Towards an Integrated Software Development Environment for Robotic Applications in MPSoCs with Support for Energy Estimations

Paulo H. Vancin[1], Anderson R. P. Domingues[1], Marcelo Paravisi[1,2], Sergio F. Johann[1], Ney L. V. Calazans[1] Alexandre M. Amory[1]

[1] *School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS –* Porto Alegre, Brazil

[2] *Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul – IFRS –* Osório, Brazil

{paulo.vancin, anderson.domingues, marcelo.paravisi}@acad.pucrs.br, {sergio.filho, ney.calazans, alexandre.amory}@pucrs.br

***Abstract*—Multi-processor Systems-on-Chip (MPSoCs) have been proposed to tackle embedded systems' requirements due to their potential for low-power consumption and high scalability. These systems fit the needs of many application domains, including robotics and autonomous vehicles, in which reliability, performance, and timeliness are critical to operation. In this paper, we propose an integrated environment for the development of robotic applications targeting MPSoCs. The proposed environment eases the evaluation of non-functional requirements by combining cycle-accurate simulations from RTL models with behavioral simulations from robotics. We present a case study of the proposed environment in the context of a UAV (unmanned aerial vehicle) stabilization software, providing performance and energy estimations for different software implementations.**

***Index Terms*—Autonomous Vehicles, MPSoCs, Robotics, Software Development Environment, UAV.**

I. Introduction

The increasing interest in robotics and autonomous vehicles applications such as autonomous cars and delivery drones require the development of robust and reliable software. Today's robotic applications demand typical embedded systems' requirements, such as low power consumption, real-time scheduling, high reliability and safety. These characteristics, along with requirements of other domains such as machine learning and computer vision, make the development of software for robotics both time-consuming and error-prone.

Hardware providers have proposed specialized computing units to fulfill the requirements of particular domains, such as graphical processing units (GPUs) for computer vision, vector units for numerical computing, and neural processing units (NPU) for machine learning. However, these units explore data-level parallelism, corresponding to only a fraction of the tasks in typical robotics systems. In this scenario, multi-processor systems-on-chip (MPSoCs) have evolved to supply the demand for heterogeneous, massively parallel, computing-intensive and energy-efficient applications.

Typical MPSoC architectures are composed by tens to hundreds of embedded processors connected to a network-on-chip (NoC), predominantly relying on a distributed memory programming model. We argue that the domain of robotics could benefit from MPSoCs due to three main reasons: (i) mesh-based topology NoCs based on NORMA (non-remote memory access) tends to be more scalable than bus-based architectures, as opposed to shared-memory multi-core architectures; (ii) larger systems favor the exploration of task mapping, reducing communication delay and energy consumption, and improving system performance; and (iii) MPSoCs ease the exploration of task-level parallelism when compared to vector units, GPUs, and NPUs, as these architectures aim for data-level parallelism.

In this paper, we present an integrated software development environment to design robotic applications for MPSoCs. The goal of the proposed environment is to integrate multiple simulators in a unique setup that covers physics, hardware, and software simulation of robots and vehicles. The environment will enable the evaluation of non-functional requirements such as energy/power consumption [2], and response time for robotics applications. Validation of the proposed environment is performed with an unmanned aerial vehicle (UAV) application in a quadrotor. This application controls the height and attitude of the quadrotor in a simulated MPSoC platform. The application consists of two tasks: (i) EKF (extended Kalman filter), which performs sensor fusion and vehicle pose estimation, and (ii) PID (proportional integral derivative), which controls the four motors of the quadrotor to perform stabilization.

II. Background

The Robot Operating System (ROS)[1] became a *de facto* standard for the development of robotic systems. ROS-based systems implement the publish-subscribe pattern [4], whose organization consists of nodes and topics. A node is a software that can subscribe to (or publish to) data channels called topics. Typical setups for robot simulation combine ROS with Gazebo[2], a robot simulator capable of simulating environmental physics, e.g., gravity, wind, terrain, and sunlight.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior – Brasil (CAPES) – Finance Code 001 and project 88887.115590/2015-01, Pro-Alertas program.

[1]https://www.ros.org/

[2]http://gazebosim.org/

978-1-7281-3320-1/20/$31.00 ©2020 IEEE

Robots interact with the environment through sensors, which capture data based on the stimuli of the simulated environment. Gazebo can be extended to push data into ROS, so that applications can access sensing information. Then, the behavior of the robot can be programmed and validated as if the software were deployed in a real robotic platform.

