Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs

Ewerson Carvalho Ney Calazans Fernando Moraes
Faculty of Informatics (FACIN)
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga 6681- Porto Alegre - 90619-900 – Brazil
{ecarvalho, calazans, moraes}@inf.pucrs.br

Abstract

Multiprocessor Systems-on-Chip (MPSoCs) is a trend in VLSI design, since they minimize the "design crisis" (gap between silicon technology and actual SoC design capacity) and reduce the time to market. Important issues in MPSoC design are the communication infrastructure and task mapping. MPSoCs may employ NoCs to integrate multiple programmable processor cores, specialized memories, and other IPs in a scalable way. Applications running in MPSoCs execute a varying number of tasks simultaneously, and their number may exceed the available resources, requiring task mapping to be executed at run-time to meet real-time constraints. Most works in the literature present static MPSoC mapping solutions. Static mapping defines a fixed placement and scheduling, not appropriate for dynamic workloads. Task migration has also been proposed for use in MPSoCs, with the goal to relocate tasks when performance bottlenecks are identified. This work investigates the performance of mapping heuristics in NoC-based MPSoCs with dynamic workloads, targeting NoC congestion minimization, a key cost function to optimize the NoC performance. Here, tasks are mapped on the fly, according to communication requests and the load in the NoC links. Results show execution time and congestion reduction when congestion-aware mapping heuristics are employed.

1 Introduction

While single processor may be sufficient for low-performance applications, that are typical of early micro-controllers, an increasing number of applications require multiprocessors to meet their performance goals [1]. Multiprocessor Systems-on-Chip (MPSoCs) are custom architectures that balance the constraints of VLSI technology with application requirements.

An important issue in MPSoC design is the communication infrastructure implementation. Networks-on-Chip (NoCs) are an alternative to busses, with several advantages, including scalability and shorter wires, which minimizes power consumption. MPSoCs may employ NoCs to integrate multiple programmable processor cores, specialized memories, and other intellectual property (IP) components on a single chip. MPSoC heterogeneity increases with IP integration and component customization to optimize performance and power consumption.

Applications running in heterogeneous MPSoCs, as multimedia and networking, normally contain a dynamic workload of tasks. This implies a varying number of tasks simultaneously running, with their number possibly exceeding the available resources. This may require the execution of task mapping at run time, to meet real-time constraints.

Most works in the literature propose *static mapping* solutions [2][3]. Static mapping tries to define the best placement of tasks at design time, for a given application. Consequently, such methods are not appropriate for dynamic workloads. *Task migration* [4][5] has also been used in heterogeneous MPSoCs to optimize the performance at run-time. Task migration relocates tasks either when a performance bottleneck is detected or to distribute the workload more homogeneously among the MPSoC processors. Differently from task migration, *dynamic mapping* can insert new tasks into the system at run time.

This work investigates the performance of different mapping algorithms in NoC-based MPSoCs with dynamic workload. The main cost function in mapping algorithms is to optimize the occupation of the NoC links. It is possible to achieve performance gains if the mapping algorithm is able to minimize NoC congestion.

The paper is organized as follows. Section 2 presents related works on task mapping. Section 3 presents the MPSoC architecture. Section 4 presents task mapping algorithms. Section 5 presents the experimental setup and the results. Finally, Section 6 presents some conclusions and directions for future work.

2 Related Works

Usually, task mapping is a two-step process. The first step, *task binding*, selects the set of IPs able to execute a

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

given task. The task may be, for example, a software task to be executed in an embedded processor, a hardware task to be executed in an embedded FPGA or in a co-processor, or an input/output operation. Next, *task placement*, the feasible placements are evaluated according to a specific cost function, as area fragmentation, power consumption or link usage, in order to get the best performance.

Examples of generic task placement are presented in [6], [7], and [8]. These authors represent the system area as a two-dimensional matrix, developing methods to manage the used and free spaces. Tabero et al. [8] presents two placement heuristics: tasks exchanging messages are placed as nearest as possible, and area fragmentation minimization. These works are target to homogeneous MPSoC (all IP are identical, not requiring task binding) and the communication infrastructure are not considered.

