Run-time mapping and communication strategies for Homogeneous NoC-Based MPSoCs

G. Sassatelli, N. Saint-Jean, P. Benoit, L. Torres, M. Robert
LIRMM - 161 rue Ada, Montpellier, France 
{nom}@lirmm.fr

Cristiane Woszezenki, Ismael Augusto Grehs, Fernando Moraes
PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil 
{cristianew, grehs, moraes}@inf.pucrs.br

Abstract

Multi-Processor Systems-on-Chip are becoming increasingly popular in embedded systems for the high degree of performance and flexibility they permit. While most MPSoCs are today highly heterogeneous for better fitting the target applications, homogeneous systems may become in a near future a viable alternative bringing other benefits such as run-time load balancing, high performance and low power consumption. The work presented in this paper relies on a homogeneous NoC-based MPSoC framework we developed which allows us to conduct cycle-accurate evaluations of 2 different techniques: proactive and reactive communications.

1 Introduction

Multiprocessor Systems-on-Chip are custom architectures that balance the constraints of VLSI technology with application requirements. The homogeneous NoC-based MPSoC presented in this paper is designed for exploring strategies of communication to improve adaptability.

Efficient Inter-tasks communication in MPSoCs is important for obvious performance reasons, but also to avoid phenomena such as deadlocks. This paper evaluates two original dead-lock free communication mechanisms.

2 The MPSoC framework

2.1 Hardware

Figure 1 presents the developed architecture. MPSoC architectures may be represented as a set of processing nodes that communicates via a network. HERMES Routers [1] compose the network and RISC processors the processing nodes (Plasma [2]). As the total number of tasks composing the target applications may exceed the MPSoC memory resources, one processor is dedicated to the management of the system resources (MP - Manager Processor). It has access to the task memory, which acts as a repository for task codes. When the MPSoC starts its execution, tasks are allocated into some processors of the system.

2.2 Programming model & Software

The adopted programming model employs a Sequential Procedural Programming basis, a multi-tasking support and communication primitives for inter-task communications. In our model, the tasks are described in C language and parallelism therefore exists at task-level.

Since tasks execution may be time-sliced, which means they can run in arbitrary bursts as directed by the operating system, the property of confluence (same result yielded regardless task execution order) must be guaranteed. The underlying programming style for ensuring the synchronization of the computation in our approach is Kahn Process Networks (KPN) [1].

In order to time-multiplex tasks on a single processor, and to handle communications between local and remote tasks, an Operating System offering the necessary functionalities is necessary. We have developed a lightweight microkernel (9 KB) which was designed for our specific needs (limited memory footprint, dynamic executable loading).

The communication primitives essentially abstract communications so that tasks can communicate with each other without knowing their position on the system.

3 Communication Strategies

3.1 Proactive communication

The principle of this communication technique is depicted in Figure 2. A producer task sends data using the
Figure 5 presents similarly to previously the different steps were written.

In order to ensure message delivery, the processor 1 sends the message to processor 2 through the message from task t2 (details the different steps of the process and the obtained task 5 can use (fig 2.b). As soon as enough data are available in the socket, the handshake process is initiated by the microkernel. Firstly, a request message is sent to the processor hosting the consumer task. The microkernel then waits for an acknowledge message (fig 2.a). Then, the remote processor informs the microkernel through an acknowledge message which specifies the number of available positions in the queue. At last, the initiator transfers data corresponding to the minimum between message size and available positions in remote queue.

Figure 3 – Remote read process for dedicated queues.

3.2 Reactive communication

Figure 4 presents the schematic view of the reactive communication scheme. When a process executes a read_socket function, a system function is called. The microkernel sends a request message through the NoC, and the task enters in wait state. When the message arrives from the network, the microkernel stops the task being executed, and schedules the task waiting the message. In Figure 4.a it is supposed that task t2 has written a message in global_pipe, addressed to task t5 (write_socket(dmsg,5)), and the task t5 is requesting the message from task t2 (read_socket(dmsg,2)). In Figure 4.b, processor 1 sends the message to processor 2 through the NoC. In order message delivery is ensured, since the write_socket() function adds to each message the relative order they were written.

Figure 5 presents similarly to previously the different steps involved in the communication.

4 Results and Conclusion

Table 1 presents implementation results for a MJPEG decoder application made of three communicating tasks. Performance figures are expressed in throughput (kB/s) for different application partitioning (number of tasks). Bracketed task groups denote time sliced execution (hosted on the same processor).

Table 1 - Performance results for the MJPEG decoder.

<table>
<thead>
<tr>
<th># of CPUs</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>(IVLC, IQUANT, IDCT)</td>
<td>(IVLC, (IQUANT), (IDCT)</td>
<td>(IVLC, (IQUANT), (IDCT)</td>
</tr>
<tr>
<td>Proactive Tp.</td>
<td>63</td>
<td>133</td>
<td>132</td>
</tr>
<tr>
<td>Reactive Tp.</td>
<td>85.7</td>
<td>116</td>
<td>116</td>
</tr>
</tbody>
</table>

Results suggest better performance for the proactive technique; however, this results in much higher memory consumption because of the separate queues used. It also implies a processing overhead for decoding write_socket() function calls and buffering data in the dedicated queues (Write proactive communication time: 3014 ck cycles, Write reactive communication time: 813 ck cycles).

5 References