Integrating an MPSoC to a Robotics Environment

Anderson R. P. Domingues
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brasil
anderson.domingues@acad.pucrs.br

ABSTRACT

Multiprocessor system on-chip (MPSoC) have been successfully used to speed up the processing of parallel, computing intensive applications. Many domains could benefit from such processing power, including the one of robotics, in which several tasks, from image processing to odometry, may compete for resources at the same time as they cooperate for the performance of the system. This paper present the results on an initial effort on integrating an MPSoC into robotic system. The integration is done with the aid of the robot operating system (ROS) middleware. We developed an special ROS node capable of communicating with the MPSoC using of UDP protocol. Also, we present a proof of concept to validate this configuration based on a synthetic random-walk application for the robotics domain.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; *External interfaces for robotics.*

KEYWORDS

MPSoC, ManyCore, ROS, Robotics

ACM Reference Format:
Anderson R. P. Domingues. 2019. Integrating an MPSoC to a Robotics Environment. In *Proceedings of (HPDC'19).* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/yy.yyy/yyy_y

1 INTRODUCTION AND BACKGROUND

In the last two decades, multiprocessor system-on-chips (MPSoC) have been used to accelerate the processing of massively-parallel, computing-intensive applications, such as those in image and video encoding [7, 19, 21] and network processing [19] domains. As a result, a couple of general purpose MPSoC were released into the market [12, 20], providing access to a variety of intellectual properties (IP) as well as allowing even more domains to benefit from the computational power offered by these platforms.

Although MPSoCs provide a considerable computational power at low power consumption cost, there are few flaws inherent to these platforms. First, parallel application are more difficult to design and implement when compared to sequential ones, which applies not only to MPSoCs, but for all multiprocessor platforms. Of course, there are solutions to these problems already [8, 11], but there is also a cost to port these solution from one MPSoC technology to another. Another important concern about MPSoCs is that the integration of MPSoCs with other systems rely on standard communication protocols that requires data to be packed (and unpacked) to be sent (and received) though the communication media. For the later, the use of serialization libraries [9] mitigate the problem, while adding some complexity to the software.

For the domain of robotics, specifically, there are a few solution to the problem of integrating several components into a single system [1, 4, 17]. These solutions usually rely on a middleware for converting data and perform system-level services, such as the abstraction of the physical parts of the system into logical ones. To the best of our knowledge, the Robot Operating System (ROS) [17] is the most notable solution, as it provides hundreds of packages for dealing with different platforms, including simulation engines commonly used within the robotics domain. However, even ROS does not provide a solution for the partially deploying of robotics applications in MPSoC at the communication level. ‘ In this paper we present a setup for integrating an MPSoC in a robotic system. In our setup we considered a specialized hardware for converting network on-chip based packets into UDP/IP packet, an access method for the MPSoC from the inside a ROS-based application and an example of use of our approach. For the rest of this section we give a little introduction to ROS communication architecture and briefly discuss the motivation for including an MPSoC into the setup. The chosen MPSoC platform is discussed in Section 2, as well as the hardware that translates packets across the ROS application and the MPSoC software stacks. In Section 2.2 we describe the application we used to validate our approach. Although simple, the application suffices the requirements for the proposed approach, which basically requires a partitioned application to run at least one of their tasks into the MPSoC, while the remaining tasks resides outside the MPSoC. Lastly, we present our final consideration and discuss some points to be explored in future, as well as the limitations of our approach.

1.1 ROS - The Robot Operating System

The Robot Operating System (ROS) [17] became the standard middleware for integrating distributed systems in robotics. One of the most beneficial features in ROS is the one of abstracting parts of the system as nodes. By implementing the publish-subscribe pattern [5], ROS allows the many participating nodes of a system to be deployed in different platforms. These platforms use of a common network media (usually over TCP or UDP sockets) to publish and subscribe to topics. While a topic is a channel of communication for a certain information (e.g., odometry, camera feed, bumper collision), publisher nodes produce such information and subscribers nodes consume the information. For instance, the motion system

HPDC'19, June 2019, Phoenix, Arizona, USA

ACM ISBN xxx-xxx-xx-xxx/xx/xx.
https://doi.org/yy.yyy/yyy_y

of a robot may publish odometry data on a topic, while the simultaneous localization and mapping system (SLAM) [2] can subscribe to the same topic to receive updates on that data. It is important to note that, depending on their function, nodes can be both publishers and subscribers to many topics at the same time, enabling the complex organizations of robotic system.

