Publish-Subscribe Programming for a NoC-based Multiprocessor System-on-Chip

Jean Carlo Hamerski*, Geancarlo Abich†, Ricardo Reis†, Luciano Ost‡, Alexandre Amory*
*FACIN - PUCRS - Av. Ipiranga, 6681, 90619-900, Porto Alegre, Brazil
†PPGC/PGMICRO Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
‡Department of Engineering - University of Leicester, Leicester, UK
jean.hamerski@acad.pucrs.br, { gabich,reis }@inf.ufrgs.br, luciano.ost@leicester.ac.uk, alexandre.amory@pucrs.br

***Abstract*—Shared memory and message passing are traditional parallel programming models used on multiprocessor system-on-chip environments. Underlying models are traditionally meant for static scenarios where all communicating entities and their intercommunication patterns are known a priori by the software engineer. The systems design following such programming models became complex due to dynamic behavior of applications at runtime. The goal of this work is to incorporate a publish-subscribe programming model to an MPSoC framework to decouple, in the time and space, the application development. The modified MPSoC framework is composed of a FreeRTOS kernel running on homogeneous processing elements distributed into a network-on-chip. The results present reduction around of 2% to 30% in DTW application execution time, and low overhead in memory footprint when comparing the original MPI primitives with the publish-subscribe programming model.**

***Index Terms*—MPSOC, Programming Model.**

I. Introduction and Related Works

MultiProcessor System-on-Chip (MPSoC) may integrate hundreds of processing elements (PEs) into a single chip, and its programming is based on shared or distributed memory models. The former model is easier to program (based on threads), but it is potentially less scalable. The second one is scalable, but the software development is more complex due to strong coupling and synchronization between communicating elements [1].

MPSoC programming frameworks have evolved in terms of functionality, but they are still based on the same main programing models. MPI-based frameworks have sofisticated quality of service (QoS) features, such as: Network-on-Chip (NoC) reconfiguration and flow prioritization [2]; virtualization [3]; multicast [4]. Threads-based frameworks present approaches to reduce the shared memory bottleneck [5] [6].

Concerning widely distributed embedded system, intra-chip or not, several authors argue that traditional programming models are not appropriate to deal with ever-increasing unpredictable and dynamic applications' behaviors [1] [7]. These models are based on static assumptions as the conventional communication and synchronization, usually defined in design time. Therefore, the applications adaptation is hard in a dynamic environment with unpredictable changes, such as failures or load fluctuations. Also, it is assumed that the nodes are on the same network at the same time, and that each node knows its communicating pair. The *publish-subscribe* (PUB-SUB) programming model has been used in middlewares for highly distributed domains, such as: MQTT[1] (Message Queuing Telemetry Transport) for sensors networks and mobile devices domains; DDS[2] (Data Distribution Service) for real-time systems domains; and ROS[3] (Robot Operating System) for robotics domains. All these middlewares evolved to provide properties, such as reliability, security, low power consumption, and QoS [8].

The *goal* of this work is to present a middleware based on the PUB-SUB programming model. The *main contribution* of this work is exploring the opportunity of developing a middleware based on the PUB-SUB programming model, which can be used to improve the programmability of distributed private memory NoC-based homogeneous MPSoCs. Developed middleware has been incorporated into FreeRTOS kernel embedded in an MPSoC platform. The proposed middleware allows: unicast/multicast-like communication, abstracting the NoC protocols and infrastructure; and decoupling in the time, space, and synchronization.

II. Publish-Subscribe Programming Model

In a PUB-SUB-based system, the participants (nodes) communicate with each other by exchanging messages. A *subscriber* node manifests interest in a particular data or event, identified by a *topic* (subscribe step). The node is notified when this *topic* is generated (publish step). The *publisher* node must register itself in the system (advertise step) as *topic* generator, so that future *subscribers* interested in this *topic* can receive notifications asynchronously. The *broker* node mediates the advertise, publish, and subscribe steps.

The application level decoupling occurs in three dimensions: (1) Space: a *subscriber* does not know who is the *topic publisher*, and a *publisher* does not know who consumes the *topic* generated by it; (2) Time: the *nodes* do not need to be active at the same time in the system; a *publisher* can generate *topics* while a *subscriber* is disconnected and vice-versa; (3) Synchronization: both *publishers* and *subscribers* are not blocked while they are generating or receiving *topics*; the *subscribers* are asynchronously notified when the *topic* of interest is received, and are processed via *callback function*.

