Communication support at the OS level to enhance design space exploration in multiprocessed embedded systems

Alexandra Aguiar alexandra.aguiar@pucrs.br
Sergio Johann Filho sergio.filho@acad.pucrs.br
Felipe Magalhaes felipe.magalhaes@pucrs.br
Fabiano Hessel fabiano.hessel@pucrs.br
Faculty of Informatics - PUCRS - Av Ipiranga 6681, Porto Alegre, Brazil

ABSTRACT

Currently, Embedded Systems (ESs) based on Multiprocessed System-on-Chip (MPSoCs) count on resources previously available only on general purpose machines, leading to an increased design complexity, especially when dealing with communication-dependent applications. In this context, we present a communication protocol developed in a Real Time OS (RTOS) to provide a transparent communication interface for both bus- and NoC-based MPSoCs' applications.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

Keywords

Embedded systems, RTOS, MPSoC, Bus, NoC

1. INTRODUCTION

ESs have presented, increasingly, a rising number of features leading to a significant growth in the design complexity of applications. Also, systems have had their implementation based in multiple processing elements integrated on the same die, running at a lower clock frequency, due to common energy consumption constraints [2]. The communication infrastructure decision can vary according to certain characteristics of the entire platform as it is desirable that the application remains the same, avoiding code rewriting and design rework and improving the overall software quality. Still, since communication-based decisions impact directly on the final system's behaviour, it is important to have both OS and framework support to explore different decisions in order to enable a wide design space exploration.

This paper presents a communication protocol implemented in an RTOS that provides a transparent communication layer in bus- and NoC-based MPSoCs. We implemented this protocol using an RTOS in which applications can be added through a design framework. In this case, the application's software can be used in two different communication infrastructures in a straightforward fashion. We evaluate our platform measuring the overhead of the protocol, its performance and throughput and show how it can be used with both communication infrastructures (bus- and NoC-based systems).

SAC'13 March 18-22, 2013, Coimbra, Portugal.

2. DESIGN SPACE EXPLORATION USING HELLFIRE FRAMEWORK

The Hellfire Framework (HellfireFW), firstly introduced in [1], allows a complete deployment and test of multiprocessed embedded applications, defining the HW/SW architecture to be employed by the designer. HellfireFW assumes the HellfireOS (HFOS) [1] as the system's kernel.

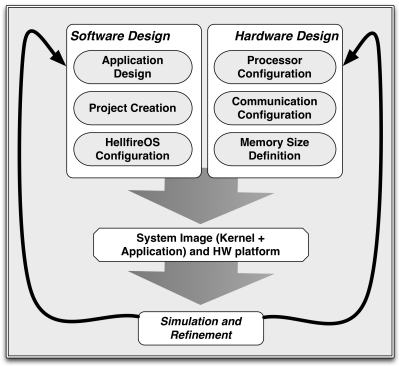

Figure 1: Hellfire Framework Design Flow

3. COMMUNICATION MODEL FOR BUS- AND NOC- BASED MPSOCS

Figure 2 presents the steps needed to perform a message exchange between two processors in bus- and NoC-based environments. For both cases, the sending primitive encapsulates the message into packets, fulfilling the task's sending queue (1). The packet is then removed from this queue and it is then copied to the hardware outgoing queue (2). The network interface in then notified as the packet is sent through the network (3). After the packet reaches its destination processor by getting into the incoming queue, an

interrupt is sent to the operating system, which removes the packet from the queue and decodes it, before forwarding it to the target task's queue (4). Finally, the message can be used at the application level (5).