Since the many nodes of a system may run in different platforms (e.g., smart phones, laptops, cloud), it is usual to run critical nodes on dedicated high-performance platforms (e.g., CPU, GPU) and low-priority nodes on best-effort, shared platforms (e.g., general purpose computers) [13]. For example, the image processing node of an arbitrary robot system may run on a dedicated board with GPU and multiple co-processing support, isolated from the rest of the nodes, which run, let us suppose, on a shared, single-threaded platform. The benefits of distributing nodes over multiple platforms include (i) reducing the communication latency by allocating communicating nodes as much as close as possible, (ii) reducing power consumption by sharing system resources among multiple nodes and (iii) using of platform-specific features to speed the processing of certain applications. For the later, GPU and MPSoC are both alternatives to tackle the processing of homogeneous and heterogeneous parallel applications, respectively.

2 MPSOC PLATFORM

The target MPSoC platform comprise several tiles interconnected by a network on-chip (NoC), based on the existing Hermes network [14]. One of these tiles is a network bridge that translates packets from UDP/IP networks to the network on-chip network protocol (referred to as *network bridge* for the rest of the paper). The remaining tiles are processing elements (PE) that carries the computation of specific heterogeneous parallel applications by providing computation services to other nodes in the system. All communication the between the MPSoC and the external network is held by the network bridge, which serves as a gateway to the system. Figure 1 shows an overview of the MPSoC organization.

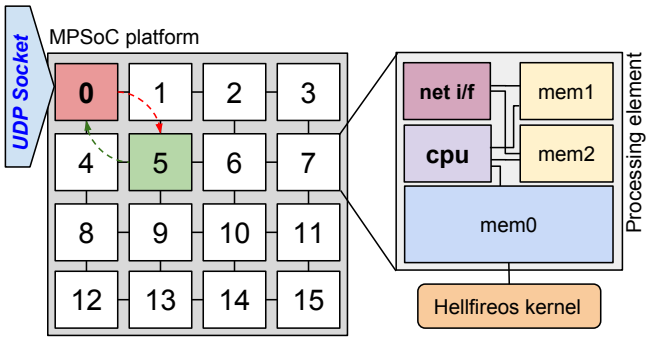

Figure 1: A 4x4 MPSoC platform and their 16 tiles. Tile zero is the network bridge, while the rest of the tiles are processing elements consisting of memory modules, cpu, network interface and router.

Each of the PE runs a copy of the HellfireOS kernel [6], deployed to the main memory ram module (`mem0`) at startup. Mem0 also contains the applications and the driver that provide access to the network interface (NI) to the kernel. The NI allows the CPU to simultaneously send and receive packets form the underlying NoC. Finally, two memory modules are attached to the CPU and the NI at the same time. When a packet is received from the NoC, the NI copies its contents from the router's buffer into `mem1` and interrupt the CPU to consume received data. The sending process follow pretty much the same process by sending data from `mem2` to the NoC. Data in `mem2` must be previously copied from `mem0` by the CPU, which also interrupts the NI to start the sending process.

2.1 A hardware for translating NoC flits to UDP packets and vice-versa

To connect our MPSoC to other networks we developed an special hardware model capable of translating packets from an UDP network to `flits`, which are the data units used to transmit information over the NoC. A flit has a 16-bit length and a `message` has 64 flits, which means that the minimum amount of data that can be transmitted over the NoC matches 128 bytes. The translation of the UDP packet into a message includes removing any protocol header up to the UDP protocol (UDP, IP and beyond) and shrink the data to fit into the 128 bytes of a message while considering the headers of the NoC protocol (16 bytes of header, plus 112 bytes of payload).

Since our MPSoC is virtual (i.e. an application is emulating the underlying hardware), we decide to use the network subsystem of the host machine to manage the communication between the host machine and the MPSoC. Basically, the network bridge awaits for the interruption from the network interface. Once the interruption is triggered, the network bridge sends the raw data (all 64 flits) through an UDP socket. At the opposite side is the `mpsoc-ros` node, listening at some UDP port that matches the client in the network bridge. Once the packets reach the UDP network, this node carries out the data treatment so that the payload can be used within the application.

