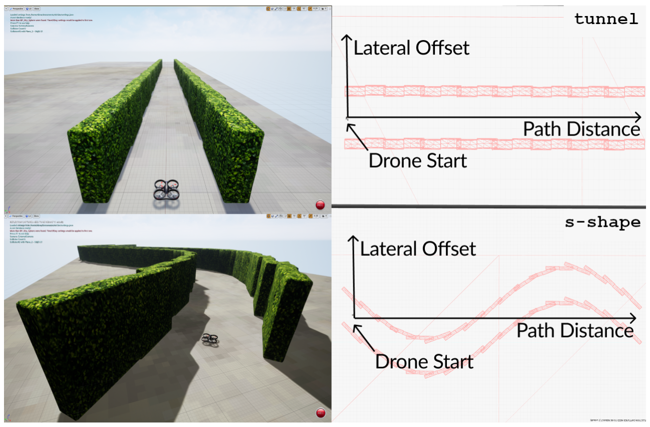


Figure 9: Visualizations of the environment setup for the drone evaluations, showing the `tunnel` environment on the top row and the `s-shape` environment on the bottom. The left column shows the rendered UE4 environment, while the right column is its top projection.

outputting a classification of the UAV's lateral offset to a trail. The ResNets evaluated in this work are based on TrailNet's architecture and are depicted in Figure 8, with y_l as the lateral classification and y_ω as the angular classification. The softmax outputs are then used to compute target velocity, v_l, and target angular velocity, ω, scaled by controller gains β as shown in Equation 2. These targets are used by the flight controller to navigate the UAV.

$$v_l = \beta_l(y_l^{\text{left}} - y_l^{\text{right}}), \quad \omega = \beta_\omega(y_\omega^{\text{right}} - y_\omega^{\text{left}}) \qquad (2)$$

We trained our own DNN models using the AirSim environment described in Section 4.2.3. To train both heads of the DNN, the data is split into an angular training dataset and a lateral training dataset. Both datasets have three classes (left, center, and right), with 2000 images sampled for each class, each with randomized positions, angles, and textures, for a total of 12,000 images. Sample images of each class are shown in Figure 8. The DNNs are then evaluated on a separate dataset of 1200 validation images, where we achieve 72% to 86% evaluation accuracy, comparable to the 85% accuracy delivered by TrailNet. The accuracy and average latency of the DNNs are listed in Table 3.

4.2.3 Environments. Figure 9 shows the environments we built to evaluate the UAV system. The first `tunnel` is a straight path 50 meters in length and 3.2 meters wide. The second `s-shape` is an "S" shaped trajectory of 80 meters in length. The task for the UAV is

On-Premise Deployment		
Simulator	AirSim	FireSim
CPU	Intel Core i7-3930K	Intel Xeon Gold 6242
Frequency	@3.2GHz	@2.8GHz
GPU	GeForceGTX TITAN X	N/A
FPGA	N/A	Xilinx U250
OS	Ubuntu 18.04.6 LTS	Ubuntu 18.04.6 LTS
Cloud Deployment (AWS)		
Simulator	AirSim	FireSim
Instance	`g4dn.2xlarge`	`f1.2xlarge`
CPU	Intel Xeon Platinum 8259CL	Intel Xeon E5-2686
Frequency	@2.5GHz	@2.3GHz
GPU	Tesla T4	N/A
FPGA	N/A	Xilinx VU9P
OS	Ubuntu 18.04.6 LTS	CentOS 7.9.2009

Table 4: RoSÉ deployment configurations.

to perform visual navigation from the start to the end of the map, avoiding collisions with the walls. `tunnel` tests the UAV's ability to precisely navigate a narrow path and achieve a stable trajectory regardless of the initial conditions. `s-shape` is a wider map with more room for error but requires constant correction from the drone to avoid navigating into the boundaries. Our DNNs were trained on `tunnel` and evaluated on both `tunnel` and `s-shape`.

4.2.4 Deployment. RoSÉ requires a GPU (for 3D rendering) and an FPGA (to accelerate RTL simulation) in order to run simulations. For the purpose of the evaluations in this paper, we tested both cloud and on-premise deployments. For our cloud setup, we host AirSim on a `g4dn.2xlarge` AWS instance, and deploy FireSim simulations on an `f1.2xlarge` instance. For on-premise evaluation, we host AirSim and FireSim on a desktop and server on a local network. Both configurations are shown in Table 4. In our evaluation, we execute the synchronizer node on the FireSim machine to reduce latency to the RoSÉ Bridge, but it can be deployed on any device.

5 ROSÉ EVALUATION

RoSÉ enables holistic evaluation of robotics SoCs by co-simulating hardware, algorithms, and environments in a closed-loop fashion. To demonstrate the effectiveness of RoSÉ, this section discusses opportunities to co-design SoC hardware, applications, and robot behaviors of UAVs. Additionally, we also quantitatively evaluate the tradeoffs between co-simulation accuracy and throughput.

