Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles

V. Nollet, T. Marescaux, P. Avasare, D. Verkest, J.-Y. Mignolet

Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles [*]

V. Nollet, T. Marescaux, P. Avasare, D. Verkest[†], J-Y. Mignolet
IMEC, Kapeldreef 75, 3001 Leuven, Belgium
[†]also Professor at Vrije Universiteit Brussel and at Katholieke Universiteit Leuven
{nollet, marescau, avasare}@imec.be

Abstract

Run-time management of both communication and computation resources in a heterogeneous Network-on-Chip (NoC) is a challenging task. First, platform resources need to be assigned in a fast and efficient way. Secondly, the resources might need to be reallocated when platform conditions or user requirements change. We developed a run-time resource management scheme that is able to efficiently manage a NoC containing fine grain reconfigurable hardware tiles. This paper details our task assignment heuristic and two run-time task migration mechanisms that deal with the message consistency problem in a NoC. We show that specific reconfigurable hardware tile support improves performance of the heuristic and that task migration mechanisms need to be tailored to on-chip networks.

1. Introduction

Future platforms will contain a mixture of heterogeneous processing elements (PEs) [1], also denoted as tiles. These programmable tiles will be interconnected by a configurable on-chip communications fabric or a Network-on-Chip (NoC) [2, 3]. Dynamically managing the computation and communication resources of such a platform is a challenging task, especially when the platform contains a special PE type such as *fine-grain reconfigurable hardware* (RH) [1]. Compared to traditional PEs, RH operates in a different way, exhibiting its own distinct set of properties.

This paper details two important components of our run-time resource management scheme: the NoC resource management heuristic and the run-time task migration. An in-depth discussion of other components such as our injection rate control mechanism and our hierarchical configuration mechanism is provided in respectively [12] and [13]. The main contribution of this paper is the description of a NoC resource management heuristic that makes efficient use of reconfigurable hardware tiles and that is closely linked to a run-time task migration mechanism. In addition, our task migration mechanisms are tailored to a NoC environment. It is the first time to our knowledge that the issues of run-time task migration are addressed in a NoC context.

The rest of the paper is organized as follows. Section 2 describes our emulated NoC system. Section 3 details and evaluates our RH enhanced resource management heuristic. Section 4 details our NoC run-time task migration mechanisms. Section 5 presents the conclusions. Related work [3-11] is discussed throughout the paper.

2. System Description

Our multiprocessor NoC is emulated by linking the StrongARM processor of a PDA (further denoted as master PE) to an FPGA containing the slave PEs (S) (Figure 1a). All PEs are interconnected by a 3x3 packet-switched bidirectional mesh NoC clocked at 30 MHz.

The central OS executes on top of the master PE and is responsible for assigning resources (both computation and communication) to the different tasks. The OS keeps track of the computation resources by maintaining a list of PE descriptors. The communication resources are maintained by means of an injection slot table that indicates when a task is allowed to inject messages onto a link of the NoC (Figure 1b). Every tile contains a destination lookup table (DLT) that enables a task to resolve the location of its communication peers. The NoC provides the OS with a set of tools to monitor the resources and to enforce its decisions [12].

In our current NoC configuration, tiles 1 and 5 expose the fine-grain reconfigurable hardware (i.e. FPGA fabric). These tiles are suited for computational intensive tasks, but can only accommodate a single task. The mapping heuristic will have to find the *best fitting* task for every reconfig-

* This work is partly funded by the European Commission (IST-AMDREL project IST-2001-34379) by the Flemish Government (GBOU-RESUME project IWT-020174-RESUME) and by Xilinx Labs, Xilinx Inc. R&D group.

urable hardware tile, since some tasks can be too big for a certain RH tile (i.e. cannot be placed on that tile), while other tasks cause *internal fragmentation* (i.e. waste RH area because the task size is smaller than the tile size).

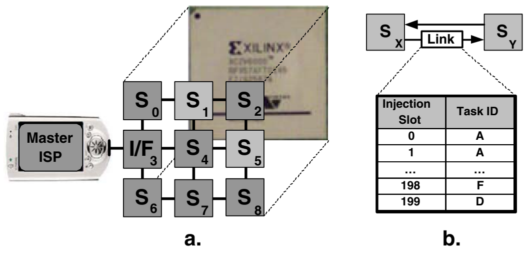

Figure 1. (a) Our NoC emulation platform. (b) Communication resource management.