[1] http://mqtt.org/documentation
[2] http://www.omg.org/spec/DDS/1.4/
[3] http://www.ros.org

978-1-4673-6853-7/17/$31.00 ©2017 IEEE

Fig. 1 shows a general scheme with two publishers, three subscribers, three topics, and one broker. A publisher can publish data in more than one topic. A subscriber can receive data about same or different topics from one or more publishers.

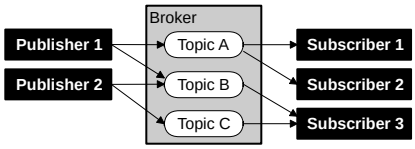

Fig. 1: General scheme of a publish-subscribe system.

In MPSoC domain, *publishers* nodes are those that want to produce data, while *subscribers* are nodes that want to consume them. The *topics* represent atomic data shared among the nodes. As an example, Fig. 2 shows the DTW (Dynamic Time Warping) application with ten tasks (Bank, P1-P8 workers, and Recognizer) [2] represented through a task graph. A directed arrow between two tasks (blocks in the figure) means that the first task sends data to the second one.

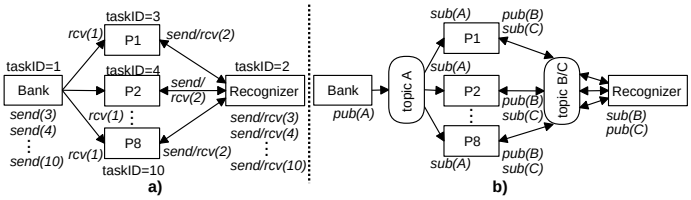

Fig. 2: DTW task graph in a) MPI and b) PUB-SUB models.

MPI typically uses the following primitives: *send(target to)* and *receive(target from)* primitives. The sender side needs to define explicitly the message target ID (represented by taskID), and the receiver side sets the message sender ID. The PUB-SUB version replaces these primitives by PUB-SUB primitives. The sender side must define a topic ID for the flow, register itself in the system as publisher of that topic and publish the data. The receiver side must register itself as a subscriber of that topic, setting a callback function to treat the incoming data.

III. The Proposed PUB-SUB MPSoC Framework

This section presents the adopted case study platform (sec. III-A) used to validate the proposed middleware (sec. III-B).

A. FreeRTOS MPSoC Platform

Fig. 3 illustrates a 4x4 NoC-based homogeneous MPSoC platform, with 2x2 cluster size. Each PE includes a Cortex-M4F, private memory, network interface, DMA and router. Each PE runs an extended FreeRTOS kernel independently, which uses MPI-based communication and cluster-based distributed management with dynamic task mapping [9].

In this platform, each PE has functions of global manager (GM), local manager (LM), or slave PE (SP). The LM is responsible for mapping application tasks onto SP PEs belonging to its cluster. The GM, in addition to the LM functions, assumes global functions such as application-to-cluster mapping and controlling the access to application repository. The SP executes the application tasks.

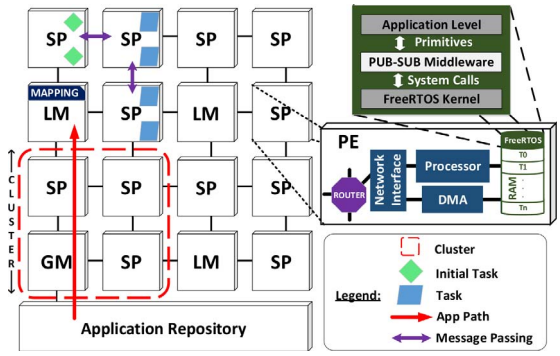

Fig. 3: 4x4 platform instance (adapted from [9])

B. Proposed Multiprocessor Publish-Subscribe Middleware

The proposed middleware is integrated on FreeRTOS between the kernel and the application level. While *system calls* provide the PUB-SUB API primitives to the user application(s), the network interruptions (NIs) are used to manage the API services at system level. Table I shows the list of primitives available on the API.

The management structure remains itself with the incorporation of the PUB-SUB middleware, with PEs assuming new management functions. Both LM and GM assume the role of brokers, but they can also be publishers and/or subscribers. The SPs can only be publishers and/or subscribers. Fig. 4 shows the sequence diagram representing the processes executed by each primitive in the kernel and the NoC.

When a node performs a *MQSoCAdvertise(topicID)* primitive (1A), the kernel adds a *topicID* and a *pageID* to the *Local Publishers* table (1B). Also, the kernel sends an *advertise control message* to the local broker in the NoC with the *topicID* and the *nodeID*, which is its respective NoC physical address (1C). The broker receives this message and stores underlying message data in its *Publishers* table (1D). Also, it forwards the message to other brokers for synchronization

TABLE I: PUB-SUB primitives available on the API.