III. Proposed Development Environment

As far as we are concerned, typical Gazebo-ROS setups consist of physics simulation (provided by Gazebo) and application (usually programmed with ROS). In contrast, the proposed environment provides a cycle-accurate MPSoC model to support the development of the robotic application. The environment shown in Figure 1 consists of four parts; (i) physics simulation in Gazebo (Sec. III-A), including environment and robot's sensors and actuators, (ii) hardware simulation (Sec. III-B), consisting of an MPSoC platform, (iii) software for the before-mentioned robotics application (Sec. III-C), and (iv) the message interface between ROS and MPSoC (Sec. III-D).

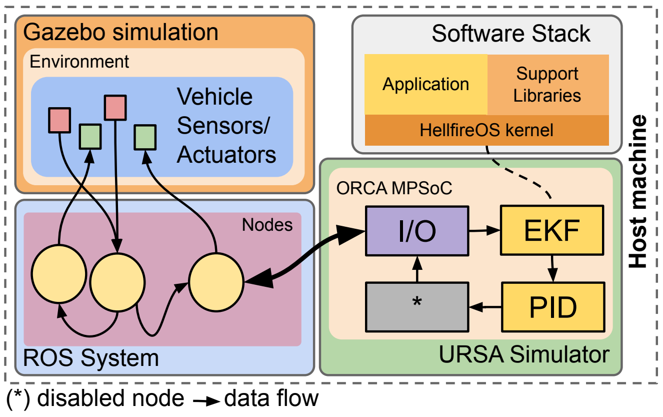

Figure 1. Organization of the proposed development environment consisting of Gazebo, a custom ROS-based system, and a simulated MPSoC.

A. Environment and Robot Simulation

Gazebo performs physics simulation, i.e., the environment and the robot. With the aid of GAZEBO/ROS[3] plugins, Gazebo populates specific topics of an ROS-based system; these topics maps to sensors the simulated robot. In the case of our control application, these topics correspond to data from accelerometer, magnetometer, barometer, and gyroscope sensors. Similarly, motors of the quadrotor can be accessed through topics. The behavior of sensors is implemented by the HECTOR_QUADROTOR [5] package, which models aerodynamics, propulsion, and sensors within Gazebo. It is important to note that we slightly modified [4] the package to bypass the internal controller. This modification was necessary to access unfiltered sensing data from outside Gazebo, removing the generation of noise in the output.

[3]http://wiki.ros.org/gazebo_plugins
[4]https://github.com/marceloparavisi/hector_quadrotor.

B. MPSoC Hardware

URSA [6], a framework that is used to develop cycle-accurate simulators based on the discrete-event simulation (DES) model, handles the simulation of the MPSoC architecture. URSA provides an API to describe hardware models as finite state machines (FSMs) written in the C++ language. The simulated MPSoC platform, called ORCA, is modeled as a 2D-mesh Network-on-Chip (NoC) combined with processing elements.

ORCA allows for the design-time configuration of multiple MPSoC parameters, including the number of nodes, router buffer width, and processor memory size. The configuration adopted in this paper has a message size of 64 16-bit flits. Communication is carried out by an instance of the Hermes NoC [3], with a buffer depth of 16 flits, XY routing algorithm, and wormhole packet switching. We organized tiles in a 2x2 mesh-based topology. One tile is reserved to handle I/O operations, and two of the remaining tiles run the tasks. In this configuration, the fourth tile is powered off due to application requirements. Each processing tile consists of a 32-bit HF-RISCV processor core [7], with a 3-stage pipeline and RV32IM instruction set. Processing tiles, shown in Figure 2, include 2MB of SRAM each, a custom network interface to attach the processor core to the NoC, and two auxiliary buffers for incoming (Recv. Mem.) and outgoing packets (Send Mem.).

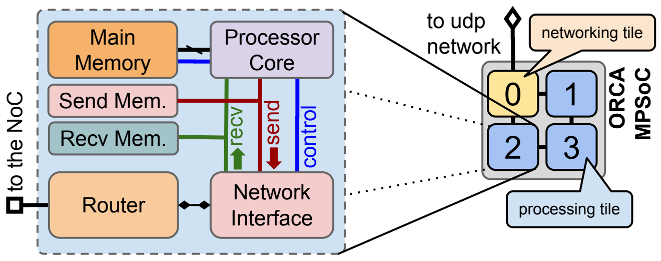

Figure 2. Illustration of the simulated MPSoC and internal hardware.

C. MPSoC Software

The application model adopted in this work includes a set of real-time tasks executing over a NORMA architecture, communicating through a message passing mechanism. Task scheduling and message passing mechanisms are implemented on top of the HellfireOS [8], a real-time operating system (RTOS) that is fully preemptive and highly configurable. The RTOS provides task management and scheduling policies, along with standard libraries, synchronization primitives, and inter-task communication drivers. Additionally, routines for fixed-point operations and matrices manipulation used in this work are implemented as libraries on the HellfireOS.