A second group of works present static mapping algorithms for bus-based ([9]) and NoC-based MPSoCs ([2][3][10]). Even if these procedures are inadequate for MPSoC with dynamic workload, such methods may be used to compute the initial task placement, or be optimized to be employed at run-time.

Ruggiero et al. [9] employs a homogeneous platform, composed by ARM processors interconnected by an AMBA bus. Applications are modeled as pipelined task graphs, targeting stream-oriented tasks (e.g. multimedia). They solve the task mapping in a two-step process: task allocation followed by task scheduling.

Hu and Marculescu [2] present a static mapping heuristic. The system area is defined as a tile-based NoC architecture. The authors propose a mapping approach named Communication Weighted Model (CWM), modeling applications as graphs, being the vertices the tasks and the edges the communication between tasks. The weight of each edge corresponds to the number of bits exchanged between tasks. The main goal of the approach is to reduce the overall power consumption by decreasing the consumed energy on communication.

Marcon et al. [3] extend the work of Hu and Marculescu, proposing the Communication Dependence Model (CDM) and the Communication Dependence and Computation Model (CDCM). The CWM considers only the communication volume. CDM includes in the model the dependence between messages, being able to consider NoC congestion. The CDCM [10] capture the volume and the timing of application communication.

Task migration algorithms, widely used in distributed systems and parallel computing domain, are been used in MPSoC to improve the overall system performance. Works [4], [5] and [11] discuss the cost to interrupt a given task, save its context, transmit all data to a new IP, and restart the task in the new IP. Nollet et al. [4] propose to migrate tasks from hardware (in reconfigurable areas) to software (in processors), and vice-versa, in function of the required application performance. The method proposed by Nollet uses task *migration points* to define when a given task can be migrated to other IP. Bertozzi et al [5] use *checkpoints*, similar to Nollet, to define when a given task may be migrated. A middleware implemented on uClinux allows migration at specific *migration points* represented by *checkpoints* inserted manually by the system user on tasks code. Kalte et al. [11] propose two methods for context saving for reconfigurable systems *shutdown process* and *readback* operations to save task state.

This paper discusses dynamic task mapping. A cost function, as communication volume or initial tasks, defines the initial task mapping. When a task sends a message, two situations may arise: the target task is already assigned to a given IP, or the task is not yet assigned. In the first case, the message is transmitted to the target task. In the second case, the task is mapped at run-time, considering the task nature (hardware or software) and the communication cost, and then the message is sent. Task migration is not yet considered, since tasks are mapped in such a way the workload is homogeneously distributed in the MPSoC.

3 MPSoC Architecture

Without loss of generality, heterogeneous MPSoC architectures may be represented as a set of processing nodes which interact via a communication network. Routers (*R*) compose the network. Processing nodes may support either hardware or software task execution. Hardware tasks execute in reconfigurable logic (*reconfigurable areas*) or dedicated IPs. If reconfigurable logic is used, the hardware presents flexibility similar to software. It becomes possible to load hardware tasks on-the-fly using dynamic reconfiguration. Software tasks execute in *instruction set processor* or ISPs. Fig. 1 illustrates the MPSoC architecture model used in this work.

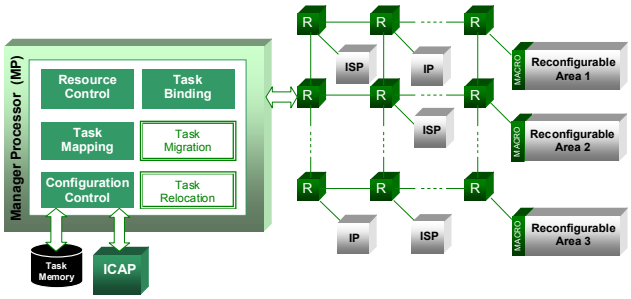

Fig. 1 - MPSoC conceptual organization.