3. Resource Management Heuristic

The resource management heuristic consists of a basic algorithm completed with reconfigurable add-ons. The basic heuristic contains ideas from multiple resource management algorithms [5, 6, 7]. Here, our contribution lies in compiling these ideas into a suitable run-time management heuristic. In addition, we created a set of RH add-ons that allow the basic heuristic to deal with the specific properties of RH. These add-ons aim to improve the performance of the heuristic and to create extra management opportunities in the presence of RH.

3.1. Basic Heuristic

In order to assign resources to an application containing multiple communicating tasks, the heuristic requires the application specification, the user requirements and the current resource usage of the platform as input. The application is specified by means of a task graph that contains the properties of the different tasks (e.g. support for different PE types) and the specification of the inter-task communication. The user requirements are specified by means of a simple in-house QoS specification language (similar to the ones specified by [4]) The different steps to come to a complete resource assignment of an application are as follows.

1. **Calculating requested resource load.** Based on the task load specification function $f_i(x, y, \ldots)$ provided by the application designer and the user requirements (specifying x, y, $\ldots$), the heuristic calculates the real computation and communication task load. In case of a video decoding task, for example, the framerate, resolution and decoding quality requested by the user will affect both the computation and communication resource requirements of the task.
2. **Calculate task execution variance.** For every task T_i in the application, determine its execution time variance on the different supported PE types and normalize that value by the number of evaluated PE types (V_{Ni}). Tasks with a high V_{Ni} are very sensitive to which processing element they are assigned to. In addition, tasks that can only be mapped on one specific PE should be mapped before all other tasks. This way, the heuristic avoids a mapping failure, that would occur if this specific PE would be occupied by another task.
3. **Calculate task communication weight.** For every task T_i in the application, determine its communication importance C_i (both incoming and outgoing) with respect to the total inter-task communication of the application. This allows the algorithm to order the tasks based on their communication requirements.
4. **Sort tasks according to mapping importance.** The mapping priority of a task T_i is equal to V_{Ni} x C_i. Tasks are sorted by descending priority.
5. **Sort PEs for most important unmapped task.** This step contains two phases. First, the allocation priority of the PEs for a task T_i is determined based on the weighted product of the current PE load and the already used communication resources to the neighboring PEs. The weights are determined by the computation and communication requirements of the unmapped task. This allows the algorithm to match tasks that combine a high need for processing power and a low need for communication resources with their counterparts. Secondly, in order to map heavily communicating tasks close together, the allocation priority is also multiplied with the hop-bandwidth product (i.e. the product of the amount of assigned communication injection slots between two tasks and the hop-distance between them) of the current task and its already placed communication peers. PEs that lack the required computation resources (phase 1) or that do not provide enough communication resources to the already placed tasks (phase 2) have their allocation priority set to infinity, indicating that the PE is not fit to accommodate the unmapped task.
6. **Mapping the task to the best computing resource.** The most important unmapped task is assigned to the best fitting PE. Consequently, the platform resource usage is updated to reflect this assignment. Steps 5 and 6 are repeated until all tasks are mapped.

Occasionally this greedy heuristic is unable to find a suitable mapping for a certain task. This usually occurs when mapping a resource-hungry application on a heavily loaded platform. *Backtracking* is the classic solution for this issue: it changes one or more previous task assignments in order to solve the mapping problem of the current task.

The backtracking algorithm starts by undoing N (start by N equals one) previous task resource allocations. Then, the PEs are sorted, but instead of choosing the best PE for a certain task, the second best PE is selected. If this does not solve the assignment issue for the current task, backtracking is repeated with N+1. Backtracking stops when either the number of allowed backtracking steps is exhausted or when backtracking reached the first task assignment of the application. In that case, the algorithm can (a) use run-time task migration (Section 4) to relocate a task of another application in order to free some resources, (b) use hierarchical configuration (Section 3.2) or (c) restart the heuristic with reduced user requirements.

3.2. Reconfigurable Hardware Add-ons

Incorporating RH tiles requires some additions to the basic mapping heuristic in order to take reconfigurable hardware properties into account.