5.1 Effects of SoC Architectures

By integrating the RTL simulation of an SoC, RoSÉ is able to provide insight into the effect of architectural choices on the behavior of a robot. For the task of DNN-based navigation, we consider three configurations described in Section 4.2.1. We deployed the SoCs

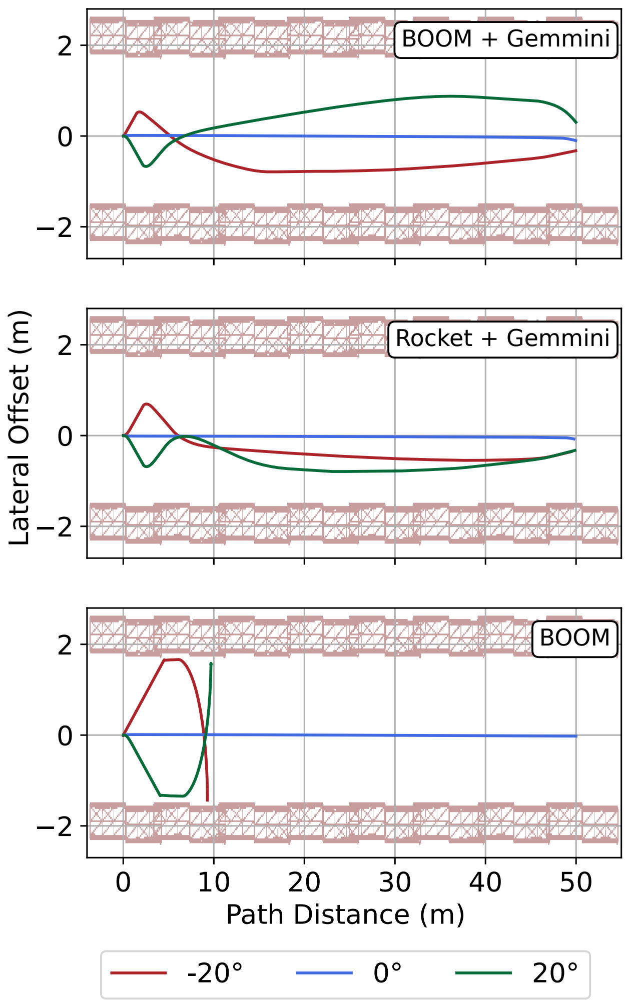


Figure 10: UAV trajectories for different hardware configurations. For each hardware configuration, three initial conditions are set for $-20°$, $0°$, and $20°$ degrees. Each UAV is running a ResNet14 controller DNN setting a target velocity of 3m/s. The tunnel boundaries are at $y = \pm 1.6$m, marked by gray dashed lines.

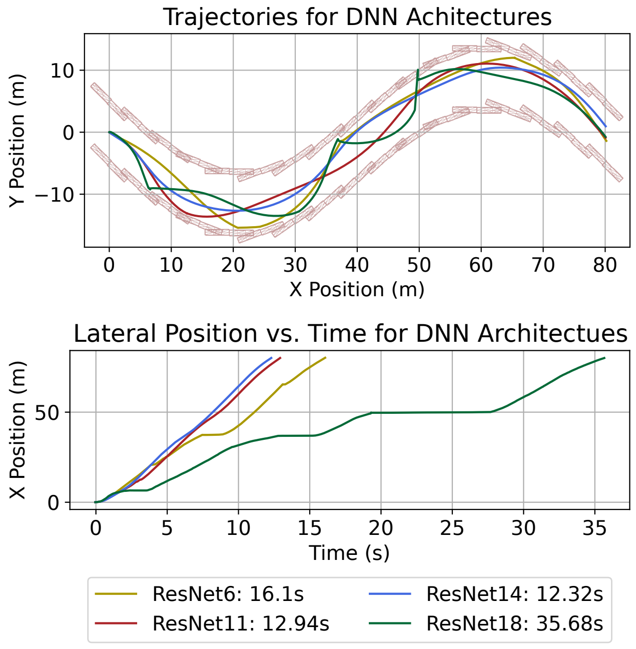


Figure 11: UAV trajectories generated by sweeping across different DNN architectures. Above, we show a visualization of the trajectories in the s-shape environment, below we show the lateral position of the UAV in the environment over time. The mission is completed upon reaching an x-coordinate of 80.

with the same initial conditions in the environment `tunnel`, simulating three different trajectories, starting at $-20°$, $0°$, and $20°$ relative to the center.

When starting at $0°$, while an ideal UAV will not need to adjust its lateral and angular motion, given that the UAV is taking off from the ground, the UAV makes minor corrections to stabilize its trajectory over time. On the other hand, for the angled starting positions, the UAV first alters its trajectory before colliding with the wall in front of it and then stabilizes its trajectory toward the center of the tunnel.

We show the results of three case studies in Figure 10, where we evaluate the hardware configurations listed in 2. First, we find that UAVs with CPU-only SoCs as shown in Figure 10 (c) cannot navigate the `tunnel` environment. We observe a 6-second latency between an image request and control target update with a BOOM-only SoC, resulting in the UAV colliding before an inference is made.

On the other UAVs controlled by SoCs with a DNN accelerator have similar trajectories, and are less sensitive to whether a more powerful CPU, i.e., BOOM, or a small CPU, i.e., Rocket, is driving the accelerator in the `tunnel` environment. This type of subtle but important hardware design space exploration for robotics SoCs cannot be done without the closed-loop co-simulation capability in RoSÉ.