The number of tasks may exceed MPSoC resources. One processor is reserved to manage the system resources (*MP* - Manager Processor). When the MPSoC starts its execution, only initially needed tasks are allocated into the

system. New tasks are allocated when a given task tries to communicate with a task not yet present. The *MP* is responsible for resource control, task binding, task mapping, task relocation/migration and to control the reconfiguration process.

All inter-task communications use messages transmitted through the network. There are four message types: request, release, notify and general. A request message to the MP contains the identification of a new task to be inserted into the system and the communication rates. This communication request is equivalent to a "contract request" between a given task and the MP, in which the communication rates between the involved tasks are stipulated. The release message notifies the MP that a processing node has finished its current task, being possible to reuse the node to a new task. The notify message is sent by the MP to the master and the slave tasks, containing both tasks addresses. The general messages are used by tasks to send data to other tasks.

4 Mapping Heuristics

This Section presents a set of heuristics for dynamic task mapping into NoC-based MPSoC with dynamic workload. Before, *task* and *applications* definitions are presented for sake of clarity.

Definition 1: A *task* is a 5-tuple *t=(tid, pt, nt, ad, ls)* where: *tid* is the task identifier; *pt* presents the task processing time; *nt* the type of the task (hardware or software task); *ad* is the initial task address in the *task memory* (Fig. 1); and *ls* is a list of slave tasks. When a task t_M requests a task t_S from its list of slaves, t_M is assumed as *master* and t_S is assumed as *slave*.

Definition 2: Elements of the *list of slave tasks* are triples *ls=(sid, ms, mr)*, where *sid* is the slave task identifier; *ms* and *mr* are the *message rates* respectively sent and received to/from this slave (the rate is expressed as a percentage of the available link bandwidth). Previous models [2][3] consider only the communication volume. As the path taken by messages is not the same when transmitting and receiving data (assuming XY routing algorithm), both rates are relevant in the model.

Definition 3: An *application* is a acyclic directed graph *AG=<T, C>*, where the vertex set *T* is a *set of tasks* and the edge set *C* is composed by triples with the form *(mid, sid, (ms, mr))*, where *mid* is the master task identifier, *sid* is the slave task identifier and *(ms, mr)* are the message rates as described in Definition 2 above. Each element in C corresponds to a pair of communicating tasks. Only one of the tasks in *T* is the *initial task*. The initial task cannot be the end point of any *C* edge. This means that this task has no master.

To simplify the model, and due to the scarce resources in reconfigurable logic, hardware tasks are always leaf vertices (they have no slaves). Processors may simultaneously run different tasks, if a multi-task kernel is employed, making software resources virtually abundant.

Definition 4: A *NoC* is a undirected graph *NG=<R,L>*, where the vertex set is a set of ordered pairs, *R={(ad, ty)}*. The elements *ad* are taken from a *set of routers* and represent the unique router address, while the information *ty* concerns the type of IP connected to the router (e. g. hardware or software IPs). *L* is the edge set, composed by pairs of addresses, representing the NoC *communication links*.

In the above definition, each link may represent any number of channels connecting two given routers, allowing to model e.g. unidirectional or bidirectional NoCs, regular or irregular topologies, etc.

4.1 Initial Task Mapping

The method used to define the *initial task mapping* has a significant impact in the performance of the dynamic mapping. As an example consider Fig. 2, where *AjTi* represents task *i* from application *j*. Tasks with label T00 are initially mapped and subsequent identifiers (T01, T02) are tasks allocated later. Applications with different identifiers do not communicate. If mapping is done randomly, it is possible that at run-time no free resources exist near some master when it needs a slave to communicate with. For this reason, communication may take more channels than necessary as in case (a). Case (b) shows the result of a more careful mapping. The method adopted here is to define clusters for each application, placing the initial task of each application inside one cluster.

