2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines

Remote Execution in Distributed Memory MPSoC

Rémi Busseuil, Luciano Ost, Rafael Garibotti, Gilles Sassatelli, Michel Robert
LIRMM – 161 rue Ada, Cedex 05 – Montpellier – 34095 – France
{remi.busseuil, ost, garibotti, sassatelli, michel.robert}@lirmm.fr

***Abstract*- Message-passing is an increasingly popular design style for MPSoCs that usually results in systems that perform better compared to external shared-memory designs performance and power-wise, this because of much decreased data transfers with external memory. This scheme relies on explicit communications between processing tasks that participate in the application. Contrarily to shared-memory multiprocessors, tasks usually get assigned to processors at design-time. In order to cope with transient performance losses originating from various phenomena such as increased processing workload or peak traffic in the communication subsystem, various adaptation mechanisms based on task migration have been proposed in the literature. As Message-passing systems usually use PE-private memory architecture, these mechanisms imply migrating application code from processor to processor, which incurs penalty in performance and power consumption. This paper proposes a local shared-memory strategy in which processors execute code hosted in a remote processor.**

Keywords-component; remote execution, memory organization, network-on-chip, MPSoCs.

I. INTRODUCTION

NoC-based MPSoC platforms have to provide quality of service (QoS - e.g. guarantee of throughput) for broad application domains (e.g. real-time and avionics systems), which further require high performance allied to low power consumption [1]. Dealing with the workload variability of such applications using static approaches becomes inadequate for two main reasons. First, considering that not all applications may execute concurrently, allocating resources for all application tasks, at design time, can lead to MPSoC underutilization, and therefore system oversizing. Second, unexpected behaviors (e.g. user requests) may have a huge impact on the overall system performance, if not handled managed properly [2][3].

In this context, adaptive MPSoCs feature run-time techniques that allow balancing workload across processing elements (PEs) therefore enabling to deal with unpredictable behaviors and maintaining performance at run-time [4]. In order to perform load balancing for achieving better performance, efficient mapping heuristics and task migration mechanisms must be employed. Task mapping defines the initial placement for a given task while task migration re-maps this task if its performance is degraded due to the system behavior (e.g. higher PE load, NoC congestion). Task migration employs different activities (e.g. context saving and restoring [5]) that are not handled by the task mapping mechanisms. In the literature, these techniques have been proposed aiming to satisfy different system requirements.

For instance, Streichert et al. [4] employ task migration to maintain the correct execution of an application by migrating executing tasks from a PE that presents a fault (failure). In [5], authors present a migration case study for MPSoCs that relies on the μClinux operating system and a check pointing mechanism. This work was extended in [6], where an OS and middleware infrastructure for task migration was provided on the top of an MPSoC platform (NoCs are not considered). Shen et al [7] discuss the software and hardware architecture features that are necessary to support task migration in heterogeneous MPSoCs. In [8] authors propose a mechanism that exploits run-time temperature and workload information to define suitable run-time thermal migration patterns. In turn, Barcelos et al. [9] explore the use of task migration in a NoC-based system that comprises both shared and distributed memory organizations, aiming to decrease the energy consumption when transferring the task code. Goodarzi et al. [10] propose the use of virtual point-to- point (VIP) in order to reduce overhead caused by migrating tasks.

In addition to these works, this paper proposes a remote execution technique that draws inspiration from NUMA (Non-Uniform Memory Access) architectures, in which PEs are entitled to access remote PE memories, therefore lifting the need of physically migrating data. The *contributions* of this paper may be summarized as follows: (*i*) proposition of remote task execution strategy in a RTL NoC-based MPSoC, (*ii*) implementation of hybrid memory architecture that can be employed to optimize the performance of load balancing mechanisms (task migration and remote execution), and (*iii*) analysis in terms of advantages and drawbacks of employing shared and distributed memory organization when applying load balancing mechanisms, aiming to improve the load balancing efficiency. To the best of our knowledge, the present work is the first RTL NoC-based MPSoC implementation that supports both task migration and the proposed remote execution mechanism into the same system. Thus, according to a particular application or run-time scenario, either task migration or remote execution may be chosen. This paper does not address the actual dynamic mapping algorithms that drive task migration but rather focus on the underlying mechanisms, which were validated onto the OpenScale platform and prototyped in different FPGAs.

