Architectural Issues in Homogeneous NoC-Based MPSoC

Gilles Sassatelli[1], Nicolas Saint-Jean[1], Cristiane Woszezenki[2], Ismael Grehs[2], Fernando Moraes[2]
[1]*LIRMM - 161 rue Ada, Montpellier, France, {sassatelli, Nicolas.Saintjean}@ lirmm.fr*
[2]*PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil, {cristianew, grehs, moraes}@ inf.pucrs.br*

Abstract

Multi-Processors Systems-on-Chip (MPSoCs) 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. The work presented in this paper relies on a homogenous NoC-based MPSoC framework we developed which allows us to conduct evaluations of 3 different techniques: proactive/reactive communication, preemptive task mapping and task replication. The impact on the system performance is discussed for several mappings of a MJPEG decoder application.

1 Introduction

The homogeneous NoC-based MPSoC presented in this paper is designed for exploring strategies of adaptability. It is made of a homogeneous array of elementary microprocessors connected through a packet-switching NoC. Each microprocessor of the network is running a lightweight microkernel that provides the necessary support for adaptability mechanisms.

Static mapping algorithms are being used for bus-based [1] and NoC-based MPSoCs ([2] [3]). Even if these procedures are inadequate for MPSoC with dynamic workload, such methods may be used to compute the initial task mapping, or be optimized to be employed at run-time. Task migration algorithms, widely used in distributed systems and parallel computing domain, are increasingly being used in MPSoC to improve the overall system performance. Works [4], [5] and [6] discuss the cost to interrupt a given task, save its context, transmit all data to a new IP, and resume processing in the new IP.

Besides task scheduling, mapping and migration algorithms have already being exploited in parallel and distributed applications, their application in MPSoCs is not straightforward. The huge available bandwidth in NoCs and the great number of processors require the development of new techniques to fully exploit performance of MPSoCs. This paper contributes in this arena, evaluating 3 different techniques: proactive/reactive communication, pre-emptive task mapping and task replication.

2 Programming Model

The adopted programming model employs a Sequential Procedural Programming basis, a multi-tasking support and communication primitives for inter-task communications. The tasks in our model are described in C language, and parallelism in our model 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) [7]. KPN is a distributed model of computation where processes are connected to each other by unbounded first-in first-out communication channels to form a network of processes. KPNs rely on the fundamental principle that communication must be: *blocking* for channel read operations; and *non-blocking* for channel write operations.

A set of communication primitives has been derived from this formalism for ensuring confluence of application execution regardless task execution order; those primitives belong to the hardware-dependent software part of our framework and are detailed in section 4.

3 The MPSoC framework

Figure 1 presents the developed architecture. MPSoC architectures may be represented, without loss of generality, as a set of processing nodes that communicate via a communication network. Routers (*R*) compose the network (HERMES [8]) and RISC processors the processing nodes (Plasma [9]). 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.

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. After checking the literature and existing embedded OS (uClinux, eCos, etc.), it appeared that our specific requirements (limited memory footprint, dynamic memory management, dynamic executable loading) were too strong to reuse any of these solutions. Therefore, we have developed a lightweight microkernel (9 KB) which was de-

18th IEEE/IFIP International Workshop on Rapid System Prototyping(RSP'07)
0-7695-2834-1/07 $20.00 © 2007 IEEE

signed for our specific needs. Despite being small, this OS does preemptive switching between tasks and also provides them with the communication support for tasks interactions (communication primitives).

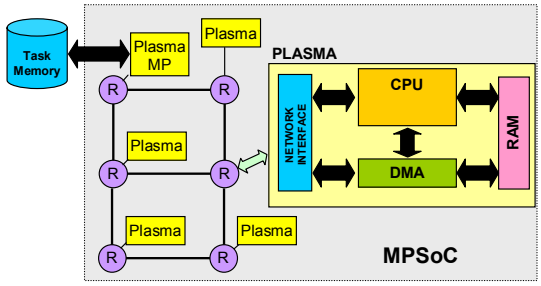


Figure 1 - Homogenous NoC-based MPSoC.