The first set of additions are applied after step 5 of the basic mapping heuristic (i.e. after sorting all suitable PEs). There are two distinct RH properties that are taken into account. First, the *internal fragmentation of reconfigurable area* is considered. In case both the first and second priority tile are RH tiles. The heuristic will re-evaluate their priority using a fragmentation ratio in order to minimize the area fragmentation. Intuitively it is easy to understand that if placing the task on the highest priority tile causes 80% area fragmentation while the second priority tile only causes 5% area fragmentation, it might be better to place the task on the latter. Secondly, the *binary state* (i.e. either 0% load or 100% load) and the *computational performance* of RH tiles are considered. Due to the attempt at load-sharing of the heuristic algorithm RH tiles are often selected as best mapping candidates. Obviously, it would not be wise to sacrifice a RH tile when a regular PE could do a similar job. Therefore, if the highest priority tile for a certain task is a RH tile, while the second priority tile is a regular PE, the heuristic will use a load ratio to re-evaluate their priority to avoid wasting RH computing power.

The second set of additions involves hierarchical configuration [13], i.e. the use of softcore PEs instantiated on RH tiles. There are two situations where this technique can improve mapping performance. First, when the task binaries are not supported by the platform PEs, a suitable softcore can be instantiated on a RH tile. This means the heuristic first needs to determine where to instantiate the softcore This is done by going over all softcores that are (1) supported by the task, (2) that fit on the available (i.e. free) RH tiles and (3) provide the required computing power. After finding a suitable location, the softcore is instantiated. From that moment on, the regular heuristic algorithm applies. Secondly, this technique can be used as an alternative to backtracking. Consider the mapping example (Figure 2a), where task T_B still needs to be assigned. Since T_B has no RH support (tile 1) and all other tiles are occupied or unsupported, T_B can only be assigned to tile 8. Although tile 8 can provide the required computing resources, it lacks the required communication resources to support the communication between T_B and T_C. Without hierarchical configuration, the heuristic has no other option but to reconsider the mapping of T_A and T_C (i.e. perform backtracking). In case T_A and T_C are only supported on respectively tile 0 and tile 2, the heuristic will even need to reallocate resources of other applications (e.g. moving tasks from tile 4 to tile 8) in order to free resources. However, by means of hierarchical configuration, T_B can be mapped on a softcore instantiated on RH tile 1 (Figure 2b). Also from a hop-bandwidth point of view (i.e. mapping quality), it is better to map T_B on a softcore on RH tile 1 than on tile 8.

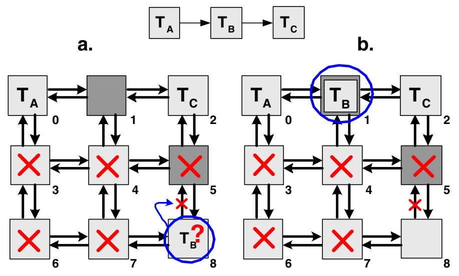

Figure 2. Hierarchical configuration example.

3.3. Heuristic Performance Evaluation

The performance of the heuristic was assessed by comparing it to an algorithm that explores the full solution space. The performance experiments consist of mapping a typical test application (Figure 3) on our 3x3 NoC containing four PE types.

In order to include the current load of the platform and the user requirements into the mapping decision process, three types of load have been defined: LIGHT, MEDIUM and HEAVY. In case of platform load, they indicate that no platform resource (both computation and communication) is used for more than respectively 25%, 50% and 75%. A random function determines the actual resource usage for every resource. If the random function returns 50% or more usage on a single-task tile (e.g. RH tile), then this tile is considered as used (i.e.100% usage). Otherwise, it is con-

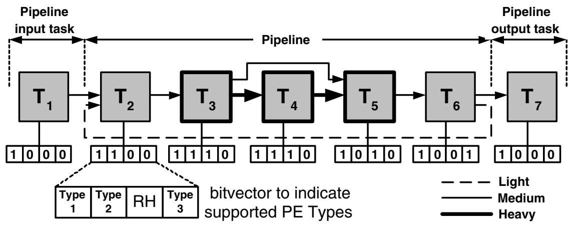

Figure 3. Example application containing a multimedia pipeline (e.g. video decoding).

sidered as free. In case of user requirements, these loads indicate that no task of the application uses more than respectively 25%, 50% and 75% of a certain resource. Placing a task on a single-task tile will result in 100% usage.