5.2 Effects of Algorithm on Robot Behavior

To evaluate the impact of DNN architectures on robot performance, we examine different ResNet configurations, listed in Table 3. Each DNN was evaluated in the environment `s-shape`, with a flight velocity target of 9m/s. Despite the fact that the larger ResNet configurations have a higher validation accuracy, they performed poorly in flight, as shown in Figure 11. This is due to multiple factors: First, because of their high latency, DNNs are prone to violating deadlines while navigating, especially at higher speeds. Second, the high capacity of the DNNs is, in practice, detrimental to the utility of the DNNs as controllers, as they classify the headings and offsets with a higher confidence level, resulting in sharper changes in trajectory after each inference is performed. This factor, compounded with the higher latency cost of DNNs, results in ResNet34 being unable to complete s-shape without multiple collisions. Furthermore, ResNet14 has the shortest mission time due to following the most optimal trajectory.

On the other hand, there is a tradeoff for using a DNN that is too small as well. As shown in Table 3, the ResNet6 and ResNet11 have lower accuracy, which results in both incorrect classifications, as well as lower prediction confidence. Despite having the fastest control loop, ResNet6 is unable to complete the trajectory without collisions. Because control gains are computed proportionally to softmax outputs, as in Equation 2, smaller DNNs result in lower magnitude angular and lateral corrections. Not only do they make incorrect predictions, moving the UAV closer to obstacles, but the fact that they make less confident predictions results in a wider turn radius. However, even if using an argmax policy on the DNN outputs to compensate, ResNet6 is unable to complete the trajectory due to poor classification accuracy and incorrect control outputs.

One of the key contributors to UAV performance is the ability to perform its control loop within required deadlines. Deadlines vary depending on the environment (e.g. the presence of obstacles), properties of the robot (e.g. maximum acceleration), and the robot's state (e.g. current velocity.) Deadlines can be used by models to set constraints on robotic systems, such as maximum safe velocity [32], and can be either statically determined or set by dynamic runtimes [14]. To demonstrate the effectiveness of RoSÉ, we define the deadline, t_{process}, as follows, to quantify the performance of the drone's trajectory navigation task:

$$t_{\text{collision}} = \frac{D_{\text{obj}}}{\text{Velocity}} \tag{3}$$

$$t_{\text{collision}} \geq t_{\text{sensor}} + t_{\text{process}} + t_{\text{actuation}} \tag{4}$$

$$t_{\text{process}} \leq t_{\text{collision}} - t_{\text{sensor}} - t_{\text{actuation}} \tag{5}$$

Here, $t_{\text{collision}}$ is defined as the time until collision, and D_{obj} is defined as the depth of the closest object in the current heading of the UAV. Equation 4 breaks down the time until collision into sensor latency, compute latency, and actuation latency [40]. We can then derive the compute latency in Equation 5; unless the UAV can alter its trajectory before the deadline expires, a collision will occur. This gives us the upper bound of compute time, which provides RoSÉ users a benchmark to tune their configurations accordingly.

In addition to the DNN architecture, different settings for system parameters will also affect robot behavior. As shown in Figure 12, the effect of the target velocity outputted by the controller impacts the UAV's trajectory and the resulting quality of flight. When the velocity is 6m/s, the UAV follows the safest trajectory; when the velocity increases to 9m/s, the UAV completes the task in the shortest mission time (12.14 seconds); however, when the velocity increases to 12m/s, it results in a collision. These collisions occur directly after deadline violations, as the 85ms needed to perform inference violates the constraints set by the UAVs flight.

5.3 Effects of Dynamic Runtimes

RoSÉ supports evaluating workloads with dynamic, environment-dependent runtimes. Instead of uniformly executing the same ResNet in all scenarios, we adaptively select which DNN is used to generate control targets depending on the system deadlines. We determine

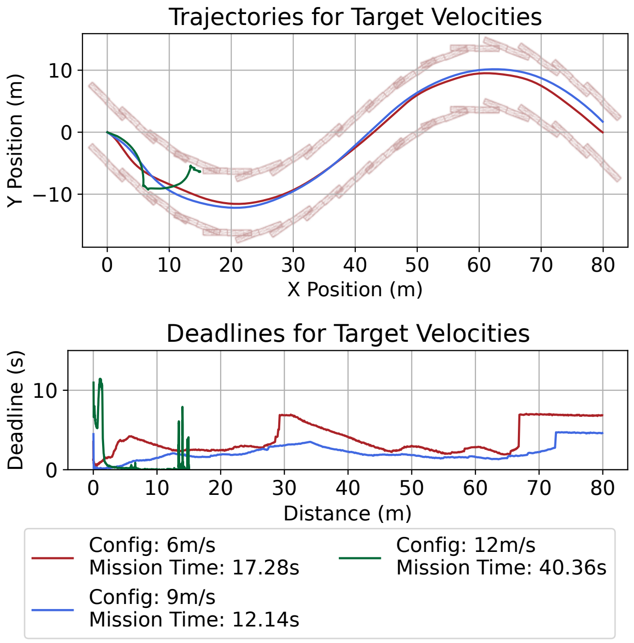


Figure 12: A sweep of the flight controller's velocity targets running ResNet14 on BOOM+Gemmini.

the deadline by measuring forward-facing depth-sensor readings from the UAV.

We select ResNet14 as our high-accuracy, high-latency network, and ResNet6 as our low-accuracy, low-latency network. When the deadline is over a threshold, we use the classifier outputs for ResNet14. However, when the UAV is at risk of collision, we dynamically switch to ResNet6 so that we can get updated control targets faster. Furthermore, instead of scaling the control targets by the DNN's confidence, we use the argmax of both the angular and lateral classes when using ResNet6, so that the UAV corrects its trajectory faster.