2.2 MPSoC-ROS Node

The mpsoc-ros node executes two simultaneous threads that treats incoming and outcoming messages from and to the MPSoC bridge. For the former, all data arriving at the `mpsoc-in` topic is pushed into the network bridge by adding the proper headers to the payload (NoC headers) and sending to the bridge UDP server. Another thread has an UDP server listening to some port, which is expected to receive the response from the network bridge and publish the data into the `mpsoc-out` topic. It is possible to add more mpsoc-ros nodes to the system if necessary, as long as they listen to different UDP ports, up to the limit of the host system. By adding such nodes we expect to several nodes in ROS to use of the MPSoC resources, similarly to remove service. There is also a limitation that impose mpsoc-ros nodes to have messages of a certain type, since mpsoc-in and mpsoc-out must specify they data structures at compilation time and prevents a single instance of a mpsoc-ros node to treat multiple data type. Otherwise it would be possible to have only one node per system, although it does not seem to be a good practice though. Figure 2 illustrate the mpsoc-node behaviour and the data flow in the system.

The mpsoc-ros nodes can be programmed to treat data as well as only sending and receiving raw data. For instance, the application in

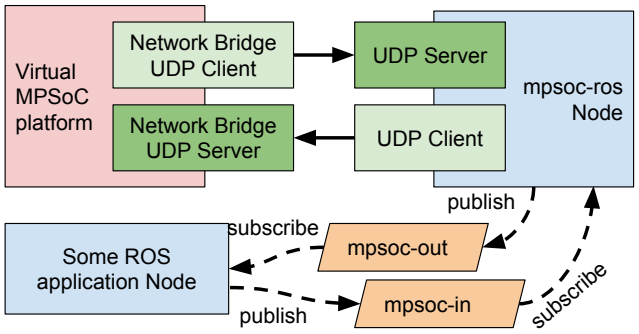

Figure 2: A general organization of the MPSoC integration into a ROS network using the mpsoc-ros node. The virtual platform connect to the msoc-ros node through UDP clients and servers. The mpsoc-node exchanges data with the rest of the system by publishing to the mpsoc-out topic, as well as subscribing to the mpsoc-in topic.

Section 2.2 required the conversion between different endianess. At one side, the emulated RiscV processors from the MPSoC platform use little-endian encoding, and the Xeon processors in host machine used big-endian. Other data treatment such as compression and seralization can be carried out in mpsoc-ros nodes as well.

3 EXAMPLE OF USE: RANDOM WALK APPLICATION

The purpose of this application is to demonstrate the basics on the integration between the MPSoC platform and the robot systems, regardless of performance issues. This application implements a random-walk mechanism based on ranges captured from a laser rangefinder Hokuyo URG-04LX-UG01 [10], attached to a custom differential robot in a simulated scenario (see Figure 3). The simulation is held by Gazebo 9 simulator [15] using of gazebo-ros [16] integration package, as well as our custom integration package that similarly couple our MPSoC to the ROS environment. An overview of the setup for the random-walk application is shown by Figure 4.

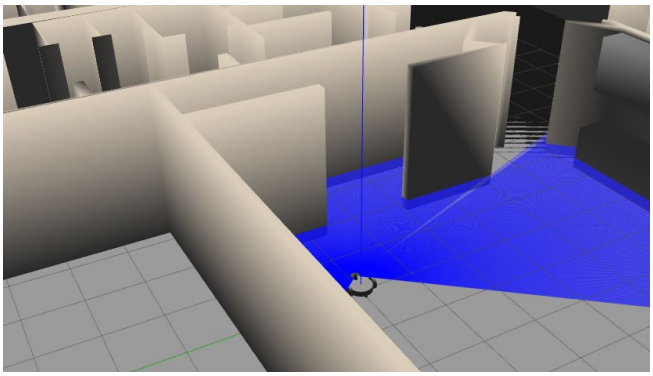

Figure 3: Screenshot of the simulated environment. The robot has a rangefinder attached to the top. The ranges captured by the rangefinder are shown in blue. The simulation is carried out in Gazebo 9.

The random-walk algorithm is implemented part by the random-walk application node (RWAN), part by the MPSoC application. For the sake of simplicity, only one core of the MPSoC (core 5 in the example) is being used. This core computes the maximum value among several ranges received from the RWAN through the mpsoc-ros node. The computed maximum value is send back to the RWAN, which command the robot (in Gazebo) to turn in order to match the computed angle. The operation repeats over and over in intervals of 2 seconds, making the robot to walk randomly while avoiding walls when possible.