Table 1 illustrates the success rate of the heuristic (with respect to searching the full mapping solution space) for LIGHT and MEDIUM loaded platforms and for varying application load. The amount of backtracking steps allowed is indicated by the *BT* value. On the StrongARM SA-1110 PE (206 MHz), the heuristic requires on average 893 μs (std. dev. 77 μs) to reach a full mapping without backtracking. With backtracking (BT=3), the algorithm requires on average 1.13ms (std. dev. 358 μs) to come to a conclusion (i.e. success or failure). Exploring the entire solution space requires about 378 ms. The experiment shows that, although backtracking clearly improves the success rate, the heuristic does not always find a suitable solution.

NoC 3x3 LIGHT				
Application	**BT=0**	**BT=1**	**BT=2**	**BT=3**
LIGHT	100%	100%	100%	100%
MEDIUM	100%	100%	100%	100%
HEAVY	78.5%	88.0%	90.0%	94.5%
NoC 3x3 MEDIUM				
LIGHT	100%	100%	100%	100%
MEDIUM	97.0%	99.0%	99.5%	100%
HEAVY	53.67%	61.00%	67.79%	71.75%

Table 1. Mapping success for varying number of backtracking steps (BT).

In the experiments leading to the results of Table 1, all tasks with RH support (i.e. T_3, T_4 and T_5) could be placed on any of the two RH tiles. However, when T_4 and T_5 only fit on tile 5, while T_3 fits on both RH tiles, the mapping success rate drops from 53.67% to 44.73% in case of a MEDIUM loaded 3x3 NoC (application HEAVY, without backtracking). The mapping success drops even further down to 36.84% in the absence of the reconfigurable hardware add-ons concerned with area fragmentation and gain. This means the RH add-ons significantly improve the mapping performance in case of different RH tile sizes.

By looking at the hop-bandwidth product (i.e. the product of the number of assigned injection slots between two tasks and the hop-distance between them), it is possible to estimate the mapping quality. Indeed, heavily communicating tasks should be mapped close together in order to minimize communication interference [12]. Table 2 shows that the heuristic algorithm performs well under various load conditions. The main reason for the very low minimum hop-bandwidth product of application LIGHT is that all tasks with heavy communication can be placed on a single tile. However, the heuristic tries to spread the communication and computation load among the PEs.

NoC 3x3 MEDIUM - HopBandwidth				
Application	**Heuristic**	**Max**	**Min**	**Mean**
LIGHT	315	825	75	466
MEDIUM	570	1530	420	936
HEAVY	945	1890	720	1275

Table 2. Hop-bandwidth mapping quality.

In contrast to the related work [5, 7], our heuristic does not consider the co-scheduling issue nor the real-time constraints of individual tasks, because currently our slave PEs (e.g. RH tiles) can only accommodate a single task (i.e. no co-scheduling or real-time issue on these PEs).

4. Run-Time Task Migration

Whenever the user requirements change (e.g. switching to another resolution in a video application) or in case of a mapping failure, the resource management heuristic can use run-time task migration to re-allocate resources. Run-time task migration can be defined as relocation of an executing task from the *source tile* to the *destination tile*. In order to overcome the architectural differences between heterogeneous PEs, tasks can only migrate at pre-defined execution points in the code (further denoted as migration points) [10]. How to handle task state when migrating between a regular PE and a RH tile is detailed in [14]. Another major issue in run-time task migration is assuring communication consistency during the migration process. This issue originates from the fact that, after receiving a migration request, the amount of time and input messages a task requires to reach its migration point is unknown. This means that the message producer tasks (i.e. the communication peers) have to keep sending messages until the migrating task signals that a migration point is reached and that it stopped consuming messages. However, at that time there might some unprocessed messages buffered in the communication path between message producer tasks and the migrating task.

The run-time task migration topic has been studied extensively for multicomputer systems since the beginning of the 1980s. However, due to the very specific NoC properties (e.g. a very limited amount of communication memory), the existing mechanisms are not directly applicable.

The message consistency mechanism described by Russ et al. [8] collects all unprocessed messages into a special input queue when a migration point is reached. After the actual migration, all communication peers are notified and their task lookup table is updated to reflect the new location of the migrated task. Communication consistency is preserved by emptying the special input message queue before receiving any messages produced after completion of the migration process. This mechanism is not well-suited for a NoC: due to the very limited amount of message buffer space it is impossible to store all incoming messages after a task reached its migration point. Adding more buffer space is expensive and the maximum amount of required storage is very application dependent.