II. OPENSCALE ARCHITECTURE AND REMOTE EXECUTION

A. Platform Description

The OpenScale[1] architecture is a homogeneous message-passing NoC-based MPSoC with distributed memory. Each

[1] Available for download at: http://www.lirmm.fr/ADAC

978-0-7695-4699-5/12 $26.00 © 2012 IEEE
DOI 10.1109/FCCM.2012.30

node comprises a router and a PE. The PE is composed of a SecretBlaze (SBlaze) CPU with instruction and data caches (i.e. Microblaze-like architecture) [11] a timer, an interrupt controller, a RAM, optionally an UART and a network interface (NI).

The platform RTOS is a pre-emptive priority-based micro-kernel. Each SecretBlaze runs this RTOS independently, and communications and synchronizations are made through a message-passing API similar to MPI. Applications are represented through Khan Process Networks, a standard representation in such a message-passing oriented system [11]. Main features of the RTOS are: (*i*) run-time dynamic applications loading, (*ii*) preemptive round-robin scheduler based on thread credits, (*iii*) system-monitoring mechanisms (e.g. SecretBlaze utilization) and (*iv*) API with different drivers support (e.g. UART).

B. Remote Execution

Different from the task migration, in the remote execution mechanism the task is remotely executed in the new node, using a remote memory access protocol implemented in the NoC. The remote execution protocol is illustrated in Figure 1.

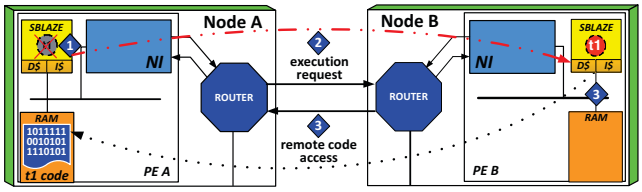


Figure 1 - Remote task execution protocol.

First SBlaze in *PE A* interrupts *t1* execution (step 1) and then sends an execution request with the current task state (step 2). Once the chosen *PE B* granted its request, *t1* is executed in SBlaze (*PE B*) by fetching its code from the remote SBlaze in *PE A* (step 3). To improve the execution efficiency, the instructions fetched remotely are cached in the L1 instruction cache of SBlaze (*PE B*).

It is important to note that the programming philosophy of the platform is message passing and not shared memory. Hence, only few data (static data) are fetched from remote memory during a remote execution, the main data flow coming from messages. In this configuration, we chose not to cache remote data during a remote execution. This strategy was taken considering that we have on-chip memory, with memory access latency few orders of magnitude less than standard shared memory architecture using off-chip memory. Furthermore, using uncached data avoids the use of complex cache coherency mechanisms between data cache and remote memories. Instructions, however, are entirely cached.

In order to support this protocol, a hardware module called remote memory access (RMA) was implemented. The RMA is composed of an *RMA-Send* and an *RMA-Reply*, which enable distant memory access through the NoC. Two asynchronous FIFOs are employed to guarantee the communication between both modules and the NoC, as shown Figure 2. The purpose of the *RMA-Send* is to answer the memory requests from its local CPU (step 1 and step 3). In case of *write request*, the *RMA-Send* component sends the data that must be stored in original task PE (e.g. in Figure 1 *PE B* transfers the resulting data of execution of *t1* to the *PE A*). In case of a *read request*, the *RMA-Send* component requests the desired data to the remote PE (e.g. in Figure 1 *PE B* requesting instruction code to the *PE A*). In turn, the *RMA-Reply* module answers the request coming from the *RMA-Send* (i.e. NoC side, step 2). In case the incoming request is granted the data is stored in the memory. In a read request, the data is read from the memory, packetized and sent to the PE that requested it.

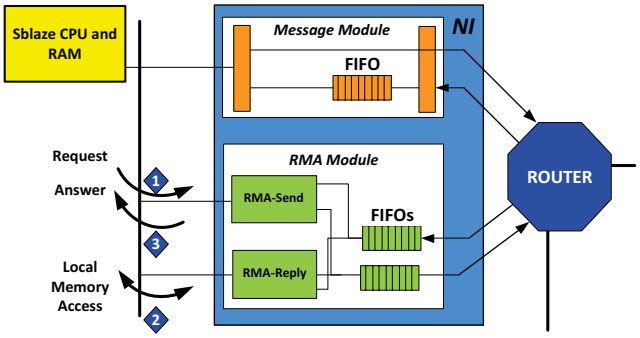


Figure 2 – Remote memory access (RMA) module and router connection.