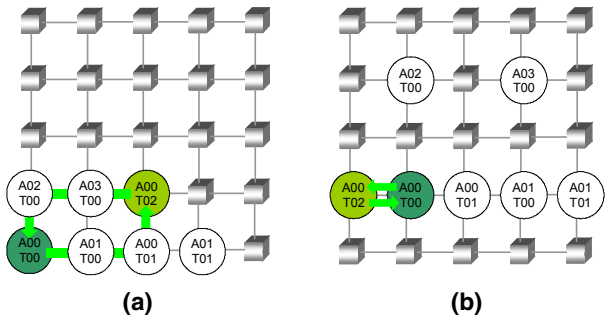

Fig. 2 - Initial mapping strategies.

4.2 Dynamic Mapping

Once *initial tasks* start their execution, communication requests are transmitted to the MP. A dynamic mapping heuristic is required to map the new tasks. This Section presents five heuristics to map tasks at run-time.

4.2.1 First Free

The *first free* (FF) method is a mapping used as a reference for comparison purposes only. This approach starts at the NoC address (0,0) and selects the first free node able to execute the requested task (task binding) when walking the network in a column by column fashion. There is no cost evaluation.

4.2.2 Nearest Neighbor

Nearest Neighbor (NN) mapping is similar to FF strategy, also with no cost evaluation. NN mapping starts searching for a free node able to execute the requested task (task binding) around the node making the request. The search tests all *n*-hop neighbors, *n* varying between 1 to the NoC limits.

4.2.3 Minimum Maximum Channel Load

Minimum Maximum Channel Load (MMC) congestion-aware mapping heuristic tries to reduce the maximum occupation of the NoC links. The goal of this heuristic is to avoid congestion in the NoC, improving overall performance. MMC computes the cost of each mapping *k* according to Equation 1, where $rate_{l(i,j)}$ denotes the total rate of each NoC link, and *lx* and *ly* denote NoC dimensions. The links with increased rates *ms* and *mr* are those used in the communication master→slave (*ms*) and slave→master (*mr*). The selected mapping is that with minimum cost.

$$cost_k = \max_{l(i,j)}(rate_{l(i,j)}), \quad 0 \le i < lx; \ 0 \le j < ly \qquad (1)$$

4.2.4 Minimum Average Channel Load

Minimum Average Channel Load (MAC) congestion-aware heuristic aims to reduce the average occupation of the NoC links. This heuristic is similar to the MMC, replacing the *max* function by the *avg* (average) function. While the MMC heuristic minimizes the peak link usage, the MAC heuristic homogenously distributes the communication load into the NoC. Equation 2 presents the MAC cost function. Links not used for communication ($rate_{l(i,j)}$=0) are not considered in the heuristic. The selected mapping is the one that has the minimum cost.

$$cost_k = \mathbf{avg}_{l(i,j)}(rate_{l(i,j)}) \ \forall \ rate_{l_{(i,j)}} > 0, 0 \le i < lx; \ 0 \le j < ly \qquad (2)$$

The MMC and MAC heuristics consider all NoC links while mapping a new task. This approach may cause the choice of a bad mapping, as illustrated in Figs. 3 and 4. Fig.3 assumes the use of MMC. When computing each possible mapping of B to an available place (nodes drawn as cubes in the Figure) the addition of channel rates does not change the maximum computed cost. This occurs because this addition does not change the computed maximum occupation (90%). Thus for both mappings of B the computed $cost_k$ will be the same. Due to this any mapping may be chosen, but mapping (b) is better than (c) in the example.

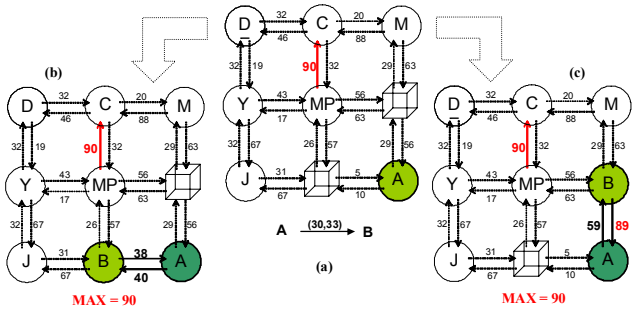

Fig. 3 – Problems using MMC. Both (b) and (c) alternatives present a 90% maximum value, but (b) is better.