By using a dynamic runtime, the companion computer SoC can achieve a lower mission time while using fewer hardware resources, as depicted in Figure 13. The application latency is determined by the total time to complete the trajectory, while the accelerator activity factor represents the fraction of time DNN accelerators are actively executing layers. A lower activity factor frees system resources for other applications and reduces energy consumption.

When statically selecting which DNN is used, system designers can reduce the activity factor by running a smaller DNN, which comes at the cost of a longer mission time due to sub-optimal UAV control. However, RoSÉ reveals that dynamically selecting DNNs can achieve reduced activity factors while also improving mission time. This is despite the overhead of hosting two ONNX Runtime sessions, which results in 15% fewer inferences performed in the dynamic application compared to statically running ResNet14. This evaluation depends on an end-to-end closed-loop simulation environment because the UAV behavior depends on data-dependent inference latencies and tradeoffs across the hardware/software stack.

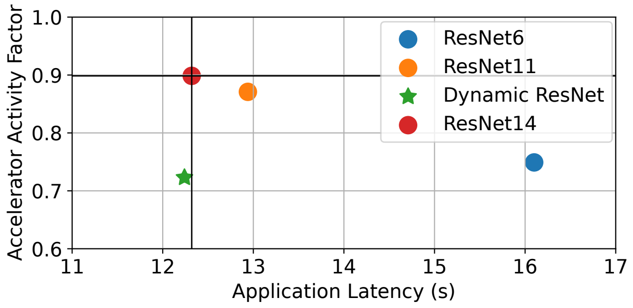


Figure 13: A comparison of the application runtimes and DNN accelerator activity factors across static and dynamically allocated DNN tasks. The dynamic case selects between executing ResNet14 and ResNet6 at runtime.

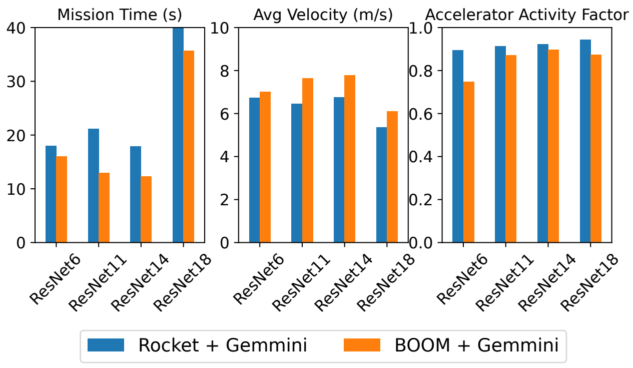


Figure 14: A comparison of low-level parameters including mission time, velocity, and DNN activity for two hardware architectures: BOOM+Gemmini, and Rocket+Gemmini. RoSÉ reveals differing optimal design points when the SoC architecture is changed.

5.4 Evaluating Hardware/Software Co-Design

Finally, RoSÉ enables the comparison of different hardware architectures in combination with different algorithms. By sweeping across DNN architectures for both BOOM+Gemmini as well as Rocket+Gemmini, we observe different system-level behavior between the two architectures. When BOOM is used, ResNet14 is the optimal design point in terms of both latency and accuracy, resulting in the shortest mission time and fastest flight velocity. However, when using Rocket, the SoC design struggles to complete the trajectory without recovering from collisions, resulting in significantly higher mission times compared to BOOM. In this case, ResNet6 even performs better compared to larger networks such as ResNet11, as the DNN's low latency is more critical to compensate for Rocket's lower performance. In this case, ResNet11 is a poor design choice compared to the lower latency of ResNet6 and the higher accuracy of ResNet14. Compared to hardware-in-loop simulation with pre-existing hardware, this experiment reveals how RoSÉ can identify how optimal design points can change when changing architectural features beyond core and frequency scaling.

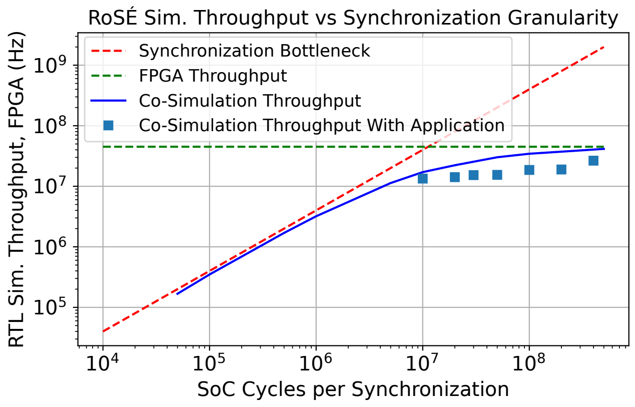
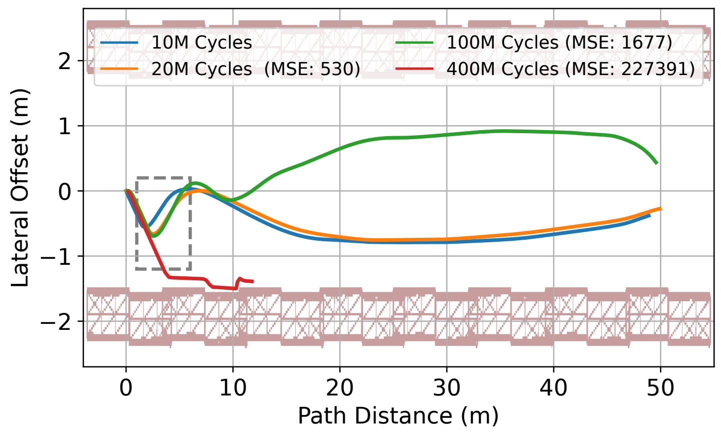


