Proposal and Evaluation of a Task Migration Protocol for NoC-based MPSoCs

Fernando G. Moraes, Guilherme A. Madalozzo, Guilherme M. Castilhos, Everton A. Carara
PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 – Brazil
{fernando.moraes, everton.carara}@pucrs.br, {guilherme.madalozzo, guilherme.castilhos}@acad.pucrs.br

***Abstract*—Task migration is a well-known strategy adopted in distributed systems for load balancing. but the adoption of such strategy in NoC-based MPSoC is scarce in the literature. This paper proposes a complete task migration protocol for NoC-based MPSoCs. The migration transfers the task code, data and context to another PE. The paper presents the communication strategy to ensure coherence in the messages delivery, the heuristic to compute the new task location, and the procedure to inform the new task position. Results evaluate the cost of the task migration using a real MPSoC (described in synthesizable VHDL), demonstrating that the cost to migrate a given task has a small impact in the system performance, enabling its use to improve the overall system performance.** *(Abstract)*

I. Introduction and Related Work

Task migration, the process of transferring a running task from one processing element (PE) to another, has been extensively researched in the field of distributed computing in the past and is now regaining crucial importance in the domain of embedded systems, specially MPSoCs. These architectures need to adjust their behavior based on runtime events, difficult to predict. Examples of runtime adaptability include fault tolerance, workload balance and thermal management.

Works [1][2][3] evaluate task migration in bus-based MPSoCs. Acquaviva et al. [1] propose a technique for reducing the on-chip temperature gradient based on task migration between processing cores. The algorithm, named MiGra, monitors the temperature and the frequency of each processor at run-time to balance the chip temperature. The algorithm improves system reliability without impacting energy dissipation and minimizing migration costs. The Authors present the conditions to fire the migration, but do not detail the way migration is executed. Pittau et al. [2] present a middleware infrastructure for task migration, targeting multimedia applications. The Authors evaluate deadline misses and energy costs, demonstrating that inter-process communication buffers can hide the impact of migration overhead. Aguiar et al. [3] evaluate the cost to migrate tasks comparing the performance of centralized and distributed load monitors, arguing that the including a dedicated load monitor in the operating system of each PE reduces the migration cost.

Jahn et al. [4] present CARAT, a context-aware runtime adaptive task migration mechanism. CARAT provides an adaptive mechanism at run-time, which balances latency and bandwidth requirements, decreasing in this way the migration latency allowing frequent migrations without impacting negatively in the system performance. The Authors compare their approach to post-copy, pre-copy and lazy-copy [5] mechanisms, running the applications x264 encoder, 7zip compressor and a robotic application. CARAT achieved the lowest task migration latency compared to the others methods. The communication infrastructure is not detailed in the paper.

Goodarzi et al. [6] propose a methodology based in virtual channels (VIP – virtual point-to-point) to create connections that provide low latency and low power paths for the task migration flows. The Authors adopts a 2D-mesh NoC, creating sub-meshes which may contain one or more cores. They compared the migration cost to two other methods: Gathering-Routing-Scattering [7] (GRS) and Diagonal [7]. All task migration strategies were implemented in a NoC simulator. The proposed method reduced the average message latency by 13%, average migration latency by 14%, and total network power by 10% with respect to the GRS method.

Almeida et al. [8] present a NoC-based MPSoC, using task migration to balance workload at run-time. Each processor has a set of running tasks, and the operating system (RTOS) of each processor is responsible to monitor the deadlines and to decide to migrate a given task to a neighbor processor. The goal is to improve the performance of the system, ensuring scalability of the approach due to the distributed control. The MPSoC is described in SystemC (PEs) and VHDL (NoC), and a MJPEG decoder is used to demonstrate the gains in terms of throughput when task migration is used. The drawback of the approach is the task is migrated without its context.

The *goal* of the present work is to propose a task migration algorithm, evaluating its cost in a real NoC-based MPSoC. The proposed task migration algorithm includes: (*i*) the migration protocol, which ensures correct task and context migration; (*ii*) the algorithm to choose a PE to receive the migrated task, minimizing the NoC communication energy; (*iii*) the control of messages inside the NoC ensuring in-order message delivery after the migration. The conditions to start the task migration is out of the scope of the present work, and may be managed by hardware or software mechanisms, that monitor some performance parameter such as latency, throughput or temperature.