The second situation illustrates possible problems when using the MAC heuristic for mapping. According to Fig. 4, there are cases when the minimum average does not represents the best mapping. This occurs because a mapping with a few channels heavily occupied can result in a smaller average when compared to a mapping with many channels with little occupation.

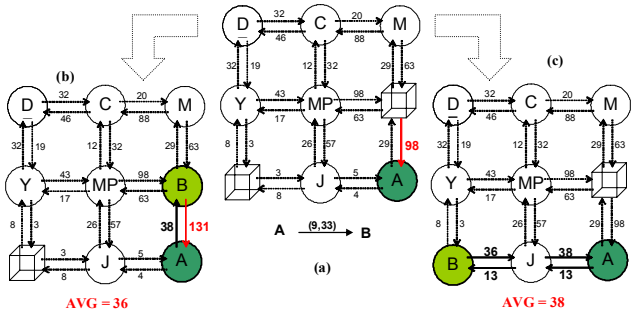

Fig. 4 – Problems using MAC. Alternative (a) is chosen as best because it leads to the smaller average, but it may lead to congestion since the channel has 131% occupation.

4.2.5 Path Load

To overcome MMC and MAC problems, the *Path Load* (PL) congestion-aware heuristic considers only the links that will be used by the task being mapped. *PL* computes the cost of each mapping *k* according to Equation 3, where $rate_{c(i,j)}$ and $rate_{c(j,i)}$ are the rates in the individual channels from the master to the new slave and the rates of the channels in the opposite direction. This is due to the asymmetric nature of the XY routing algorithm. The selected mapping is the one that has the minimum cost.

$$cost_k = \sum rate_{c(i,j)} + \sum rate_{c(j,i)} \qquad (3)$$

5 Results

This Section presents the experimental setup, including the system, NoC and application modeling; followed by the results.

5.1 Experimental Setup

Fig. 5 presents two examples of applications. Each application is modeled as a graph, being white vertices software tasks and black vertices hardware tasks, with one initial task (represented with double lines). Communication between tasks is represented by rates from/to the master node (see definition 3 – Section 4).

All presented results use the applications of Fig. 5, each one composed by 10 tasks, including 1 initial, 6 software and 3 hardware tasks. The number of applications varies from 1 to 100. Half of applications are modeled with the left graph of the Fig. 5, and the other half with the right graph. The processing time (pt) is randomly chosen between 10 and 40 microseconds, and the *ms* and *mr* rates according to the rates of Fig. 5.

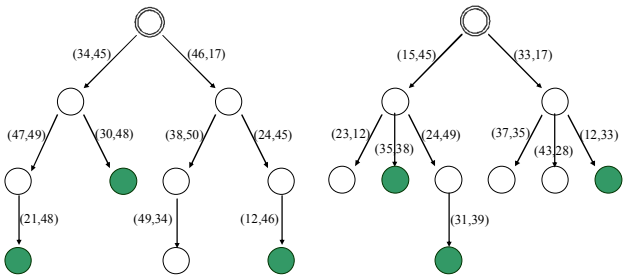

Fig. 5 - Examples of applications.

It is important to mention that the MP does not know the application graphs, it knows only the initial tasks. The MP starts the initial tasks, and dynamically maps new slave tasks according to requests sent by the tasks.

The system is modeled using SystemC. Two different threads model the system: MP (manager processor) and task. The MP thread is responsible for resource management, task scheduling, task mapping, and task configuration. This thread contains data structures to manage system use (i.e. links and resources) and scheduling queues. The task thread implements the tasks behavior. The number of task threads in the system is equal to the total number of the tasks in all applications.