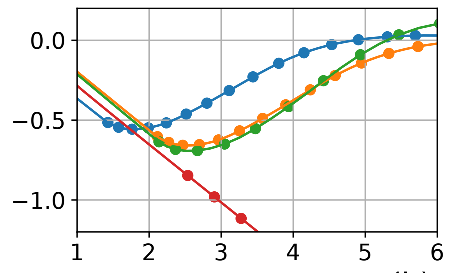
Figure 15: RoSÉ throughput measurements and bottlenecks on a local FPGA deployment, sweeping across synchronization granularities. Simulation throughput is bottlenecked both by the maximum possible FPGA throughput as well as by the minimum synchronization period.

5.5 Evaluating Effects of Synchronization Granularity

RoSÉ supports a user-configured synchronization rate between the RTL simulation and the environment simulation. To evaluate the trade-off between fine- and coarse-grained synchronization, we perform an experiment sweeping the synchronization granularity of the simulator while keeping initial conditions the same, as shown in Figure 16. We show the trajectories generated by RoSÉ in Figure 16 (a), as well as a close-up of the first few meters of the UAV's flight in Figure 16 (b) where the trajectories start to diverge. Each point marked on the trajectory marks the end of a synchronization period during which the simulated SoC requested an image. We notice that as the synchronization granularity increases, the UAV becomes less responsive due an artificial latency induced by synchronization, both when requesting images and when sending a command back to the UAV.

An important consequences of changing synchronization granularity is impact on the delay between subsequent inferences, measured in simulation time. At a finer granularity, the inferences are unaffected by the synchronization rate. However, as granularity increases to greater than 20 million cycles, inferences have a much more significant delay. This is because AirSim finishes its simulation quickly and then waits for FireSim to finish. When FireSim request the next image for reference, it has to stall until the next synchronization period, at which point AirSim resumes and transmits the necessary data. The synchronization period contributes to latency, reducing the UAV's performance. This effect is measured in Figure 16 (c), where the latency between an image request and DNN controller response is measured. While at a 10M granularity synchronization the result is slightly above the expected 125ms compute latency for the DNN (due to the overhead of loading the image from the I/O), the measured latency continues to increase as synchronization grows coarser. By 400M cycles, the observed latency of 400ms is over 3× higher than the expected end-to-end latency under ideal conditions.





(b)

Sync. Cycles	10M	20M	100M	400M
DNN Latency (s)	0.13	0.14	0.20	0.40

(c)

Figure 16: A comparison of the effects of synchronization granularity on the simulated trajectories of the UAV. For (a), the UAV was set to the same initial angle of 20° in the `tunnel` environment, running ResNet14 at a target velocity of 3m/s. The synchronization granularity was swept from 10M clock cycles/1 AirSim frame to 400M cycles/40 AirSim frames. Despite the identical starting configuration, trajectories start to diverge due to different synchronization granularity. (b) zooms into the start of each trajectory, marking each image request from the SoC with a point. Finally, (c) compares the measured simulation-time delay between image requests and DNN outputs for each synchronization granularity.

However, despite the higher synchronization granularity introducing artificial latency, executing finely synchronized simulations comes with a cost. Figure 15 shows the trade-off between FPGA cycles executed per synchronization period and the FPGA simulation throughput. Finer granularity results in fewer cycles executed per synchronization. This penalty becomes more severe when the simulator is bottlenecked by the FireSim scheduler polling the RoSÉ Bridge. Therefore, the synchronization granularity is a trade-off between accuracy and performance. If users prefer accurate simulation, then they should run a fine granularity simulation; if users want a quick parametric sweep over a series of designs, then larger granularity will provide faster simulation execution. However, given the results of this sweep, a synchronization granularity of 10-100 million cycles per frame grants reasonable performance without sacrificing accuracy.

6 FUTURE DIRECTIONS

Beyond the evaluation presented in this paper, users can use RoSÉ to analyze dynamic behavior throughout the robotics stack. For example, compared to DNN-based tasks, many classical algorithms such as SLAM and nonlinear MPC build upon iterative optimization algorithms or dynamically scaling data structures. These applications have data-dependent runtime behaviors and access patterns, where RoSÉ can capture their performance implications on both hardware and software.

Additionally, compared to the regular execution patterns of DNNs such as TrailNet, state-of-the-art DNN workloads in robotics also have more irregular execution patterns. For instance, controller networks that perform sensor fusion have separate backbones for each class of sensor [40]. In this case, branches of the network can be executed at different rates depending on sensor data, providing opportunities for both software and hardware schedulers to improve performance.

Finally, robots face different constraints across application domains and robot morphologies, such as latency, energy, power, and robustness. These domains include autonomous vehicles, industrial robots, and humanoid robots. Future directions for RoSÉ include integrating additional simulators such as CARLA [20] and Isaac Gym [42] to enable more comprehensive coverage across robotics.