The overall memory access performance depends on two main factors. First, as our architecture provides only a *L1* cache, the cache-miss rate is particularly important, as it determines the number of remote memory accesses, hence the traffic. Therefore remote memory access performance depends on: (*i*) cache size, (*ii*) cache policy, and (*iii*) number of remote memory accesses resulting from application. As the cache policy is fixed in the present platform, only the cache size and the remote memory access rate were considered as parameters for the performance measures (discussed in Section III).

In the target architecture, all PEs have local private memory. Therefore, every node may have the same address mapping. However, the RMA module brings shared memory features, thus it is necessary to devise a new memory mapping accordingly. Figure 3 shows this memory mapping. To match the adopted platform protocol, the four highest bits are kept for the selection of a local component that is connected to the CPU. In case of RMA module address, the eight following bits (Figure 3) are used to select the XY node coordinates. The last twenty bits are used for memory address. The adopted allows defining systems made of 16x16 PEs, with 1 MB of RAM each.

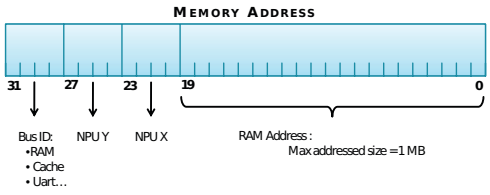


Figure 3 – Adopted memory mapping.

A second key factor of the overall memory performance is the remote memory access latency, which greatly depends upon NoC features such as channel width. The platform

configuration used to evaluate the remote memory access latency is described as follows: 2x2-mesh topology, XY routing algorithm, buffer-depth of 4 positions, handshake control flow, executing remotely an AES application (see Section III A). The application instructions are put in the top left PE, and accessed either by its right neighbor (1 hop remote access), or by the diagonal PE (two hops remote access). Table 1 shows the latency of read and write remote memory accesses, varying the NoC channel width in 4, 8, 16 and 32 bits, when 8 words of 32 bits (data size) are considered.

TABLE 1. MEMORY ACCESS TIME (IN CLOCK CYCLES) FOR READ AND WRITE OPERATIONS REGARDING DIFFERENT (DIFF.) CHANNEL WIDTH AND # OF HOPS.

	Read			Write		
Channel Width	**1 hop**	**2 hops**	**diff. (%)**	**2 hops**	**1 hop**	**diff. (%)**
4 bits	1052	1194	13.50	1308	1377	5.28
8 bits	560	638	13.93	702	748	6.55
16 bits	308	354	14.94	391	422	7.93
32 bits	182	212	16.48	231	251	8.66

III. EXPERIMENTAL RESULTS

This section evaluates the proposed remote execution, as well as compares it to a task migration mechanism validated in [12]. In order to fairly assess the performance overhead induced by those two mechanisms, the first set of experiments are conducted in best effort mode, therefore all implementations aim at maximizing performance rather than ensuring quality of service. This allows stressing the architecture as much as possible, hence migration / remote execution cost becomes prominent rather than compensated by transient increase in CPU usage. All applications were first profiled in order to capture a reference CPU workload, which guides possible task migration and remote execution configurations.

Finally, as most embedded applications require real-time execution, the second set of experiments focuses on QoS issues using a streaming application as case study.

A. Experimental Setup

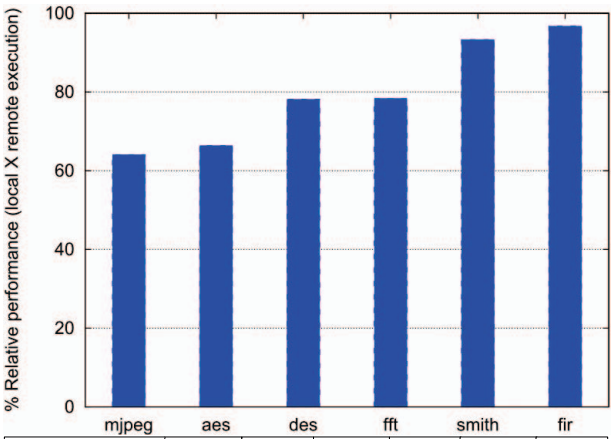
Real applications were employed for evaluating both task migration and remote execution mechanisms:

- *MJPEG decoding:* A quick profiling shows the average percentage of CPU time of each task in a sequential execution: SENDER only 2%, IVLC take around 85% of the CPU time, while IDCT 8% and IQUANT 5%,
- *Smith-Waterman*: used to find similar regions between two DNA sequences,
- *DES (Data Encryption Standard)* and *AES* (Advanced Encryption Standard) cryptography application,
- *FFT* (*Fast Fourier Transform*) and *FIR* (*Finite Impulse Response*), which are matrix computation application.