The *originality* of this work is twofold. Firstly, the task migration process for a NoC-based MPSoC is detailed, including the algorithm to select the PE able to receive a new task. It is important to mention that the proposed mechanism does not require migration checkpoints, condition required by most migration methods [9][10]. Secondly, the evaluation is carried out in a real MPSoC, instead of high-level models, presenting cycle accurate results.

II. MPSoC Architecture Assumptions

The MPSoC architecture adopted in the present work is homogeneous, with a 2D mesh NoC interconnecting PEs [11]. Each PE contains a 32-bit RISC processor, a network interface to the NoC, a DMA, and a private memory for code and data. One PE, *master-PE*, is responsible to manage the MPSoC, executing task mapping, task migration (herein proposed), and has an interface with an external memory, which contains the applications binary codes. All other PEs, *slave-PEs*, execute user applications, being controlled by its own small preemptive multi-task operating system, named *microkernel*.

Applications are assumed to be represented using task graphs [12], $A=<T,C>$, where $T = \{t_1, t_2, ..., t_m\}$ is the set of application tasks corresponding to the graph vertices, and $C= \{(t_i, t_j, w_{ij}) \mid (t_i, t_j) \in T \text{ and } w_{ij} \in \mathbb{N}^*\}$ denotes the communications between tasks, corresponding to the graph edges. The edge weight w_{ij} in (t_i, t_j, w_{ij}) represents communication volume transmitted from t_i to t_j. All communication among tasks occurs through message passing, using two communication primitives: *Send()* and *Receive()*. The C code of a simple producer-consumer application is shown in Figure 1, while its task graph illustrated in Figure 2 (as a *disturbing application*). In this example task *n1* transmits 100 30-word messages to task *n2*.

```
#include<MPSoC.h>

int main(){
    message msg1;
    int i,j;

    msg1.length = 30;
    for(i=0; i<100; i++) {

     // initialize the message
     for(j=0;j<msg1.length;j++)
        msg1.msg[j]= j*10+i;

    // do something

    Send(&msg1,TASK_N2);
    }
    return 0;
}
```

(a) Task N1

```
#include<MPSoC.h>

int main(){
   message msg1;
   int i;

   for(i=0; i<100; i++) {
     Receive(&msg1,TASK_N1);

    // do something
   }

   return 0;
}
```

(b) Task N2

Figure 1 – C code illustrating a producer-consumer application.

Two relevant features of the MPSoC architecture for task migration need to be discussed: (*i*) memory organization; (*ii*) message passing mechanism. The memory may be organized in equally sized pages, or segments sized according to the task code length. Paging may limit the task size (one task per page), and waste memory resources due to the internal fragmentation [13]. Despite these drawbacks, the present work adopts a paging scheme, since it simplifies the search for available pages in *slave-PEs*. The only requirement for the task migration heuristic is to maintain a data structure with the available pages in the MPSoC.

Regarding the message passing mechanism, the present work adopts non-blocking write operations (*Send*) and blocking reads (*Receive*). When a given task executes a *Send()* primitive, the message is stored in a vector located in the microkernel memory area, named *pipe*, and then computation continues. The *pipe* is software implemented as a parameterizable array with random access. In this way, problems such as head-of-line blocking and deadlocks are avoided. When a *Receive()* is executed, the *microkernel* seeks the sender task location in a *task table*. If the sender task is located in the same processor, the receiver task simply executes a read in the *pipe*. If the sender task is located in another processor, the microkernel sends a *message request* through the NoC and the receiver task enters in *wait* state. When the sender message arrives from network, the microkernel stops the executing task and reschedules the waiting task (receiver). This communication protocol (*read request*) simplifies the task migration process due to the following features:

i. Due to the blocking nature of the *Receive()* operation, any *message request* need to be served before execute a new *Receive()*. Therefore, the sender only injects data in the network after receiving a *message request*. So there is no risk to have sender task messages being simultaneously transmitted from the original location and the new location (after migration), to the same receiver task.

ii. The *pipe* decouples the messages from the task, i.e. a given sender task may be migrated, and the messages addressed to other tasks may remain in the pipe, waiting to be requested.

iii. If the task is in the executing state there is no messages to be received (the *receive*() primitive puts the task in the wait state), ensuring a safe task state to perform the migration. This is a key feature, since it guarantees that none PE will send messages to migrating tasks.