7 CONCLUSION

In response to the growing complexity of robotics workloads, hardware, and applications, we propose RoSÉ, enabling robotics UAV researchers and architects to comprehensively evaluate robotic UAV SoCs. RoSÉ captures a full-stack simulation of a robot by integrating the simulation of a robot UAV's environment and its RTL, enabling design space exploration of robot environments, algorithms, hardware, and system parameters within a unified simulation environment.

RoSÉ enables researchers to better study and analyze the trade-offs of robotics systems without the overhead of taping out an SoC. Building upon RoSÉ by introducing new applications, robotics environments, and architectures will enable the agile development of robotics SoCs across diverse domains.

8 ACKNOWLEDGEMENTS

This research was supported in part by the NSF Awards CCF-2238346 and CCF-1955450 and in part by SLICE Lab industrial sponsors and affiliates. This research was also funded in part by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] [n. d.]. MAVLink. https://mavlink.io/en/
[2] [n. d.]. MoveIt: Moving robots into the future. https://moveit.ros.org/.
[3] [n. d.]. ONNX: Open Neural Network Exchange. https://onnx.ai/.
[4] [n. d.]. ONNX Runtime: Optimize and Accelerate Machine Learning Inferencing and Training. https://microsoft.github.io/onnxruntime/.
[5] [n. d.]. OpenSLAM Gmapping. https://openslam-org.github.io/gmapping.html/.
[6] [n. d.]. RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim.
[7] [n. d.]. tf2: Transform Library 2. http://wiki.ros.org/tf2/.
[8] Miguel Acevedo. 2016. FPGA-Based Hardware-In-the-Loop Co-Simulator Platform for SystemModeler.
[9] Srikrishna Acharya, Amrutur Bharadwaj, Yogesh Simmhan, Aditya Gopalan, Parimal Parag, and Himanshu Tyagi. 2020. Cornet: A co-simulation middleware for robot networks. In *International Conference on COMmunication Systems & NETworkS (COMSNETS)*.
[10] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* (2020).
[11] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* (2011).
[13] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. 2018. Mavbench: Micro aerial vehicle benchmarking. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[14] Behzad Boroujerdian, Radhika Ghosal, Jonathan Cruz, Brian Plancher, and Vijay Janapa Reddi. 2021. Roborun: A robot runtime to exploit spatial heterogeneity. In *58th ACM/IEEE Design Automation Conference (DAC)*.
[15] Gary Bradski and Adrian Kaehler. 2000. OpenCV. *Dr. Dobb's journal of software tools* (2000).
[16] David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
[17] Benjamin Chrétien, Adrien Escande, and Abderrahmane Kheddar. 2016. GPU robot motion planning using semi-infinite nonlinear programming. *IEEE Transactions on Parallel and Distributed Systems* (2016).
[18] Erwin Coumans and Yunfei Bai. 2016. Pybullet, a python module for physics simulation for games, robotics and machine learning. (2016).
[19] Jeffrey Delmerico, Titus Cieslewski, Henri Rebecq, Matthias Faessler, and Davide Scaramuzza. 2019. Are we ready for autonomous drone racing? the UZH-FPV drone racing dataset. In *International Conference on Robotics and Automation (ICRA)*.
[20] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. In *Conference on robot learning*.
[21] Pierre-Emile Duhamel, Judson Porter, Benjamin Finio, Geoffrey Barrows, David Brooks, Gu-Yeon Wei, and Robert Wood. 2011. Hardware in the loop for optical flow sensing in a robotic bee. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.
[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.
[23] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*.
[24] Ramyad Hadidi, Bahar Asgari, Sam Jijina, Adriana Amyette, Nima Shoghi, and Hyesoon Kim. 2021. Quantifying the Design-Space Tradeoffs in Autonomous Drones. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[25] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* (2019).
[26] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. 2021. Exploring Extended Reality with ILLIXR: A new Playground for Architecture Research.
[27] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[28] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
[29] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. MoCA: Memory-Centric, Adaptive Execution for Multi-Tenant Deep Neural Networks. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[30] Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
[31] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. 2011. A Flexible and Scalable SLAM System with Full 3D Motion Estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*.
[32] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Ninad Jadhav, Aleksandra Faust, and Vijay Janapa Reddi. 2022. Roofline model for uavs: A bottleneck analysis tool for onboard compute characterization of autonomous unmanned aerial vehicles. *arXiv preprint arXiv:2204.10898* (2022).
[33] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Programmable Interconnects. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.
[34] Christos Kyrkou, George Plastiras, Theocharis Theocharides, Stylianos I Venieris, and Christos-Savvas Bouganis. 2018. DroNet: Efficient convolutional neural network detector for real-time UAV applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
[35] Nathan O Lambert, Daniel S Drew, Joseph Yaconelli, Sergey Levine, Roberto Calandra, and Kristofer SJ Pister. 2019. Low-level control of a quadrotor with deep model-based reinforcement learning. *IEEE Robotics and Automation Letters* (2019).
[36] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
[37] Ruige Li, Xiangcai Huang, Sijia Tian, Rong Hu, Dingxin He, and Qiang Gu. 2019. FPGA-based Design and Implementation of Real-time Robot Motion Planning. In *International Conference on Information Science and Technology (ICIST)*.
[38] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. 2018. Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*.
[39] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. 2018. Gpu-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning*.
[40] Antonio Loquercio, Elia Kaufmann, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. 2021. Learning high-speed flight in the wild. *Science Robotics* (2021).
[41] Antonio Loquercio, Ana I Maqueda, Carlos R Del-Blanco, and Davide Scaramuzza. 2018. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters* (2018).
[42] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. 2021. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470* (2021).
[43] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. 2016. The microarchitecture of a real-time robot motion planning accelerator. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[44] Sabrina M Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. 2021. Robomorphic computing: a design methodology for domain-specific accelerators parameterized by robot morphology. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.
[45] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[46] Brian Plancher, Sabrina M Neuman, Radhika Ghosal, Scott Kuindersma, and Vijay Janapa Reddi. 2022. Grid: Gpu-accelerated rigid body dynamics with analytical gradients. In *International Conference on Robotics and Automation (ICRA)*. IEEE.
[47] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*.
[48] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *Proceedings of the International Symposium on Performance*