All tasks threads have the following behavior:

1. The task thread waits to be started by the MP.
2. If the task is not an initial task, it waits a NOTIFY message, containing the master address and the communication rates (*ms,mr*), to start its execution. The NOTIFY message also implies successful task binding and task placement.
3. If the task is not an initial task, it should receive all GENERAL messages from the master. The number of messages is computed from the communication rate.
4. The task thread sends REQUEST messages to the MP, one for each slave task.
5. The task thread starts the communication (GENERAL messages) with each slave task when a NOTIFY message with the slave address arrives from the MP. The number of parallel executing tasks is a function of the available resources.
6. When the task thread identifies that all messages with a given slave were transmitted, it sends a RELEASE message to the MP with the slave address to be released.
7. When the task thread releases all slave tasks, the processing time (Definition 2) is added to the total execution time, and the task sends GENERAL messages to its master (if the task is not an initial task).

5.1.1 NoC Model

An 8x8 2D-mesh topology, modeled in SystemC, is responsible to transfer data between tasks. One node (the router with its IP) is used for the *MP* (Manager Processor), 16 nodes are hardware tasks (25% for reconfigurable logic) and 47 nodes are software tasks (75% for processors).

To avoid application deadlocks, due to the lack of resources, the number of simultaneously applications running in parallel is limited. For the experiments, this number is fixed in nine simultaneously active applications. The justification for this number to avoid application deadlock is outside the scope of this paper.

Fig. 6 presents the placement of the nodes, according to its type, as well as the positions reserved for initial placement. The platform has been divided in nine clusters and each cluster is composed by nine nodes. Each cluster corresponds to an independent application. However, an application can occupy resources of other clusters, if necessary, because the presented clusters limits in the Figure are virtual, not real. Hardware resources are placed in the network limits because hardware tasks are always leaf nodes.

In the current state of the work, processors execute only one task. Multi-task processors are subject for future work.

Each node has an input buffer to store messages. The message delay considers the number of hops to transmit a message according to the xy routing, and the link occupation. When a given link is saturated (occupation > 100%), the message delay is multiplied by a constant.

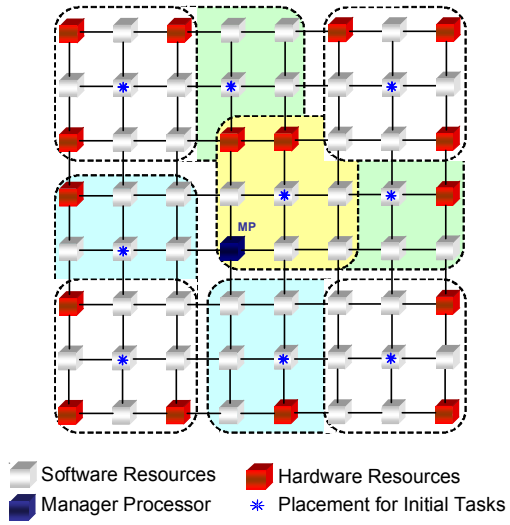

Fig. 6 - Type of nodes used in the NoC. Dashed lines denote the cluster limits.

5.2 Experimental Results

This Section evaluates the NoC occupation and execution time, using the heuristics presented in Section 4.

When internal links of the NoC do not present congestion, data is transmitted without interruptions. When congestion arrives in the internal links, task execution time and power consumption increase. The congestion minimization is a key cost function to optimize the NoC performance.

Graphs in Fig. 7 present the execution time and link occupation (average, maximum and minimum values) for the different mapping heuristics. These graphs are plotted for 13 applications.

The average NoC occupation is higher (± 80%) for the FF mapping, since no congestion-aware heuristic is used. The heuristics MAC and MMC have similar average NoC occupation. The NN mapping, a very simple heuristic, and the PL mapping present the lower NoC occupation. Note that the execution time of the applications running in the system, for NN and PL mappings, is smaller, corroborating the fact that reducing congestion the execution time is also reduced.

Even if two mapping heuristics present similar average occupation, it is important to analyze the maximum NoC occupation. The worst case is the FF mapping, used as reference (occupation peaks of 280%). The simple NN mapping, which leads to a small value of average occupation, has occupation peaks of 200%.