The communication primitives essentially abstract communications so that tasks can communicate with each other without knowing their position on the system (either on the same processor or a remote one). Communication management between tasks is ensured by two dedicated read and write functions under microkernel control. These functions, *Read_Socket()* and *Write_Socket()*, are respectively blocking and non-blocking in accordance to the previously detailed programming model (KPN). These functions allow transparent data communications between tasks regardless their position on the network since data routing is done following a dynamically updated task table.

4 Architectural explorations

Three main families of adaptability strategies have been evaluated in this paper: communication strategies, preemptive task mapping, and task replication.

4.1 Communication Strategies

Proactive communication / dedicated queues

This technique employs a dedicated queue per edge of the communication graph. The microkernel may nevertheless allocate existing queues to different tasks over time. Figure 2 depicts the principle of such a communication technique.

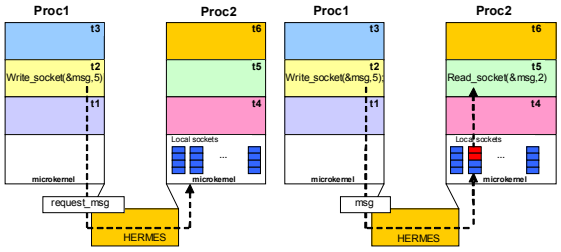


Figure 2 - Proactive communication principle.

A producer task sends data using the *write_socket*() function that immediately establishes a communication with the processor hosting the consumer task. Data are then transmitted over the network and buffered locally. When the two communicating tasks are hosted onto the same processor, the same principle is used locally: the microkernel just copies data produced by the task to the dedicated queue of the consumer task.

Reactive communication / centralized queues

Figure 3 presents the schematic view of the reactive communication scheme. Each processor may execute a given number of tasks (`t1` to `t6` in the Figure). The microkernel has a dedicated buffer, named *global_pipe*, which receives messages form local tasks.

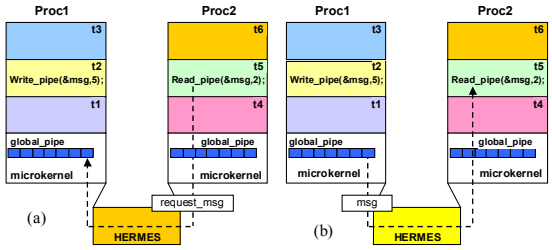


Figure 3 – Reactive communication principle.

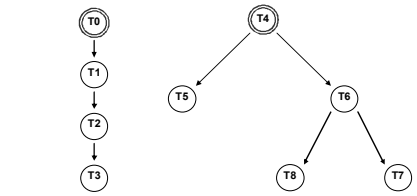
The remote communication starts with a process executing a write operation. When a given process executes a *write_pipe* function, the message is transferred from the task memory to the microkernel memory (*global_pipe*).

The read operation as follows. In Figure 3 it is supposed that task `t2` has written a message in the *global_pipe*, addressed to task `t5` (*write_pipe(&msg,5)*). Asynchronously, task `t5` request the message from task `t2` (*read_pipe(&msg,2)*). The microkernel sends a request message through the NoC, and task `t5` enters in wait state (Figure 3(a)). When the message arrives from the network, the microkernel stops the task being executed, and schedules the task waiting the message – `t5` (Figure 3(b)). In order message deliver is ensured, since the *write_pipe* adds to each message the relative order they were written.

4.2 Preemptive task mapping

As applications tend to grow both in size and in features, an increase in the number of tasks used for described them is to be expected. As depicted in Figure 1, tasks are stored in the *task memory*. When the MPSoC starts executing, the MP (*manager processor*) sends to the slave processors the initial applications tasks. Figure 4(a) illustrates two hypothetical applications, with 4 and 5 tasks respectively. The code in Figure 4(b) contains the function `InitializeTasks`, which is responsible to assign tasks T0 and T4 to processors 1 and 5 respectively (functions `InsertTaskLoc` and `OccupedPage`).

(a)



(b)

```
void InitializeTasks()
{       InsertTaskLoc(T0,SLAVE1);
        OccupedPage(SLAVE1);
        InsertTaskLoc(T4,SLAVE5);
        OccupedPage(SLAVE5);
}
```

Figure 4 – (a) two applications, with 4 and 5 tasks respectively; (b) initial task allocation.