Analysis of Systems and Software (ISPASS).

[49] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator Simulator. *arXiv preprint arXiv:1811.02883* (2018).

[50] Louis K Scheffer. 2021. The Physical Design of Biological Systems-Insights from the Fly Brain. In *International Symposium on Physical Design*.

[51] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*.

[52] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[53] Nikolai Smolyanskiy, Alexey Kamenev, Jeffrey Smith, and Stan Birchfield. 2017. Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[54] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. 2019. Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones. *IEEE Journal of Solid-State Circuits* (2019).

[55] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *IEEE/RSJ international conference on intelligent robots and systems*.

[56] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. 2021. A Survey of FPGA-Based Robotic Computing. arXiv:2009.06034

[57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix describes how to use RoSÉ to run end-to-end robotics simulations, and how to reproduce results as shown in Section 5. The instructions assume that a user already has robotic applications and hardware configurations developed, and provides reference examples used in the evaluation of this work. While RoSÉ can be used to develop new hardware and software, instructions to do so are outside of the scope of the artifact evaluation.

A.2 Artifact Meta-Information Checklist

- **Runtime environment:** Ubuntu 18.04.6 LTS, Vitis v2021.1
- **Hardware (FireSim):** Intel Xeon Gold 6242, Xilinx U250
- **Hardware (AirSim):** AWS EC2 Instance (`g4dn.2xlarge`), Intel Xeon Platinum 8259CL, Tesla T4.
- **Required Disk Space:** 200GB
- **Experiments:** AirSim/FireSim end-to-end full stack simulations of a UAV using RoSÉ, running DNN-based controllers. Experiments evaluate both UAV and simulator performance.
- **Languages:** Chisel (RTL), C++ (FireSim bridge drivers, robotic control software), Python (Synchronizer and scheduler.)
- **Quantitative Metrics:** DNN Latency, mission time, average flight velocity, accelerator activity factor, as in Section 5.
- **Qualitative Metrics:** Flight trajectories, flight recordings.
- **Output:** CSV logs from the synchronizer, tracking UAV dynamics, sensing requests, and control targets.
- **Installation Time:** 4 hours (scripted installation).
- **Experiment Duration:** 48 hours (scripted execution and parsing)
- **Publicly available:** Yes.
- **Code licenses:** Several, see download.

A.3 Description

A.3.1 How to access. The artifact consists of the core RoSÉ repository, as well as modifications to Firesim, Chipyard, and ONNX Runtime, deployed by RoSÉ through patches.

(1) RoSÉ Core: Deployment, synchronization, and evaluation software, as well as hardware configurations, and patches to FireSim, Chipyard, and ONNX Runtime. (https://doi.org/10.5281/zenodo.7824144)
(2) FireSim: Top-level FPGA-Accelerated RTL Simulation Environment (https://github.com/firesim/firesim)
(3) Chipyard: RISC-V SoC generation environment (https://github.com/ucb-bar/chipyard)
(4) RISCV ONNX Runtime: Software for executing HW-accelerated DNN models, modified for use in UAV control (https://github.com/ucb-bar/onnxruntime-riscv/tree/onnx-rose).

Additionally, RoSÉ builds upon the following infrastructures. For the purpose of the evaluation, binaries for simulators built from Unreal Engine and AirSim are provided.

(1) Unreal Engine: 3D Environment development platform (https://www.unrealengine.com/en-US/ue-on-github)
(2) AirSim: UAV simulation plugin for Unreal Engine (https://github.com/microsoft/AirSim)

A.3.2 Dependencies - Hardware. To run a full simulation with RoSÉ, access to a GPU and FPGA is required, although these can be hosted on separate computers. For this artifact evaluation, instructions for running simulations on a locally-provisioned FPGA are provided. However, RoSÉ can also be used using AWS EC2 FPGA instances (e.g. `f1.2xlarge`). In this artifact we provide build scripts for generating bitstreams for locally-provisioned FPGAs.

Additionally, GPU access is needed in order to run robotics environment simulations with rendering. For this evaluation, AirSim binaries packaged using Unreal Engine are provided.

A.3.3 Dependencies - Software. To use RoSÉ, ensure that Vitis and Vivado v2021.1 are installed, licensed, and are correctly set on the system `PATH`. All other requirements are automatically installed by scripts in the following sections. If developing AirSim environments, Unreal Engine 4.25 is needed.

A.4 Installation

Running all the steps below in a `screen` or `tmux` session is recommended, as some commands may take several hours to execute.