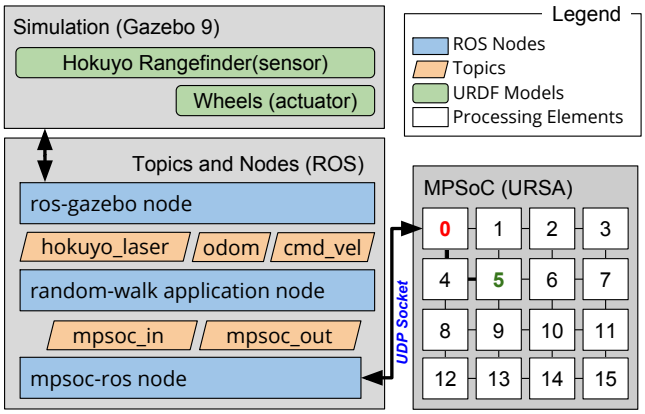

Figure 4: Setup for the random-walk application containing Gazebo, ROS nodes and the MPSoC platform. ROS nodes communicate with each other through topics, while special nodes (ros-gazebo and mpsoc-ros) handle the communication with simulated platforms.

The application part running into the MPSoC computes the maximum value by comparing every 53 incoming ranges against the maximum value acquired at the moment. When the first range arrives, it becomes the maximum computed value. For the rest of the ranges, the value is compared to the current maximum value and, in case the received range is greater than the current maximum value, the application replaces the old value with the new one. Once the 53^{th} range arrives, the MPSoC send the stored maximum value back to the ROS part of the application. The value of 53 can be arbitrarily changed to any value less than the number of ranges captured by Hokuyo's rangefinder (which is 624). Please note that the number a higher number of ranges may compromise the performance of the system or ocasionally crash the application, which depends on the throughput of the loopback interface.

4 METRICS AND EVALUATION

In this section we present some performance evaluation on the mpsoc-ros node. This evaluation is important due that application from the domain are usually communication intensive, requiring high throughput of data. When adding the mpsoc-ros node to the system, we also add complexity, since we need to add and remove header from packets, as well as converting endianess, compressing, serializing, among other operations. Although some of these operations are optional, one cannot avoid to add some delay on

the communication when comparing the integrated MPSoC with ROS node running in the same system. However, the benefits of adopting the proposed integration rely on the fact that the running application has some potential for parallelism. Otherwise, a sequential application will always surpass the MPSoC, due the absence of communication between processes.

4.1 Simulation speed

The simulation engine being used to emulate our MPSoC is URSA [3], which relies on discrete event simulation [18]. By doing so, hardware models are described as finite state machines (FSM), in which the simulation time does not depend on the time taken by a model to change states. The amount of time spent to run the transition function is linear to the number simulated models of the same kind. For instance, it takes twice the time to simulate two CPU when compared to the simulation of a single CPU.

Table 1: Collected metrics for the several running experiments using different configurations of MPSoC.

Simul. Cycles	Avg. Time (s)	Config.	Avg. Freq. (Hz)
1,000,000	41.50	15 PE, 1 NB	24,096
1,000,000	2.10	8 PE, 1 NB	476,190
1,000,000	0.50	3 PE, 1 NB	1,250,000
1,000,000	0.12	1 NB	8,333,334
1,000,000	0.16	1 PE	6,250,000
1,000,000	0.38	2 PE	2,631,578
1,000,000	0.56	3 PE	1,785,714
1,000,000	0.83	4 PE	1,204,819
1,000,000	1.20	5 PE	833,334

PE: processing element, NB: network bridge

We ran some experiments so that we could tune the platform properly to work with the random-walk application. The objective has to find a proper rate to send messages from the ROS application node to MPSoC so that all data could be processed, without dropping any of the packages or congesting the platform' network.

$$Rate = \frac{Frequency}{(Time_t + 2 \times Time_p)} \quad (1)$$

We obtained the frequency for several MPSoC configurations (see Table 1) and estimated the rate by replacing values in Equation 1, where $Frequency$ is the value got from experiments, $Time_t$ is twice the time to transmit a message, and $Time_p$ is the time the PE takes to process the data and interrupt the network interface. The sum of $Time_t$ and $2 \times Time_p$ represent the total amount of cycles that a message takes to enter the MPSoC network, being processed and return to the external network, discarding processing from mpsoc-ros. The value of $Time_t$ being used is 64, plus the Manhattan distance of the network bridge to the processing element, while $Time_p$ depends on the instructions being executed into the processor.