4.3 Task replication

Figure 5(a) depicts a synthetic task graph. A profiling may show that *task2* is (*i*) the most demanding and (*ii*) exhibit data-dependent computational load. Replicating it helps in increasing the performance; which would lead to the scenarios depicted on Figure 5(b) and Figure 5(c). In such cases, of course all three instances of task 2 would be hosted on a dedicated processor. The experiments conducted do not yet implement the automated replication strategy but show the performance figures for different task graphs. Strategies enabling run-time replication may either be simple (*fork*() and join()) or more difficult, therefore requiring programmer attention.

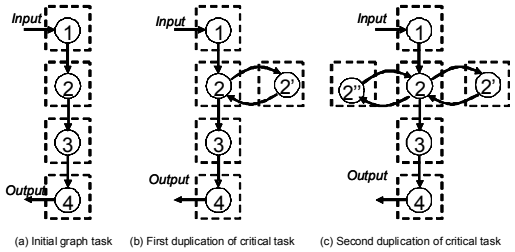


Figure 5 – Initial application task graph(a) and replication of task 2 (b and c).

5 Results

The MPSoC architecture has been developed in synthesizable RTL VHDL; therefore all results presented have been produced either thanks to RTL simulations or directly observed on the FPGA-based prototype. A Motion-JPEG (MJPEG) decoder application has been chosen for the experiments conducted on our platform. This video decompression algorithm, which exhibit few inter-task dependencies, is relatively well suited for evaluating the some of the principles presented in section 4.

5.1 Communication strategies evaluation

Proactive communication evaluation. Figure 6 shows the performance expressed in clock cycles we obtained for transferring a 100 bytes message between two tasks located onto the same processor. This technique is *proactive* since it transfers data without waiting a read request.

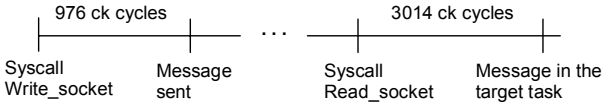


Figure 6 – Performance for a 100-byte message when tasks belonging to the same processor exchange data.

When two communicating tasks are hosted on different processors a remote write process is triggered, as depicted in Figure 7.

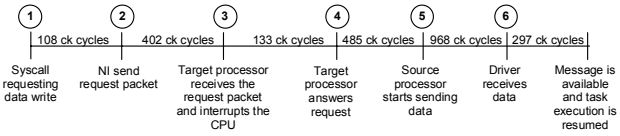


Figure 7 – Remote read process for dedicated queues.

Reactive communication evaluation. Figure 8 presents the performance, in clock cycles, to transfer a 100-byte message between two tasks, mapped on the same processor. A source task invokes the *write_pipe* function, executing a system call (*syscall*) which writes the message in the *global_pipe*. When the target task is scheduled, the system function *read_pipe* transfers the message to the task.

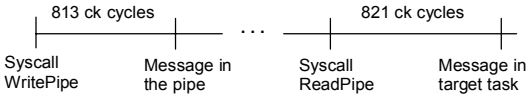


Figure 8 – Performance for a 100-byte message when tasks belonging to the same processor exchange data.

When tasks located in different processors communicate, as depicted in Figure 3, the time spent to write the message in the *global_pipe* is the same of the Figure 8. The read process requires a request transaction, before effectively receiving data. For that reason, this technique is *reactive*. Figure 9 details the process.

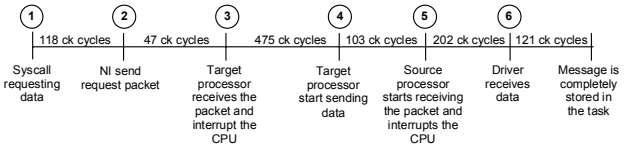


Figure 9 – Remote read process for unified pipe.

Table 1 presents implementation results for the MJPEG decoder application. Performance figures are expressed in throughput (kB/s) for different application partitioning (number of tasks), for both the proactive and reactive communication techniques. Throughput of both communication methods is similar. Inter-task communication in the same processor (column *1 CPU*) favors the reactive method, since most write operations bypass the pipe. Meanwhile, the cost to receive a read request, and after transmit data penalizes the reactive

method, as depicted in columns *2* and *3 CPUs*.

Table 1 - Throughput Performance results.