To begin installation, download the artifact from Zenodo:

```
$ wget https://zenodo.org/record/7824144/files/RoSE.zip
$ unzip RoSE.zip && cd ./RoSE/
```

A.4.1 FireSim Installation. Begin by installing FireSim by running the following commands within the RoSÉ repository.

```
$ git submodule update -init ./soc/sim/firesim
$ cd ./soc/sim/firesim
$ ./scripts/machine-launch-script.sh
$ ./build-setup.sh
$ source sourceme-f1-manager.sh
$ firesim managerinit -platform vitis
```

A.4.2 RoSÉ Installation. Begin by entering the RoSÉ repository:

```
$ cd RoSE
```

Next, within RoSÉ, run the setup script to set the proper environment variables. Make sure to run this script whenever starting a new interactive shell.

```
$ source rose-setup.sh
```

After this is complete, run the following script to patch FireSim and Chipyard with the modifications described in Figure 5, and to instantiate submodules.

```
$ bash soc/setup.sh
```

After this setup is complete, run the following script to build binaries for the trail-navigation controllers evaluated in Section 4 for generating RISC-V Fedora images containing the controllers and ONNX models.

```
$ bash soc/build.sh
```

Next, run the following script to install dependencies and configure parameters for the RoSÉ deployment scripts, using the IP address of the GPU system that will be used to run the provided AirSim binaries.

```
$ source deploy/setup.sh [AIRSIM IP]
```

A.4.3 Bitstream Generation. To build bitstreams for Rocket+Gemmini and BOOM+Gemmini configurations, run the following.

```
$ bash soc/scripts/buildbitstreams.sh
```

A.4.4 DNN Training. This artifact provides pre-trained models for evaluation. To train new classifier DNNs using the provided datasets, run the following, selecting between the given ResNet configurations. Each training run will output an ONNX model named `trail_dnn_resnet[xy].onnx`.

```
$ bash env/train/train_resnet.py (6|11|14|18|34|50)
```

Finally, the steps for building custom Unreal Engine maps are out of the scope of this evaluation. However, new environments can be built using the documentation provided at https://microsoft.github.io/AirSim/build_linux/.

A.5 Experiment Workflow

Before running any experiments, first run provided AirSim executable on a GPU instance, which can be left running for all simulations. This binary contains models for both the `tunnel` and `s-shape` environments. Ensure that port 41451 is open for TCP on the GPU instance to be able to access AirSim.

```
$ bash env/world/airsim_s/LinuxNoEditor/Blocks.sh
```

Now that the environment has been set up and the target hardware and software have been built, one can run the experiments in this work by launching an AirSim simulation and running the following scripts. All the experiments can be executed by running `run-all.sh`. This will generate CSV files as well as videos recorded from the front-facing camera of the simulated UAV in `deploy/hephaestus/logs/`.

```
$ bash deploy/scripts/run-all.sh
```

To run individual experiments corresponding to the figures in this work, we provide the following scripts, which are all included in the main script.

```
# Figure 10
$ bash deploy/scripts/tunnel-exp.sh
# Figures 15, 16
$ bash deploy/scripts/rose-perf-sync-only.sh
$ bash deploy/scripts/rose-perf-tunnel-exp.sh
# Figures 11, 14
$ bash deploy/scripts/rose-hw-sw-sweep.sh
# Figure 12
$ bash deploy/scripts/rose-velocity-sweep.sh
# Figure 13
$ bash deploy/scripts/rose-dynamic-exp.sh
```

A.6 Figures and Evaluation

After executing the prior experiments, figures can be generated using the CSV outputs by running the following command. The figures will be available in `deploy/figures/`.

```
$ python3 deploy/scripts/generate-figures.py
```

A.7 Interpreting Results

It is expected to have some variations in trajectories and mission times across multiple simulations with the same initial conditions. This is caused by randomness in Unreal Engine that influences experimental results despite the fact that FireSim itself is deterministic. Examples include animation in the environment, as well as noise in the AirSim physics models used for simulated sensing and actuation.

This variation has more impact on unsafe/poorly performing configurations, particularly those with UAV collisions, as minor variations in the angle of attack during a collision can result in major trajectory differences afterward. On the other hand, stable flights that do not approach any obstacles will have minimal variation between simulations.

In addition to using the generated plots, it is useful to view the generated first person videos stored in the log directories to qualitatively analyze how a controller performs on the recorded data.

A.8 Experiment Customization

A.8.1 Building New FPGA Images. In addition to the provided SoC configurations, users can evaluate other designs. To evaluate new designs, refer to the Chipyard documentation, as well as the example RoSE-annotated configs found in `soc/src/main/scala/RoSEConfigs.scala`.

A.8.2 Designing AirSim Environments. If users install Unreal Engine as well as AirSim, it is possible to create new maps/environments for robot agents to interact with. By default, one can modify the `Blocks` environment provided by AirSim. Additional assets and maps can be designed by users or obtained from the Unreal Marketplace.

A.8.3 Changing Simulation Parameters. RoSÉ provides flags that can be used to select different simulation parameters. To view available parameters for deploying simulations, refer to `deploy/hephaestus/runner.py`. Example configurations include changing simulation granularity, or deploying a car vs a drone simulation.

Additionally, new controller ONNX models can be trained using the provided dataset and evaluated using the provided `drone_test` executable.