One questionable choice adopted in the present work is the centralized approach. The Authors agree that this choice may penalize the performance, and also compromise fault tolerance. However, the migration process is not supposed to happens frequently, which avoids an overload in a central PE and results in a small impact in the overall performance. The solution to modify the centralized approach to a distributed one relies in partitioning the MPSoC in virtual clusters, with one PE responsible to manage the cluster and communicate with other clusters. The experiments presented later adopt a 6x6 MPSoC, which can be seen as a cluster of a larger MPSoC.

III. MOTIVATIONAL EXAMPLE

In this section, we show an example to motivate the techniques to be presented. Task migration may be used, as mentioned before, for load balancing, fault tolerance and to restore the performance of a given application due to, for example, the insertion of a new application in the system.

Figure 2(a) presents the mapping of a given application, composed by 6 tasks (A-F). In a dynamic scenario, where applications are inserted/removed at run time, the MPSoC may have most of its resources in use, restraining the search space of the mapping heuristic. At a given moment, new applications may be loaded in the MPSoC (disturbing applications in Figure 2(b)), with their communications competing with the communication of the main application. In this way, the performance of the main application is penalized, and the microkernel of the PE executing *task C* asks to the *master-PE* (M) to be migrated. The *master-PE* chooses the new position and the task is migrated to a position closer to its communicating tasks (Figure 2(c)), restoring/improving the application performance.

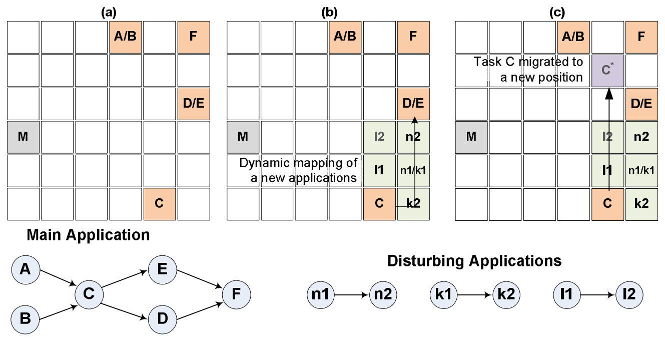

Figure 2 – Motivational example showing the benefits of task migration.

IV. TASK MIGRATION PROTOCOL AND HEURISTIC

The task migration protocol is illustrated in Figure 3 and may be summarized as follows:

1) Detection of a migration need, resulting in a task migration request to the master-PE. This detection is out-of-the-scope of the present work, requiring monitoring schemes.
2) The master-PE executes the heuristic to compute the new task position. Note that the task to be migrated (T1) continues its execution in parallel.
3) The slave-PE, which requested the task migration, receives the new task position.
4) The task can only be migrated *iif* it is in the *executing* state, i.e. the task cannot be waiting data from other task. The microkernel scheduler verifies this condition. If the task can be migrated, the microkernel sends to the target PE a packet with

the complete contents of the task page and the corresponding task descriptors (TCB – Task Control Block).

5) The migrated task (T1*) is scheduled once the code and TCB is completely received.

The performance of the migration protocol is a function of t_{req} and t_P, time to request migration and compute the new task position respectively; and t_{Mig}, time to transmit the task and re-schedule it in the new PE. During t_{req} and t_P the task to be migrated continues its execution. During t_{Mig} the task stops its execution. Therefore, t_{Mig} is critical in our method and it is evaluated in the next section.

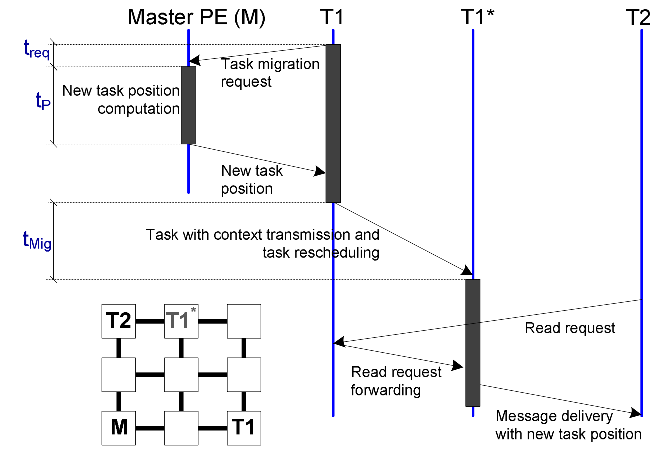

Figure 3 – Task migration protocol.