The MAC mapping is not effective in terms of congestion. The reason was advanced before: when a new mapping does not reduce the average link load, the algorithm selects the first mapping option available. The MMC has the same deficiency of the MAC, but with smaller values of maximum NoC occupation. The PL mapping presents the smallest values for the maximum NoC occupation. This arrives because this heuristic minimizes, for each mapping, the load added by the new task, in an opposite way to MAC and MMC which try to minimize globally the system.

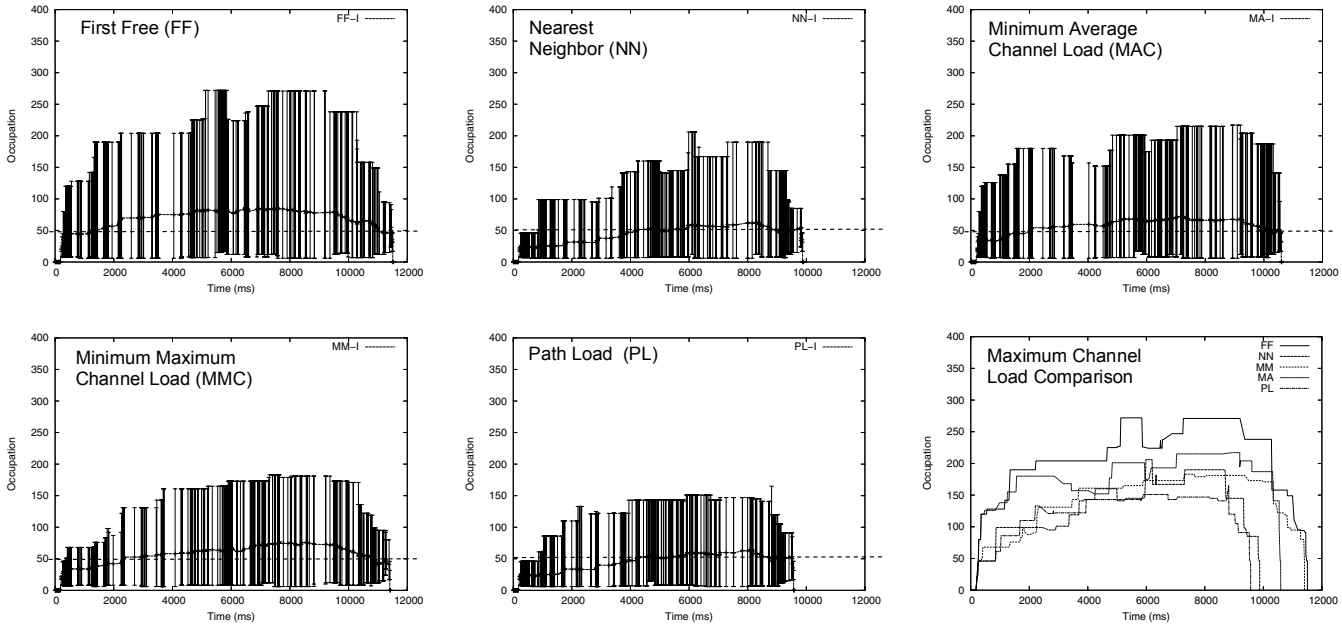

Fig. 7 - Execution time versus link load (occupation) for the dynamic mapping strategies (initial mapping: clustering), for 13 applications. The continuous line in the graphs denotes the average link load.

Fig. 8 shows the results for five applications running simultaneously, for NN and PL mapping heuristics. Both heuristics present similar average link occupation, with an important congestion reduction for the PL mapping heuristic.

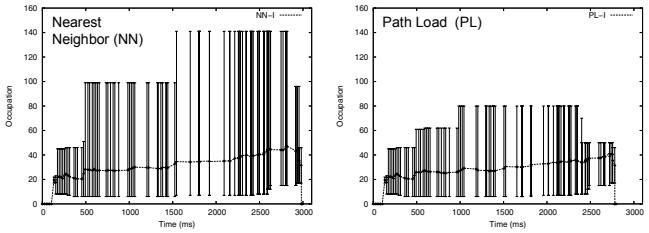

Fig. 8 - Execution time vs. link load for 5 applications.