5 CONCLUSION AND FUTURE WORK

We present an integration of an MPSoC platform to a robotic application, in which a special ROS node is used to connect to the MPSoC through the UDP protocol. Also, we present an example application to validate our approach. Although we could run the application using of our proposed infrastructure, there are a few enhancements necessary to the support complex applications in the future. First, implement some task partitioning method to reduce the network throughput and thus increase the allowed frequency. Also, we intend to run more tests on a version of the platform in a FPGA plaform.

REFERENCES

[1] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. 2005. RT-middleware: distributed component middleware for RT (robot technology). In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 3933–3938.

[2] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. 2001. A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation* 17, 3 (June 2001), 229–241.

[3] Anderson R. P. Domingues. 2018. andersondomingues/URSA : An environment or simulating multiprocessed platforms. https://github.com/andersondomingues/ursa/

[4] Brian Gerkey et al. 2014. The Player Project. http://playerstage.sourceforge.net/

[5] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131.

[6] Sergio Johann Filho. 2012. *Suporte para aplicações dinâmicas em sistemas multiprocessados intra-chip homogêneos*. Ph.D. Dissertation. Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil.

[7] F. Gharsalli, A. Baghdadi, M. Bonaciu, G. Majauskas, W. Cesario, and A. A. Jerraya. 2004. An efficient architecture for the implementation of message passing programming model on massive multiprocessor. In *Proceedings. 15th IEEE International Workshop on Rapid System Prototyping, 2004.* 80–87.

[8] J. C. Hamerski, G. Abich, R. Reis, L. Ost, and A. Amory. 2017. Publish-subscribe programming for a NoC-based multiprocessor system-on-chip. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4.

[9] Jean Carlo Hamerski, Anderson Domingues, Fernando Gehm Moraes, and Alexandre Amory. 2018. Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs. In *28th International Conference on Eletronics and Systems*. [to be published], Bordeaux, France.

[10] LTD. Hokuyo Automatic CO. 2018. Scanning Rangefinder Distance Data Output/URG-04LX-UG01 Product Detail. https://www.hokuyo-aut.jp/search/single.php?serial=166

[11] Chunsheng Li, Xi Li, Chao Wang, Xuehai Zhou, and Fangling Zeng. 2012. A Dependency Aware Task Partitioning and Scheduling Algorithm for Hardware-Software Codesign on MPSoCs. In *Algorithms and Architectures for Parallel Processing*, Yang Xiang, Ivan Stojmenovic, Bernady O. Apduhan, Guojun Wang, Koji Nakano, and Albert Zomaya (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 332–346.

[12] Mellanox. 2018. Mellanox Products: BlueField (TM) Multicore System On A Chip (SOC) for NVMe storage. http://www.mellanox.com/page/products_dyn?product_family=256&mtag=soc_overview

[13] S. A. Miratabzadeh, N. Gallardo, N. Gamez, K. Haradi, A. R. Puthussery, P. Rad, and M. Jamshidi. 2016. Cloud robotics: A software architecture: For heterogeneous large-scale autonomous robots. In *2016 World Automation Congress (WAC)*. 1–6.

[14] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. 2004. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration* 38, 1 (2004), 69 – 93. http://www.sciencedirect.com/science/article/pii/S0167926004000185

[15] Open Source Robotics Foundation (OFSR). 2014. Gazebo. http://gazebosim.org/

[16] Open Source Robotics Foundation (OFSR). 2018. gazebo_ros_packages - ROS Wiki. http://wiki.ros.org/gazebo_ros_pkgs

[17] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

[18] Gabriel A. Wainer. 2009. *Discrete-event modeling and simulation: a practitioner's approach* (1 ed.). CRC Press.

[19] W. Wolf. 2004. The future of multiprocessor systems-on-chips. In *Proceedings. 41st Design Automation Conference, 2004.* 681–685.

[20] Xilinx. 2018. SoCs, MPSoCs & RFSoCs. https://www.xilinx.com/products/silicon-devices/soc.html

[21] Jiang Xu, Wayne Wolf, Joerg Henkel, and Srimat Chakradhar. 2006. A Design Methodology for Application-specific Networks-on-chip. *ACM Trans. Embed. Comput. Syst.* 5, 2 (May 2006), 263–280.