D. Interface Between MPSoC and Robot Simulation

In our environment, we use ROS nodes to carry messages from the MPSoC platform to Gazebo and vice-versa. Two ROS nodes, `to-mpsoc` and `to-gazebo`, wrap up the network protocols. From the ROS point of view, the MPSoC is an ordinary ROS node, as if all the logic of applications reside in the to-mpsoc node. Similarly, from the MPSoC perspective, peripherals (sensors and actuators) are connected to the NoC. Thus, the to-gazebo node has to handle all necessary peripheral-specific protocol translation. For this application, all sensors are updated at the same frequency, and all sensing data is packed in the same data structure. Sensors in Gazebo update at 100Hz, which is faster than the maximum frequency supported by the MPSoC at the moment, 10Hz, as presented in the results. The to-mpsoc node adds latency compensation to sensing data, packing multiple sensor readings in the same packet. On the to-gazebo side, the node uses interpolation to fill the gaps generated by the latency compensation. It is important to note that this approach permits multiple instances of the MPSoC to be associated with multiple robots in Gazebo, enabling the simulation of complex scenarios.

Figure 3 illustrates the communication between the system components. Sensing data depart from Gazebo, and become accessible in the ROS system through three topics (B, U, and M in Figure 3), to which the to-mpsoc node subscribes to. Since data arrive in these topics at the same rate, we pack them all in the same data structure and then send the structure through the UDP network to the MPSoC simulator. Once the data arrive at the I/O interface, it traverses the simulated network-on-chip until they reach the target processing node. The task responsible for the reception of data from the ROS system is EKF. The estimated pose from EKF feeds the PID controller, whose output is transferred through the UDP network to the to-gazebo node. After the PID generates its output, the PWM vector is forwarded directly to the motors of the quadrotor in Gazebo.

IV. Flight Stabilization Application

The quadrotor flight stabilization application consists of two tasks. First, a quaternion-based EKF performs estimation of the altitude and attitude of the vehicle. The orientation information (quaternions) is converted to Euler angles (roll, pitch, and yaw) and fed to another task, a PID controller. The PID controller, shown in Figure 4, forces the error between the estimated states and the desired states (setpoint) to approximate zero.

A. Extended Kalman Filter (EKF)

The assumptions of linear state transitions and linear measurements with added Gaussian noise are rarely fulfilled in practice. Linear next state transitions cannot describe most of the robotic vehicles. Then, simple Kalman filters do not apply to most trivial robotics problems. In this work, the Extended Kalman Filter (EKF, shown in Algorithm 1) is used instead, since it allows for nonlinear equations.

In the EKF algorithm, $x_k \in \mathbb{R}^{7\times 1}$ represents the states of the system (orientation quaternion and gyroscope covariances), $z_k \in \mathbb{R}^{4\times 1}$ are the sensor inputs, $P_k \in \mathbb{R}^{7\times 7}$ is the covariance matrix of the states of the system, $F_k \in \mathbb{R}^{7\times 7}$ is the Jacobean matrix of system's function $f(x_{k-1}) \in \mathbb{R}^{7\times 1}$, $H_k \in \mathbb{R}^{4\times 7}$ is the linearization of the function for the sensor models $h_k \in \mathbb{R}^{4\times 1}$, $K_k \in \mathbb{R}^{7\times 4}$ is the correction gain matrix, $R \in \mathbb{R}^{4\times 4}$ and $Q \in \mathbb{R}^{7\times 7}$ are the covariance matrices for uncertainties of sensors, and system's model, respectively.

Algorithm 1 EKF Algorithm (x_{k-1}, P_{k-1}, z_k)

```
1: $\overline{x}_k \leftarrow f(x_{k-1})$
2: $\overline{P}_k \leftarrow F_k P_{k-1} F_k' + R$
3: $K_k \leftarrow \overline{P}_k H_k' (H_k \overline{P}_k H_k' + Q)^{-1}$
4: $x_k \leftarrow \overline{x}_k + K_k (z_k - h(\overline{x}_k))$
5: $P_k \leftarrow (I - K_k H_k) \overline{P}_k$
6: return $P_k, x_k$
```