Tab. 1 presents the execution time to complete the applications, using the mapping strategies. The execution time considers for each task the processing time and the communication time. The allocation time is indirectly considered, since it is a function of the delay to receive the NOTIFY message. In fact, the only variable in the execution time is the communication time. The *Path Load* mapping, compared to the FF, reduces in average 19.3 % the total execution time. Note that the simple NN mapping also reduces the execution time (18.7%). The advantage of the *Path Load* mapping is the congestion reduction.

Tab. 1 - Total Execution Time in seconds.

	Number of Simulated Applications						
	1	***5***	***9***	***13***	***19***	***50***	***99***
FF	0.58	4.67	8.04	11.53	16.60	44.50	89.61
NN	0.46	3.00	6.77	9.90	14.50	37.55	74.95
MAC	0.55	3.76	7.50	10.61	15.59	40.04	80.77
MMC	0.52	3.96	8.01	11.43	16.56	44.10	88.40
PL	0.46	2.80	6.90	9.59	14.30	37.61	76.81

6 Conclusions

The nearest neighbor heurist mapping, adopted in generic task placement, effectively reduces the average link occupation, as showed in the results Section. However, in NoC infrastructures, to reduce execution time and power consumption, congestion in links should be avoid. To reach congestion reduction, the mapping heuristic must include some structural parameters of the NoC. The structural parameters modeled in this work are the NoC topology, the routing algorithm, and the communication delay (as a function of the number of hops between nodes and the saturation in links). The proposed mapping algorithm, named Path Load, minimizes the occupation of the internal links of the NoC, for each new task inserted in the system. The presented graphs, for 5 and 13 applications, demonstrate the effective congestion reduction. Varying the number of simultaneous applications results in the same result profile.

As mentioned before, multi-task processors increase the number of the software tasks available in the system. Future works includes extending the mapping heuristics to multi-task processors. Since multi-task processors have a microkernel to manage tasks execution, such processors may send their load, as well the links loads, to the manager processor. In such a way, the mapping heurist could take decisions based in the actual system load, instead the load furnished by applications and link load estimations. Finally, if the manager processor knows the actual load of processors and links, task migration algorithms can be implemented to ensure QoS to applications.

References

[1] Jerraya, A.; Tenhunen, H.; Wolf, W. Guest Editors' Introduction: Multiprocessor Systems-on-Chips. IEEE Computer, v.38(7), 2005.

[2] Hu, J.; Marculescu, R. Energy- and Performance-Aware Mapping for Regular NoC Architectures. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, v.24(4), 2005.

[3] Marcon, C.; Borin, A.; Susin, A.; Carro, L.; Wagner, F. Time and Energy Efficient Mapping of Embedded Applications onto NoCs. ASP-DAC, 2005.

[4] Nollet, V.; Marescaux, T.; Avasare, P.; Mignolet, J-Y. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. DATE, 2005.

[5] Bertozzi, S.; Acquaviva, A.; Bertozzi, D.; Poggiali, A. Supporting task migration in multi-processor systems-on-chip: a feasibility study. DATE, 2006.

[6] Ahmadinia, A.; Teich, J. Speeding up Online Placement for XILINX FPGAs by Reducing Configuration Overhead. VLSI-SoC, 2003.

[7] Handa, M.; Vemuri, R. An Integrated Online Scheduling and Placement Methodology. FPL, 2004.

[8] Tabero, J.; Septién, J.; Mecha, H.; Mozos, D. A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management. FPL, 2004.

[9] Ruggiero, M.; Guerri, A.; Bertozzi, D.; Poletti, F.; Milano, M. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. DATE, 2006.

[10] Marcon, C.; Calazans, N.; Moraes, F.; Susin, A.; Reis, I.; Hessel, F. Exploring NoC mapping strategies: an energy and timing aware technique. DATE, 2005.

[11] Kalte, H.; Lee, G.; Porrmann, M. Context Saving and Restoring for Multitasking in Reconfigurable Systems. FPL, 2005.