After migrate a given task, all tasks communicating with it should receive somehow its new position. The adopted *read request* communication protocol, presented in section II, exempts the method to propagate the new task position. The method works as follows. All *message requests* (generated by the *Receive()* primitive, see Figure 1) are sent to original task position. Due to the decoupling between messages and task, while there are messages to be transmitted in the *pipe*, they are transmitted to the receiver task, even if the sender task has a new location. So, the receiver task continues using the original sender task position. When the *pipe* of the original sender task position is empty, the *message request* is forwarded to the new sender task position (see Figure 3). The migrated task delivers the requested message, with its new position, allowing the receiver PE to update its task table.

A. *Task Migragion Heuristic*

The task migration heurist is executed in the master-PE, after receiving a *task migration request*. The proposed heuristic presented in Figure 4 is similar to a dynamic mapping process [14]. Line 1 computes the initial communication energy [15], with the task to be migrated in its original position. Assuming as example the main application task graph illustrated in Figure 2, the task to be migrated (t_M) is task C, which communicates with tasks A, B, D and E. In this way, the set of communicating tasks is N={A,B,D,E}. The heuristic, in line 4, defines the set of PEs inside the bounding box defined by N – function *processor box* - P. Lines 6-11 verifies if the PE in the center of the bounding box (p_c) may receive t_M, computing the communication energy between t_M, mapped in p_c, and all elements in N.

Next, lines 14-18, the other elements in P are evaluated to receive t_M. If a given PE in P may receive t_M, the chosen PE is the one that minimizes the communication energy (line 21). Otherwise, the search space is enlarged by one hop (line 23), and if new PEs may be added to P, the heuristics continues to search a position to t_M. If it is not possible to add elements in P, this means that all MPSoC resources are being used, and the migration is not possible.

```
Input: $t_M$: task to be migrated; $p_M$: PE executing $t_M$; $N$:set containing the tasks $t_M$ communicates with
Output: $p_T$: PE to receive the task $t_M$
1.  migration_cost = cost($p_M$,$t_M$,$N$)        // initial migration cost
2.  $p_T$ ← NULL
3.  // define the PE set that may receive $t_M$
4.  $P$ ← processor_box($N$)
5.  // define the PE address in the center of the set P
6.  $p_c$ ← center($P$)
7.  // if the central PE may receive $t_M$, the initial migration cost is computed, and $p_T$ receives $p_c$
8.  IF migrate($p_c$,$t_M$ )=TRUE and cost($p_c$,$t_M$,$N$)<cost($p_M$,$t_M$,$N$) THEN
9.      migration_cost= cost($p_c$,$t_M$,$N$)
10.     $p_T$ ← $p_c$
11. ENDIF
12. DO
13.     // verifies all other PEs in $P$
14.     FOR ALL ELEMENTS $p_i$ IN $P$
15.         IF cost($p_i$,$t_M$,$N$)<migration_cost THEN
16.             migration_cos t= cost($p_i$,$t_M$,$N$)
17.             $p_T$ ← $p_i$
18.     ENDIF
19.     // verifies if migration is possible in the set $P$
20.     IF $p_T$ ≠ NULL THEN
21.         return $p_T$
22.     ELSE
23.         $P'$ ← extend_search_space_1_hop($P$)
24.         IF $P' = P$ THEN
25.             return NULL          // migration is not possible
26.         ENDIF
27.         $P$ ← $P'$
28      ENDIF
29. ENDDO
```

Figure 4 – Task migration pseudo-code.

The presented heuristic only migrates a given task if the migration minimizes the communication energy, otherwise the task remains in its original position. The complexity of the heurist is O(N_{PE}), where N_{PE} is the number of PEs in the MPSoC. The processor load is not yet considered in the heuristic. If there is an available page to receive the task, the PE is a potential candidate to receive the new task. One way to add the processor load in the heuristic is to consider the t_M load, verifying in the cost function the impact of the t_M load in the total PE load, not allowing the migration if the total load is superior to a given threshold.