Primitive	Used by	Description
MQSoCAdvertise(topicID)	Publisher	Advertise the broker that the respective node is the publisher of the topic identified by topicID.
MQSoCPublish(topicID, message)	Publisher	Send the message to the topic identified by topicID.
MQSoCSubscribe(topicID, callbackf)	Subscriber	Subscribe to the topic identified by topicID passing the pointer to the function that will process the message when it arrives.
MQSoCYield(timeout, cnt_rcv, suspend)	Subscriber	Generate a loop that verifies in a *timeout* frequency whether a message was received; the callback function defined in *MQSoCSubscribe* primitive is executed whether there is a message to the respective topic; the loop is finished when all *cnt_rcv* messages are received; the task is suspended when *suspend* is set and no message is received (it is resumed when a message arrives).

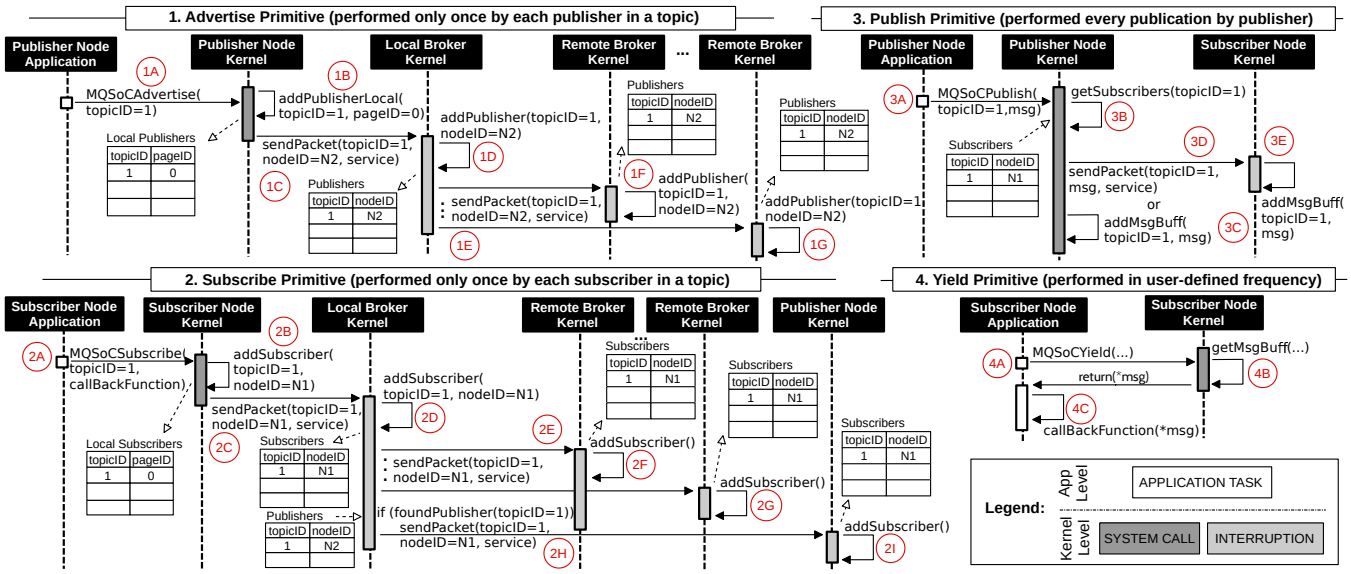

Fig. 4: Sequence diagram showing the publish-subscribe communication protocol execution, considering the adopted platform.

(1E). The forwarding is necessary to keep a copy of the location of each publisher into all brokers of the system. Thus, publishers and subscribers do not need to be in the same cluster. When the message is received by other brokers, they store the data in their *Publishers* table (1F, 1G).

When a node performs a *MQSoCSubscribe(topicID, callBackFunction)* primitive (2A), the kernel adds *topicID* and *pageID* to the *Local Subscribers* table (2B). This table is necessary to the case of subscriptions in multiple topics into the same node. Also, the kernel sends a *subscribe control message* to its broker with the *topicID* and its *nodeID* (2C). When received, the local broker stores the data in its *Subscribers* table (2D). Also, it forwards the message to other brokers for synchronization (2E), which store the data in their *Subscribers* table (2F, 2G). The subscription is carried out only in the local broker, that searches for a publisher in its *Publishers* table. If there is a publisher to that topic, a *subscription control message* is sent to the publisher node containing the *topicID* and the subscriber *nodeID* (2H). When this message arrives at the publisher node (2I), it adds the data to its *Subscribers* table. If no publisher is found, the request is stored, waiting for an *advertise* to that topic.