# of CPUs	1	2	3
Proactive system Tp (kB/s)	63	133	132
Reactive system Tp (kB/s)	85.7	116	116

Results may change depending on the network traffic (small in our experiments). The proactive communication technique may present superior throughput when the network is saturated, since data transfer starts in the write operation, not in the read operation. Nonetheless, the proactive communication technique makes use of several dedicated queues, which results in much higher memory consumption. It also implies a processing overhead for decoding *write_socket* function calls and buffering data in the dedicated queues (see Figure 6 and Figure 7).

5.2 Preemptive task mapping

In the conducted experiment, the time to allocate a new task is in average 2 to 3 times superior the time slice. The result is a small performance penalty in terms of execution time (less than 4%), without incurring throughput reduction compared to a static allocation (all tasks initialized by the MP when system starts executing). These results are application dependent and the induced task mapping latency is proportional to the task code size. This allocation method enables the MPSoC to execute a number of tasks superior to the available memory resources in the system, and it can be very useful for automatically load/unload from memory tasks that are rarely used to the benefit of more intensive ones.

5.3 Task replication

Table 2 shows the performances achieved when replicating some of the tasks. As clearly suggested by the throughput values, the IVLC task is the critical one in our case since duplicating and triplicating it (columns 5 and 6) significantly increase the performance. Allocating a fourth processor to this task does not further improve the performance meaning that another task then became the critical step in the processing pipeline.

Table 2 - Performance results for task replication.

Replicated task/degree	none	IQUANT/2	IDCT/2	IVLC/2	IVLC/3
# CPUs	3	4	4	4	5
Tp (kB/s)	132	132	131	352	418

This technique show promising results for achieving adaptability in MPSoC platforms. Many potential applications for which computational load or quality of service (QoS) requirements vary over time may take a direct advantage of task replication.

5.4 FPGA Prototype

RTL Simulations are too slow for significant applications such as MJPEG performed on streams of data. We decided then to use a Xilinx Development Kit to synthesize and validate our design. Six processors could be fitted on a XC2VP30 FPGA from Xilinx. A processor occupied 2151 slices on the FPGA which is rather small. Debugging on the prototype takes place thanks to a UART interface between one Plasma processor and a PC. Some additional services were added to the Plasma processor microkernel for feeding back debugging information directly to the PC.

6 Conclusion

Homogeneous MPSoCs are likely to emerge in the coming years because of the innovative adaptability features they enable. We have in this paper presented a NoC-based homogeneous MPSoC platform made of an array of tiny microprocessors running a lightweight multitasking microkernel. Due to the relatively limited processor performance task-level parallelism must be exploited for reaching high performance. This results in an additional constraint during application specification but also enable performing task-migration based techniques without unaffordable overhead.

Among the techniques, we estimate that preemptive task mapping and task replication are innovative in MPSoC design and promising features which may soon bring a high-level of adaptability for embedded systems. Together with some increasingly popular features such as DFS (Dynamic Frequency Scaling) and DVS (Dynamic Voltage Scaling), these mechanisms may contribute in lowering the performance and power consumption overhead, which are the main stated flaws of homogeneous MPSoCs.

7 References

[1] Ruggiero, M.; et al. *Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip*. In: DATE, 2006, 6p.

[2] Hu, J.; Marculescu, R. *Energy- and Performance-Aware Mapping for Regular NoC Architectures*. IEEE Transaction on CAD, v.24(4), 2005, pp. 551-562.

[3] Marcon, C.; et al. *Exploring NoC mapping strategies: an energy and timing aware technique*. In: DATE, 2005, pp. 502-507.

[4] Nollet, V.; et al. *Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles*. In:DATE, 2005, pp. 234-239.

[5] Bertozzi, S.; et al. *Supporting task migration in multi-processor systems-on-chip: a feasibility study*. In: DATE, 2006, pp. 1-6.

[6] Kalte, H.; Lee, G.; Porrmann, M. *Context Saving and Restoring for Multitasking in Reconfigurable Systems*. In: FPL, 2005, pp. 223-228.

[7] G. Kahn. *The semantics of a simple language for parallel programming*. In: IPIP Congress, 1974, pp 471-475.

[8] Moraes, F. et al. *Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip*. Integration the VLSI Journal, 38(1), October 2004. pp. 69-93.

[9] Open Cores, http://www.open.org.