B. PID

As previously stated, the PID controller is used to find the control signal (u_i) for all of the states that guarantees the desired orientation (roll α_{sp}, pitch β_{sp}, yaw θ_{sp}) and height (z) of the vehicle. The PID controller for the roll parameter is described in Equation 1, where K_p is the proportional gain, K_i is the integral gain, K_d is the derivative gain, and α_{sp} is the setpoint for the roll parameter. The same applies for pitch and yaw parameters. The height control also uses a modified PID controller [1], shown in Equation 2, where $T_{gravity}$ is the minimum thrust that overcomes the gravity force. The aforementioned control laws are used as components for the signal output for each of the four motors. Equation 3 shows the control signals of motors for the dynamics of the quadrotor.

$$u_\alpha = K_p(\alpha_{sp} - \alpha) + K_i \int_0^t (\alpha_{sp} - \alpha) dt + K_d \frac{d}{dt}(\alpha_{sp} - \alpha); \quad (1)$$

$$u_z = \frac{1}{\cos\alpha\cos\beta}\Big(K_p(z_{sp} - z) + K_i \int_0^t (z_{sp} - z) dt + K_d \frac{d}{dt}(z_{sp} - z) + T_{gravity}\Big); \quad (2)$$

$$\begin{array}{ll} u_1 = -u_\beta + u_\theta + u_z & u_2 = -u_\alpha - u_\theta + u_z \\ u_3 = u_\beta + u_\theta + u_z & u_4 = u_\alpha - u_\theta + u_z \end{array} \quad (3)$$

V. Energy Estimation and Preliminary Results

The performance of the proposed environment is bound to the host machine. In this paper, we used a DELL Precision Tower 3420 Workstation (Intel Xeon E3-1220 v5 @3GHz, 4x core, 32GB of RAM, 1TB SSD hard-disk) as host machine, with ROS (Kinect Kame) and Gazebo (version 7.0.0) running under Ubuntu 16.04.6 Desktop (x64). The control application (EKF and PID) was written in C (cross-compiler GCC/G++ version 8.1.0 targeting RISCV32). Models implemented by the PID and EKF tasks were developed using Matlab and Python and validated before their port to the platform. In the experiments, we executed the MPSoC platform running the EKF plus PID application and calculated the processing time and energy necessary to run a complete iteration of EKF and PID, respectively. This means that every sensor message received into the MPSoC triggers a complete EKF and PID iteration to update the UAV actuators.

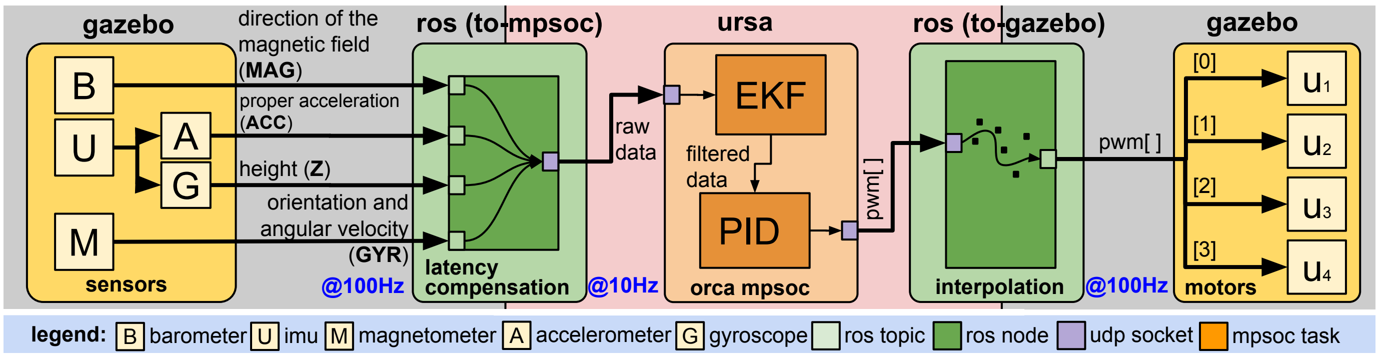
gazebo
direction of the magnetic field (MAG)
ros (to mpsoc)
ursa
ros (to gazebo)
gazebo
B
U
A
G
M
sensors
proper acceleration (ACC)
height (Z)
orientation and angular velocity (GYR)
@100Hz
latency compensation
@10Hz
raw data
EKF
filtered data
PID
orca mpsoc
pwm[ ]
interpolation
@100Hz
pwm[ ]
[0]
[1]
[2]
[3]
u1
u2
u3
u4
motors
legend: B barometer U imu M magnetometer A accelerometer G gyroscope ros topic ros node udp socket mpsoc task

Figure 3. Diagram representing the dataflow between the components of the environment.