When a node performs a *MQSoCPublish(topicID, message)* primitive (3A), the kernel searches by subscriber nodes to the respective topic in its *Subscribers* table (3B). If there is a subscriber to that topic, then a data message can be sent to the subscriber node in two ways: by the NoC, if the publisher is not in the same PE (3D); or directly added to the subscriber buffer, when publisher and subscriber are in the same PE (3C). When the data message is sent by the NoC, the subscriber node adds the message to the subscriber buffer (3E).

When a node performs a *MQSoCYield(timeout, cnt_rct, suspend)* primitive (4A), the kernel verifies in the buffer whether there is a message to the node (4B). In this case, the message is processed by respective callback function informed in the *MQSoCSubscribe(topicID, callBackFunction)* primitive (4C). Otherwise, the task could be suspended until a message is received, or execute any other part of its software.

IV. EXPERIMENTAL SETUP AND RESULTS

The current platform was implemented and validated using the OVPSim[4], which has an instruction accurate simulation. All hardware architecture was described using OVPSim APIs.

The experiments are based on the DTW application (Fig. 2). This application has been chosen because it uses a communication pattern of 1:N and N:1 (N is the number of workers). This experiment uses eight workers. We analyze three scenarios: *MPI-all*, *MPI-dem*, and *PUB-SUB*. The first two scenarios use MPI primitives with all tasks mapped at begin of execution (*MPI-all*), or tasks mapped on demand (*MPI-dem*), where only the initial tasks are mapped at begin of execution and the other tasks are mapped as soon as there is a communication among them. The *PUB-SUB* scenario uses the proposed publish-subscribe primitives and middleware, with all the tasks mapped at begin of execution. All scenarios use a single-cluster 5x5 MPSoC, with each PE executing a single task to stimulate the NoC communication between the tasks and evaluate the middleware protocol.

Fig. 5 shows the results of DTW application execution time, using a model that captures the executed instructions for each PE, generating an execution time from total executed instructions [10]. PUB-SUB reduces the execution time from 2.6% to 29.9% as the number of patterns is increased, respectively, from 16 to 256. Compared to MPI, the PUB-SUB model requires an initial setup time to advertise the topics. Besides, the PUB-SUB application object code is lightly bigger, taking more time to finish the task mapping. Therefore, the MPI has advantage for small communication volumes. However, the MPI model presents the drawback of generating more System Calls and NIs caused by the messages, as detailed next.

Fig. 6 is a detailed view of the results obtained for 64 patterns, presented in Fig. 5. The X axis represents the order

[4]http://www.ovpworld.org/technology_ovpsim

of System Calls or NIs generated in the system, and the Y axis represents the instant of time (timestamp) in which each of them was executed. The figure also presents two lines representing the MPI and the PUB-SUB execution trace. Since both *MPI-all* and MPI-dem had the same behavior, only one is illustrated. The figure is divided into the three main phases of DTW application: *setup*, *data fork*, and *data join*.

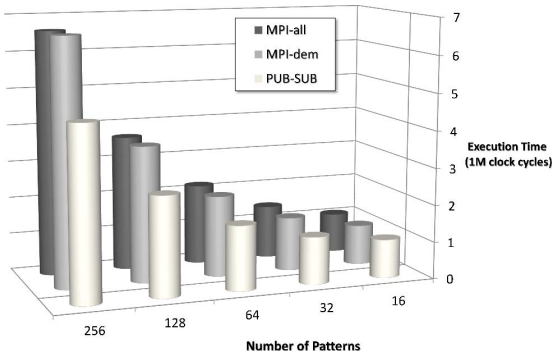

Fig. 5: DTW execution time using MPI and PUB-SUB.

The *setup phase* for MPI is the time between the first (1) and last (3) task allocation. The setup phase for PUB-SUB is longer because it allocates the tasks (1 to 2), and it also performs the topic advertisement, concluded at (4).

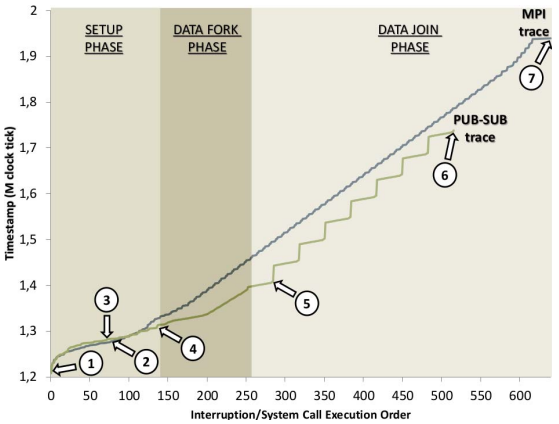

Fig. 6: MPI vs PUB-SUB time spent in System Calls and NIs.