V. RESULTS

Results were obtained on a 6x6 instance of the HeMPS [11] MPSoC, described in synthesizable VHDL, resulting in clock-cycle accurate results. Relevant architectural parameters for task migration include:

- PE word and flit sizes: 32 and 16 bits respectively. The injection of a PE word in the NoC takes two clock cycles.
- Page size: 16 Kbytes (4,096 words, code and data).
- Task time slice: 16,384 clock cycles.
- Scheduler: round-robin, consuming in average 260 clock cycles. This represents a *microkernel* overhead inferior to 2% (260/16,384 clock cycles).
- NoC router: wormhole packet switching. XY routing, centralized round-robin arbitration, input buffering (buffer depth equal to 16 flits).

A. *Task migration cost*

The first set of evaluated parameters comprises t_{req} and t_P (Figure 3). The packet to request a task migration and the packet containing the new task position are small, 8 flits, being their latency proportional to the number of hops in the path. Considering that the routing and arbitration takes 5 clock cycles, a distance of 7 hops to the master (C-M, Figure 2) results in t_{req}=35 clock cycles, if there is no other traffic competing for the same resources. The time to compute the new task position, t_P, is in average 4,100 clock cycles. Therefore, the NoC latency has a very small impact in this step of the protocol. Also the time to compute the new task position can be considered small, since it is inferior to one time slice.

For the scenario presented in Figure 2, the migration of task C took 11,895 clock cycles (the packet size containing the task, TCB and some control variables is 8,590 flits). The total time spent between the last execution of task C in its original position, and the time it was scheduled in the new position (t_{Mig}) was 12,326 clock cycles. Note that t_{Mig} is related to the migrated task size. This result reveals that task migration has a small impact in the system performance, as it is demonstrated in the next subsection.

B. Evaluation using a synthetic application

The performance of the task migration protocol was evaluated according to the three scenarios presented in Figure 2:

a) *main application* executing alone in the MPSoC;
b) *main application* executing with 3 disturbing applications (producers-consumers);
c) same as scenario ´b´, with task C migrated after 2.1 ms.

The *main application* executes 150 interactions for all scenarios. Figure 5 presents the throughput of task F, in Mbps, for the first 40 interactions. The initial applications tasks (‘A’ and ‘B’) are mapped first, producing data for tasks not yet mapped. When the consumer tasks are mapped, they can consume faster, what explains the higher throughput observed in the first interactions. Scenario ‘a’ stabilizes the throughput in the 7th application interaction. Scenarios ‘b’ and ‘c’ stabilizes later, due to the higher number of tasks to be mapped (main application and disturbing applications). It is noticeable the throughput reduction in scenario ‘c’ (interaction 21), during the task migration period, and then the throughput reaches the same value of scenario ‘a’, scenario without disturbing applications.

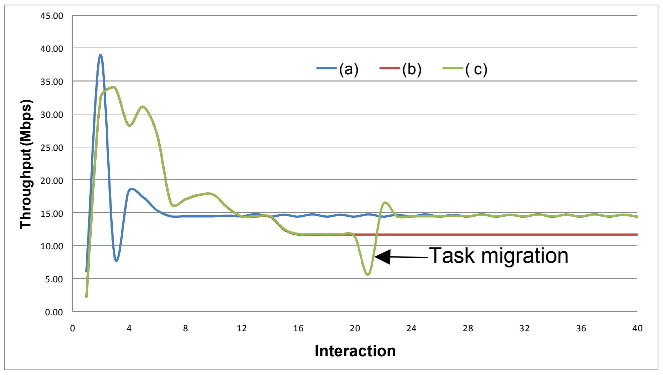

Figure 5 – Throughput of task F, in Mbps. (a) corresponds to the *main application* executing alone in the MPSoC; (b) *main application* with disturbing traffic; (c) same as ‘b’ with task C being migrated.

The throughput of the *main application*, without disturbing tasks is in average (measured after throughput stabilization – 30th interaction) 14.62 Mbps, dropping to 11.97 Mbps (-18.1%) when the disturbing applications are inserted. Such result demonstrates the impact of the NoC traffic in the overall system performance. Migrating task C closer to its communicating tasks, the average throughput becomes 14.63 Mbps, slightly superior to the original one, since task C is closer to its communicating tasks after migration.

The main application execution time, in clock cycles, was 1,337,172 (reference value), 1,626,812 (+21%) and 1,365,272 (+1.02%), for scenarios ‘a’, ‘b’ and ‘c’ respectively. As mentioned in the previous subsection, the expected overhead of the task migration in the execution time would be small. In the presented experiment the total execution time increased only 1.02%

VI. Conclusion and Future Works

The current paper presented, step-by-step, a task migration protocol. Task may be migrated at any moment, not requiring migration checkpoints, and its context is also migrated. Another important point of the protocol is the in-order message delivery, without requiring the broadcast of the new task position.