The message consistency mechanism of the Amoeba OS [9] drops the unprocessed messages (instead of queuing them) during task migration. The message producer is responsible for resending the message. After migration, any task that sends a message to the old location of the migrated task will receive a *not here* reply. This response triggers a mechanism to update the producer's task lookup table. A drawback of this technique is the loss of migration transparency (i.e. messages need to be resent to a new destination). In addition, dropping and re-transmitting packets reduces network performance, increases power dissipation [3] and leads to out-of-order message delivery. Getting messages back in-order in a task-transparent way requires (costly) additional re-order functionality and buffer space.

4.1. NoC Task Migration Mechanisms

This section details two task migration mechanisms that ensure message consistency in a task-transparent way, assuming a very limited amount of message buffer space per tile. In addition, both mechanisms ensure in-order message delivery without requiring message re-order functionality.

The first mechanism, further denoted as general mechanism is illustrated by Figure 4. When the OS issues a migration request (1), the task running on the source tile may require more input data from the producer tasks before it can reach a migration point. Neither the OS, nor the producer tasks know how many input messages are still required. When the task on the source tile reaches a migration point, it signals this event to the OS (1 to 2). In turn, the OS instructs the producers to send one last tagged message and

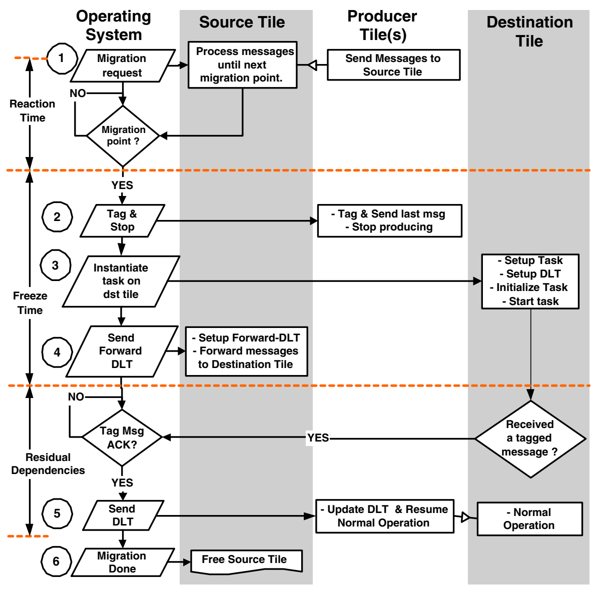

Figure 4. General task migration mechanism.

then to stop sending (2). The OS consequently sets up, initializes and starts the migrating task on the destination tile (3). The next step is to forward all buffered and unprocessed messages to the new location of the migrated task. To this end, the OS initializes a so-called forward-DLT (containing the new destination for the messages) on the source tile and instructs it to orderly forward all incoming messages (4). The destination tile informs the OS whenever it receives a tagged message. Consequently, the OS updates the DLT of the respective message producer tile to reflect the new location of the migrated task before instructing the producer to resume sending messages (5). When all tagged messages have been received, the migration process is finished and the OS can free the resources of origin tile (6).

The second migration mechanism, further denoted as pipeline mechanism, is based on the assumption that many algorithms are pipelined (e.g. video decompression pipeline) and that they contain *stateless points*. Meaning that at certain moments, new and independent information is put into the pipeline. For example, an MPEG decoding pipeline periodically decodes an I-frame. This I-frame does not depend, in any way, on previously processed information. Hence, an I-frame could be decoded by a newly instantiated MPEG decoding pipeline. This assumption allows a migration mechanism to move multiple pipelined tasks at once without being concerned about transferring task state. This is useful when new QoS requirements affect a set of tasks within the application.

In this mechanism, the OS instructs the pipeline input task (Figure 3) to continue feeding data into the pipeline until a stateless point is reached. Then, the pipeline input task should flush the pipeline by sending a tagged message through the pipeline or by informing the pipeline output task. As soon as the pipeline is flushed, the pipeline output task notifies the OS. In contrast to the general mechanism, there are no unprocessed or buffered messages in the path between pipeline input and pipeline output. So the OS can re-instantiate every task of the pipeline with its respective DLT in a different location. After updating the DLT of the pipeline input task, normal operation can be resumed and the OS can free the resources of the original pipeline.