The *data fork phase* represents the time the *bank* task sends the first message to a *worker*. The PUB-SUB (4) starts later than MPI (3) due to the advertisement time. This graph shows that, once PUB-SUB starts the data fork phase (4), it starts to present advantage because the *bank* task performs only one System Call to send the message to all *workers*. On the other hand, the *bank* task of the MPI version waits for an NI caused by a message request from each *worker*, then the message is sent to the *worker* that requested it.

The *data join phase* represents the time between the first and the last message received by the *recognizer* task from a *worker* task. In the MPI version, when the *recognizer* task executes an *MPI_receive(target from)* primitive, it sends a message request to a single *worker*, going to a suspended mode until the message is received. The message request causes an NI at the *worker*. If the data is ready, the data message is sent by *worker* to the *recognizer* task, causing another NI at the destination. However, if the data is not ready, the *recognizer* task remains in suspended mode. These steps are repeated for each worker. On the other hand, in the PUB-SUB version, all *workers* publish on the same topic, and the *recognizer* task has a callback function to treat these messages. The stair case behavior observed in (5) is repeated whenever the *recognizer* task goes to suspended mode. It is resumed when a new message arrives. The *recognizer* task does not need to request data, and it does not need to block waiting any *worker*. The application is finished in (6) to PUB-SUB and (7) to MPI.

Although the PUB-SUB advertisement step adds some initial latency to start the task communication, this pays off because both publishers and subscribers know that the broker connects each other when they are ready to start. This way, the message request-response and the blocking receive executed in MPI for each transaction are not required.

In terms of memory footprint, the proposed publish-subscribe middleware with FreeRTOS kernel has 23kB, which corresponds to a 7kB increase compared to the MPI-based FreeRTOS kernel.

V. Conclusion

This work presented a publish-subscribe middleware for MPSoCs. Results show that proposed middleware is a worthwhile alternative to programming NoC-based multiprocessor platforms, with low footprint overhead, and lower execution time when compared to an MPI-based FreeRTOS kernel implementation. Future works include to incorporate QoS and reliability-oriented features to the underlying middleware.

Acknowledgment

The authors would like to thank Imperas Software Ltd. and Open Virtual Platforms for their support and access to their models and simulator. Jean Carlo Hamerski is supported by CNPq (process 460205/2014-5) and IFRS.

References

[1] P. T. Eugster *et al.*, "The Many Faces of Publish/Subscribe," *ACM CSUR*, pp. 114–131, Jun. 2003.

[2] M. Ruaro, E. A. Carara, and F. G. Moraes, "Runtime Adaptive Circuit Switching and Flow Priority in NoC-Based MPSoCs," *IEEE Transactions on VLSI Systems*, pp. 1077–1088, Jun. 2015.

[3] D. Gohringer *et al.*, "RAMPSoCVM: Runtime Support and Hardware Virtualization for a Runtime Adaptive MPSoC," in *IEEE FPL*, Sep. 2011, pp. 181–184.

[4] E. A. Carara, N. L. V. Calazans, and F. G. Moraes, "Differentiated Communication Services for NoC-Based MPSoCs," *IEEE Transactions on Computers*, pp. 595–608, Mar. 2014.

[5] X. Chen *et al.*, "Reducing Virtual-to-Physical address translation overhead in Distributed Shared Memory based multi-core Network-on-Chips according to data property," *Computers & Electrical Engineering*, pp. 596–612, Feb. 2013.

[6] R. Garibotti *et al.*, "Efficient Embedded Software Migration towards Clusterized Distributed-Memory Architectures," *IEEE Transactions on Computers*, pp. 2645–2651, Aug. 2016.

[7] S. Dobson *et al.*, "A Survey of Autonomic Communications," *ACM TAAS*, pp. 223–259, Dec. 2006.

[8] P. Bellavista, A. Corradi, and A. Reale, "Quality of Service in Wide Scale Publish-Subscribe Systems," *IEEE Communications Surveys & Tutorials*, pp. 1591–1616, Apr. 2014.

[9] G. Abich *et al.*, "Extending FreeRTOS to Support Dynamic and Distributed Mapping in Multiprocessor Systems," in *ICECS*, 2016.

[10] F. Rosa *et al.*, "Fast energy evaluation of embedded applications for many-core systems," in *PATMOS*, Sep. 2014, pp. 1–6.