The throughput result was expected, since one of the goals of task migration is to restore the performance of a given application in the presence of some disturbing event. However, the main contribution of the present work is to answer the following question: “*How much it cost to migrate a task in a NoC-based MPSoC?*”. The cost is small, enabling its use to improve/restore the overall system performance. It was shown that the time to compute the new task position may be neglected, since such computation is executed in parallel with the task to be migrated. The amount of time the task is interrupted to be migrated can also be considered small, since it corresponds roughly to one time slice. This time can increase if the task size increases, but it will not exceed few time slices.

Future work lies in including in this approach monitoring schemes to detect when task migration is required. Such monitoring may include hot-spot detection (congested NoC regions), overloaded processors, application fragmentation (communicating tasks not mapped closer to each other). It is also important to include in the approach: (*i*) processor load; (*ii*) decentralized control; (*iii*) evaluation with real benchmarks.

Acknowledgments

The Author Fernando Moraes acknowledge the support of CNPq and FAPERGS, projects 301599/2009-2 and 10/0814-9, respectively.

References

[1] A. Acquaviva, S. Carta, F. Mereu, G. Micheli. “*MiGra: a task migration algorithm for reducing temperature gradient in multiprocessor systems on chip*”. In: SoC, 2007, 6p.

[2] M. Pittau, A. Alimonda, S. Carta, A. Acquaviva. “*A. Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation*”. In: ESTImedia, 2007, p.59-64.

[3] A. Aguiar, S. J. Filho, T. G. Santos, C. Marcon, F. Hessel. “*Architectural support for task migration concerning MPSoC*”. In: WSO Workshop, 2008, pp.169-178.

[4] J. Jahn, M. A. Faruque. “*CARAT : Context-Aware Runtime Adaptive Task Migration for Multi Core Architectures*”. In: DATE11, 2011, 6p.

[5] M. Richmond, M. Hitchens, “*A new process migration algorithm*”. In: SIGOPS, 1997, pp. 31-47.

[6] B. Goodarzi, H. Sarbazi-Azad. “*Task Migration in Mesh NoCs over Virtual Point-to-Point Connections*”. In: EUROMICRO, 2011, pp.463-469.

[7] T. S. Chen. “*Task migration in 2D wormhole-routed mesh multicomputers*”. Information Processing Letter, 2000, pp.103-110.

[8] G. M. Almeida, S. Varyani, R. Busseuil, G. Sassatelli, P. Benoit, L. Torres. “*Evaluating the Impact of Task Migration in Multi-Processor Systems-on-Chip*”. In: SBCCI, 2010, pp. 73-78.

[9] V. Nollet, P. Avasare, J-Y. Mignolet, D. Verkest. “*Low Cost Task Migration Initiation in a Heterogeneous MPSoC*”. In: DATE, 2005, pp. 252-253.

[10] S. Bertozzi, A. Acquaviva, D. Bertozzi, A. Poggiali. “*Supporting Task Migration in Multi-Processor System-on-Chip: A Feasibility Study*”. In: DATE, 2006, 6p.

[11] E. Carara, R. Oliveira, N. Calazans, F. Moraes. "*HeMPS - a Framework for NoC-based MPSoC Generation*". In: ISCAS, 2009, pp. 1345-1348.

[12] C. Marcon, N. Calazans, E. Moreno, F. Moraes, F. Hessel, A. Susin. “*CAFES: A Framework for Intrachip Application Modeling and Communication Architecture Design*”. Journal of Parallel and Distributed Computing, v.71(5), pp. 714-728, 2011.

[13] A. D. Patterson, J. L. Hennessy. "*Computer Architecture: A Quantitative Approach*". Morgan Kaufmann Publishers, 4th ed., 2007.

[14] M. Mandelli, L. Ost, A. Amory, F. Moraes. "*Multi-Task Dynamic Mapping onto NoC-based MPSoCs*". In: SBCCI, 2011, pp. 191-196.

[15] J. Hu, R. Marculescu. “*Energy-aware mapping for tile-based NoC architectures under performance constraints*”. In: ASP-DAC, 2003, pp. 233-239.