4.2. Migration Mechanism Benchmarking

The migration mechanism performance can be assessed by some typical benchmark parameters [11]. The *reaction time* (i.e. time between migration request and the task actually reaching the migration point) of the pipeline mechanism will depend on the time required to reach a stateless pipeline migration point and the time required to empty the pipeline. For the general mechanism, the reaction time for migrating a single task is dependent on the amount of migration points implemented within that task.

The *freeze time* (i.e. the time during which a task is suspended) depends on the time required to set up DLT's ($t_{DLT} \approx 90\mu s$), issue OS commands ($t_{CMD} \approx 60\mu s$), etc. Consider a pipeline containing T tasks, C inter-task communication channels and S pipeline input tasks. It can be shown that the freeze time of the general mechanism (not considering task state extraction and initialization) is higher than that of the pipeline mechanism. The difference is:

$$\Delta \text{ freeze time} = (T - S).t_{DLT} + (T + C - S).t_{CMD}$$

If the required PE resources are available upfront, setting up the new pipeline could be performed during the reaction time. In that case the freeze time would be independent of the amount of migrating pipeline tasks.

Once a migrated task has started executing on its new tile, it should no longer depend in any way on its previous tile. This is denoted as *residual dependencies*. These so-called *residual dependencies* are undesirable because they waste both communication and computing resources. The pipeline mechanism has no residual dependencies. The residual dependencies of the general mechanism (Figure 4) are caused by acknowledging the arrival of tagged messages and updating the producer DLT(s) before instructing every producer to resume sending messages. The time required to forward the unprocessed messages heavily depends on the NoC conditions (e.g. congestion, etc.).

In short, the pipeline mechanism is useful when simultaneously moving a set of tasks (e.g. due to changed user requirements). Otherwise, when moving a single task in order to e.g. resolve a mapping issue, the general mechanism is more appropriate (due to the prolonged reaction time of the pipeline mechanism). Both mechanisms require the application designer to explicitly introduce migration points.

5. Conclusion

This paper details two components of our overall resource management scheme for NoCs containing reconfigurable hardware tiles: the run-time resource assignment heuristic and the task migration mechanisms. We show that although the basic heuristic produces good results, specific support for the reconfigurable hardware tiles improves its performance. In addition to showing that the classic task migration mechanisms are not suitable for an NoC environment, we detail two task migration mechanisms tailored for our NoC emulation platform (i.e. with limited communication memory and to a simple communication protocol).

References

[1] R. Tessier, W. Burleson,"Reconfigurable Computing for Digital Signal Processing: A Survey", VLSI Signal Processing 28, p7-27, 2001.

[2] L. Benini, G. DeMicheli, "Networks on Chips: A new SOC paradigm?", IEEE Computer magazine, January 2002

[3] William J. Dally, Brian Towles, "Route packets, not wires: on-chip interconnection networks," DAC 2001, p684-689.

[4] J. Jin, K. Nahrstedt, "Classification and Comparison of QoS Specification Languages for Distributed Multimedia Applications", University of Illinois at Urbana-Champaign, 2002.

[5] Y. Wiseman, D. Feitelson, "Paired Gang Scheduling", IEEE Trans. Par. and Distr. Systems, pp 581-592, June 2003.

[6] Jong-Kook Kim et al., "Dynamic Mapping in a Heterogeneous Environment with Tasks Having Priorities and Multiple Dealines.", Proc. 17th International Parallel and Distributed Processing Symposium, France, 2003.

[7] J. Hu, R. Marculescu, "Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints", DATE 2004, pp234-239.

[8] S. H. Russ, J. Robinson, M. Gleeson, J. Figueroa, "Dynamic Communication Mechanism Switching in Hector", Mississippi State University, September 1997.

[9] C. Steketee, W. Zhu, P. Moseley, "Implementation of Process Migration in Amoeba.", Proc. of the 14th Conference on Distributed Computing Systems, pp194-201, 1994.

[10] P. Smith, N. Hutchinson, "Heterogeneous Process Migration: The Tui System", Univ. of British Columbia, 1996.

[11] Pradeep K. Sinha, "Distributed operating systems: concepts and design", 1997, ISBN 0-7803-1119-1.

[12] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde, "Operating System controlled Network-on-Chip", Proc. of Design Automation Conference, pp.256-259, 2004.

[13] V. Nollet et al.,"Hierarchical run-time reconfiguration managed by an operating system for reconfigurable systems", Proc. ERSA 2003, pp.81-87.

[14] J-Y. Mignolet et al., "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip", Proc. DATE 2003, pp.986-992.