The platform is configured as follows: (*i*) NoC with 4 positions input buffer and 32 bits channel width, (*ii*) NoC router and CPU running at 50MHz, (*iii*) CPU using hardware multiplier, divider and barrel shifter, (*iv*) default cache size set to 16kB, 8 words per lines. Such configurations were used in best-effort and QoS scenarios.

B. Experiments in Best-Effort Scenarios

The first experiment evaluates the performance of six applications considering the number of caches misses during this execution. Adopted applications present different computation time and diverse behaviors in terms of memory accesses. Figure 4 shows resulting performance, normalized versus local execution (equivalent to task migration). The performance overhead varies between 37% and 4% depending on the application. Cache miss rates are indicated for each application, showing a direct link between the efficiency of remote execution and number of cache misses.



	mjpeg	aes	des	fft	smith	fir
# of cache miss	**3.97%**	**3.21%**	**1.74%**	**1.28%**	**0.64%**	**0.31%**

Figure 4 – Performance results of remote execution normalized versus local execution and corresponding cache miss rates.

C. Experiments in QoS Scenarios

The MJPEG streaming application was employed to compare the throughput gain using: (*i*) task migration (TM), and (*ii*) remote execution (RE). Figure 5 shows the initial (a) and final configurations (b) regarding the task migration experiment; (c) depicts the same for remote execution.

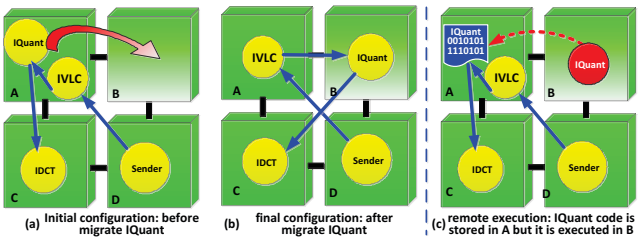


Figure 5 - Task configurations used in the throughput evaluation.

In the initial configuration, the *Sender* and the *IDCT* tasks are executed in different nodes (C, D), while *IVLC* and *IQuant* tasks are executed in the same node (A), making this node overloaded. In the final configuration, each task is executed in a different node, making these tasks being executed in parallel, resulting in a better throughput. Figure 6 shows the transient profile of application throughput obtained by both techniques (2 simulations), both TM and RE are triggered at the same time. The throughput of both is maintained until the migration

start point, which comprises TM and RE protocols initialization. Task migration leads to a loss in performance due to the required time to transfer task code from PE A to PE B. During this period, the TM throughput is almost half that of RE, which achieves an average throughput of 400 kB/s, just after 4ms. However, after code transfer is completed, TM performs better than RE, since *IQuant* is executed locally.

As real-time embedded systems must guarantee strict timings, we abstracted a complete video application through inserting a buffer at the last stage of the video pipeline. Data are read out of this buffer at periodic time intervals so as to model video frames displayed on a monitor at a given frame rate. In order to meet the real-time constraint; that buffer should never run empty so as to ensure smooth video decoding. Readout is set at 300kB/s, corresponding to the performance of the initial mapping. Figure 7 shows the buffer occupation regarding the application throughput.

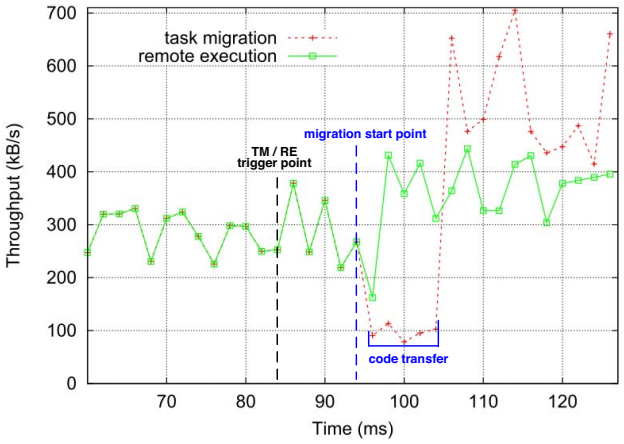


Figure 6 - Throughput of MJPEG for task migration and remote execution.