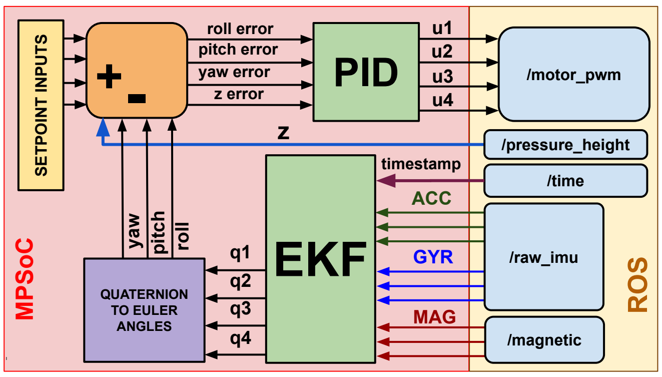

Figure 4. Illustration of the control scheme.

The proposed MPSoC model can estimate the energy required by a task. The average energy consumption of every processor instruction was previously characterized using the method described by Martins [2]. We compared four different application setups to determine the best configuration, considering energy consumption and response time for the task set. For both tasks, we ran variations of the environment using a software-emulated floating-point (SEFP), and a hardware multiplier unit (MU). There were four variations: (i) system with SEFP only, (ii) SEFP+MU, (iii) MU only, and (iv) none. For each variation, we collected statistics for 100 iterations of each task. The fixed-point data is represented in 16.16 format (16 bit integral, 16 bit fractional parts). The response time results assume a clock period of 4 ns.

Results, shown in Table I, indicate that MU-only configuration presented the best average response time and average energy consumption, where less is better for both criteria, for both tasks. The system with fixed-point arithmetic without MU performed closer to the SEFP+MU configuration, indicating that those choices are nearly equivalent when considering only average response time and average energy as criteria. For both applications, fixed-point without MU is worse than SEFP+MU in response time, although SEFP+MU outperforms fixed-point without MU in energy consumption.

For the numerical quality of results, we compared the output of EKF and PID tasks against the reference models in Matlab,

Table I
RESPONSE TIME AND ENERGY EVALUATION FOR PID AND EKF TASKS.

	Task	Configuration			
		SEFP		fixed point arith.	
		with MU	no MU	with MU	no MU
Avg. Response Time (ms)	PID	25.5	34.0	20.0	27.0
	EKF	107.01	196.5	87.5	173.0
Avg. Energy (nJ)	PID	1781.1	2517.4	1480.0	1776.7
	EKF	7252.7	13761.6	6460.3	11776.9

as well as against the built-in controller of the Hector package. As a result, we achieved a mean error between fixed point and floating point of 0.19%, that is, each component of the resulting quaternion has a mean value error from its reference of 0.0019. When comparing the output of the EKF task to the real pose estimated by the internal controller of Hector, the mean error was of 0.44%.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a software development environment for robotic applications for MPSoCs, integrating several tools to simulate computing architecture and physics. The environment has been evaluated for a robotic application in a quadrotor, in which an EKF and a PID controller have been developed to achieve attitude and height stabilization. Results demonstrated that the proposed environment allows a quick evaluation of performance and energy trade-offs of different software implementations.

Future works for the project include (i) adding more control methods and operation modes to the application, (ii) redoing the energy evaluation, considering other tasks and the operating system overhead, (iii) synchronize Gazebo with the MPSoC to eliminate rate compensation, (iv) prototype the MPSoC in a FPGA attached to a real quadrotor.

REFERENCES

[1] Huang, H. et al. "Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering." In: IEEE International Conference on Robotics and Automation (ICRA). pp. pp. 3277–3282. 2009.

[2] Martins, A. L. D. M. "Multi-Objective Resource Management for Many-Core Systems". PhD Thesis. PPGCC - Pontifícia Universidade Católica do Rio Grande do Sul (2018).

[3] Moraes F. et al.“HERMES: an infrastructure for low area overhead packet-switching networks on chip.” Integration the VLSI Journal, Vol. 38, no. 1 (2004).

[4] Colouris, G. et al., “Distributed Systems: Concepts and Design”. 5th Ed. Addison-Wesley Publishing Company, USA (2001). pp. 242 – 250.

[5] Meyer J. et al. “Comprehensive Simulation of Quadrotor UAVs using ROS and Gazebo”. In: International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR). pp. 400-411. 2012.

[6] Domingues, A. “URSA: A framework to the simulation of multiprocessor platforms”. Online. Accessed October 7, 2019. Available at https://github.com/andersondomingues/ursa.

[7] Filho, S. J. “HF-RISC SoC”. Online. Accessed October 7, 2019. Available at https://github.com/sjohann81/hf-risc.

[8] Filho, S. J. “HellfireOS: Realtime Operating System”. Online. Accessed October 7, 2019. Available at https://github.com/sjohann81/hellfireos.