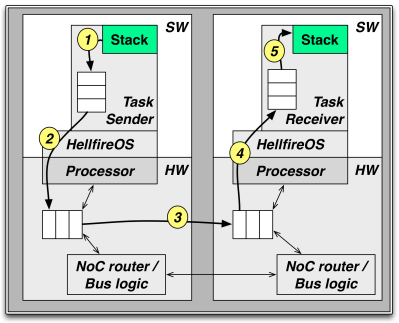

Figure 2: Communication among application tasks, hardware and software queues

Our communication primitives were implemented in two different levels of abstraction. High level primitives are exposed in the OS's API and are responsible for encapsulating and dividing messages into data packets, dealing with details such as padding and sequencing. Internally, system's drivers are responsible for transferring packets between source and target. They work with fixed-sized packets and signal the communication network interface while sending data to hardware queues. Likewise, these drivers are signalled while receiving data, as they take the data off the hardware receiving queues and copy them to the circular packet queues of each task.

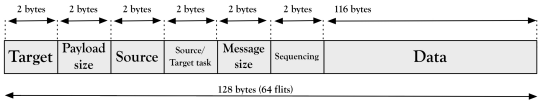

Figure 3: Packet's structure

4. EVALUATION

In order to evaluate the message-passing performance at the application level a simple yet highly communicating application was implemented, varying the message size from 50 bytes to 1000 bytes. Half the processors contain senders, and the other half, receivers. All tasks were configured to execute each 104ms (period) during 82ms (capacity) through the period yielding an 80% processor utilization. Along with the application task, other operating system tasks execute at the same time on the same core. The same application was used in several configurations, where it was only scaled for the desired number of cores (the same code was recompiled without modifications).

Figure 4 presents the results observed for several MPSoC configurations. As it can be seen, bus-based communications for this application start to decrease its performance due to network congestions when a higher number of processors communicates. The last simulated case was a 256 core, 16x16 mesh MPSoC and the NoC could keep the high throughput, avoiding congestions. A 256 core bus-based MPSoC suffers from high penalty for this application, highlighting the superior performance of the NoC approach for

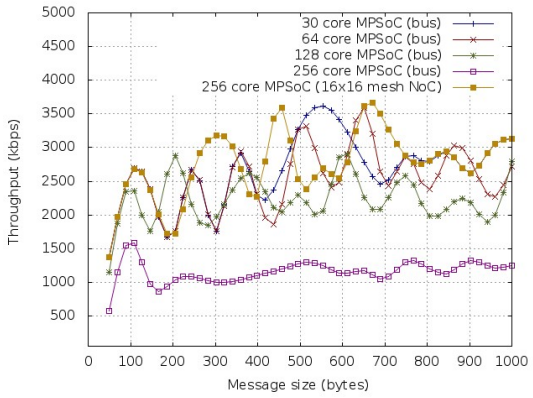

Figure 4: Throughput for bus and NoC-based MPSoCs

communication-intensive applications. In cases where performance is closely related to communication efficiency (such as in streaming applications), the proposed model offer a scalable and manageable way of describing distributed applications.

5. DISCUSSION

The proposed design flow and the implementation strategies of HellfireOS can be combined, enabling a very powerful design space exploration. Designers do not need to change neither the application nor the OS's code to test their systems under various different scenarios.

The scenarios to be explored concern: (i) different application strategies in multiprocessed systems (using the same communication API, only changing the application's logic itself); (ii) different OS configuration (heap size, tasks' memory use, scheduling options, etc); (iii) different processor options (architecture, type, frequency); (iv) different memory sizes, and; (v) different communication strategies (bus or NoC) and, for each, its own specific parameters.

Each execution on such a rich simulation environment allows the designer to evaluate the best possible scenario for the multiprocessed system. Besides, if there is an equivalent hardware platform described, for example, in VHDL, it is possible to use the application and OS binaries in a straightforward fashion, preventing the occurrence of the model continuity problem.

Acknowledgment

The authors acknowledge the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology Embedded Critical Systems Brazil), processes 573963/2008-8 and 08/57870-9.

6. REFERENCES

[1] A. Aguiar, S. F. Johann, F. G. Magalhaes, T. D. Casagrande, and F. Hessel. Hellfire: A design framework for critical embedded systems' applications. In *ISQED '10*, pages 730–737, 2010.

[2] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.