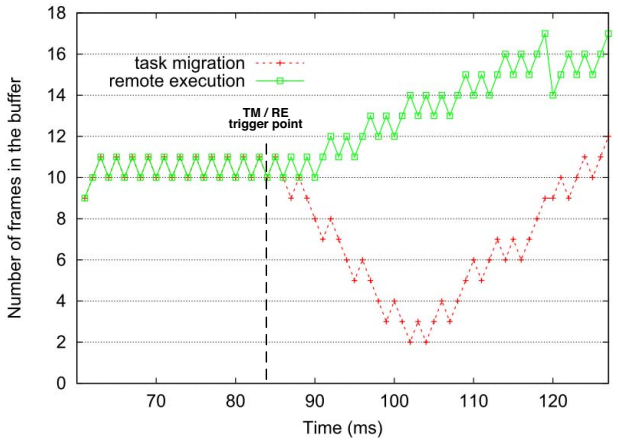


Figure 7 – Buffer occupation for task migration and remote execution.

As shown in Figure 7 the buffer is stable ranging from 10 to 11 frames in accordance with the production and consumption of these frames. After the trigger point, it is observed that in the case of TM, the buffer occupation decreases, a buffer depth of 8 positions being required so as to avoid any break in the data stream. In turn, in RE there is no buffer drop and it can reach the throughput constraint whatever the number of elements stored in the buffer before its execution.

IV. Conclusions and Future Works

This paper presented the remote execution mechanism, which can be used to optimize performance parameters at run-time, aiming to satisfy different system requirements is more efficient for tasks having a small cache-miss rate during execution, and with a platform having low NoC activity. It also proved a better reactivity during the transient migration phase. On the other hand, task migration achieved better long-term performance and scalability. The long-term goal will be to conduct experiments with dynamic selection of the best suited load-balancing techniques with respect to either application specifications (e.g. average cache miss rates of each task) or local parameters (e.g. CPU workload). Future work also includes the evaluation of other performance figures such as power dissipation.

References

[1] Marculescu, R.; et al. “Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28(1), 2009, pp. 3–21.

[2] Singh, A. K.; at al. “Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms”. Journal of Systems Architecture, vol. 56(7), 2010, pp. 242–255.

[3] Chou, C.-L. and Marculescu, R. “Run-Time Task Allocation Considering User Behavior in Embedded Multiprocessor Networks-on-Chip”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29(1), pp. 78–91.

[4] Streichert, T.; et al. “Dynamic task binding for hardware/software reconfigurable networks”. In: Symposium on Integrated Circuits and System Design (SBCCI), 2006, pp. 38–43.

[5] Bertozzi, S.; et al. “Supporting task migration in multi-processor systems-on-chip: A feasibility study”. In: Design, Automation and Test in Europe Conference (DATE), 2006, pp. 1–6.

[6] Pittau, M.; et al. “Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation”. In: Embedded Systems for Real-Time Multimedia, 2007, pp. 59–64.

[7] Shen, H. and Pétrot, F. “Novel task migration framework on configurable heterogeneous MPSoC platforms”. In: Asia and South Pacific Design Automation Conference (ASP-DAC), 2009, pp. 733–738.

[8] Mulas, F. ; et al. “Thermal balancing policy for multiprocessor stream computing platforms”. IEEE Transaction on Computer-Aided Design of Integrated Circutsand Systems, vol. 28(12), 2009, pp. 1870–1882.

[9] Barcelos, D.; et al. “A hybrid memory organization to enhance task migration and dynamic task allocation in NoC-based MPSoCs”. In: Symposium on Integrated Circuits and System Design (SBCCI’07), 2007, pp. 282–287.

[10] Goodarzi, B. and Sarbazi-Azad, H. “Task Migration in Mesh NoCs over Virtual Point-to-Point Connections”. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2011, pp. 463-469.

[11] Busseuil, R.; et al. “Open-Scale: A Scalable, Open-Source NOC-based MPSoC for Design Space Exploration”. In: International Reconfigurable Computing and FPGAs (ReConFig), 2011, pp. 357 - 362.

[12] G. Marchesan Almeida, et al. “Evaluating the impact of task migration in multi-processor systems-on-chip”. In: Symposium on Integrated Circuits and System Design (SBCCI’10), 2010, pp. 73–78.

[13] Weicker, R.P. “Dhrystone benchmark: rationale for measurement rules”. ACM SIGPLAN Notices, vol. 23(8), 1988, pp. 49-62.