Distributed Resource Management in NoC-Based MPSoCs with Dynamic Cluster Sizes

Guilherme Castilhos, Marcelo Mandelli, Guilherme Madalozzo and Fernando Moraes
FACIN – PUCRS – Av. Ipiranga 6681– Porto Alegre – RS – Brazil
{guilherme.castilhos, marcelo.mandelli, guilherme.madalozzo }@acad.pucrs.br, fernando.moraes@pucrs.br

ABSTRACT
Scalability is an important issue in large MPSoCs. MPSoCs may execute several applications in parallel, with dynamic workload, and tight QoS constraints. Thus, the MPSoC management must be distributed to cope with such constraints. This paper presents a distributed resource management in NoC-Based MPSoC, using a clustering method, enabling the modification of the cluster size at runtime. This work addresses the following distributed techniques: task mapping, monitoring and task migration. Results show an important reduction in the total execution time of applications, reduced number of hops between tasks (smaller communication energy), and a reclustering method through monitoring and task migration.

Keywords
MPSoC, NoC, Distributed Management, Mapping.

1. INTRODUCTION AND RELATED WORK
MPSoCs are able to execute several applications in parallel, supporting dynamic workload, i.e., applications may start at any moment. Another important feature is QoS (quality of service), because multimedia and telecom applications have tight performance requirements that must be respected by the system.

The background enabling several simultaneous applications, with QoS constraints executing on the MPSoC is the system management. System management may include monitoring, task mapping, task migration, NoC control, DVFS. The monitoring is responsible to detect deadline violations, and may be applied at the processor and/or at the NoC level. According to the violation severity, the system management selects a strategy to restore the application performance (diagnosis), as task migration, switching method, DVFS. Once selected the adaptation strategy, the system management execute it. This characterizes a closed-loop control: monitoring, diagnosis, and action.

The system management may be centralized or distributed. Centralized management is suited for small MPSoCs due to scalability reasons. A central manager may be overloaded very quickly, due to the execution of mapping actions and treatment of monitoring events, for example. Also, the traffic around the central manager induces a hot-spot, compromising reliability in long term

An alternative is the distributed management [1][2]. Two main approaches are discussed in the literature: one manager per application, and one manager per MPSoC region. The second approach is preferable, since the number of management resources remains constant, regardless the number of applications executing in the MPSoC. The regions are defined as clusters. All application tasks are executed inside the cluster, if possible, favoring composabilitity.

Distributed management can guarantee gains of performance, fault tolerance and scalability [3]. Table 1 compares our proposal to state-of-art. The present work has as main originality the deployment of a full set of heuristics, enabling to adaptively control the MPSoC execution. The proposed work adopts distributed management, with dynamic task mapping, and task migration. The MPSoC is modeled at RTL level (SystemC), enabling accurate evaluation of performance figures.

The *goal* of the present paper is to present a distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. At system start-up each cluster has a fixed size, and during runtime clusters may borrow resources from neighbor clusters to map applications.

2. ARCHITECTURAL ASSUMPTIONS
Applications are assumed to be represented using task graphs, $A=<T,C>$, where $T = \{t1, t2, ..., tm\}$ is the set of application tasks corresponding to the graph vertices, and $C= \{(t_i, t_j, w_{ij}) \mid (t_i, t_j) \in T$ and $w_{ij} \in \mathbb{N}^*\}$ denotes the communications between tasks, corresponding to the graph edges. The communication between tasks occurs through message passing.

The present work adopts a homogeneous MPSoC architecture, interconnecting PEs through a 2D-mesh NoC. Each PE contains a MIPS-like processor, a network interface, a DMA module, and a private memory for code and data. An external memory, named *task repository*, contains all applications tasks (set T), which are loaded into the system at runtime, using a dynamic task-mapping heuristics [10].

Table 1. State-of-the art in adaptive techniques to control MPSoC, compared to the proposed work.

Ref.	Type of management	Type of distributed management	Monitoring	Task migration	Migration complete task	Dynamic mapping	Clusters	Goal of the work
[1]	Distributed	Application	No	No		No	Yes	Workload Balancing
[3]	Distributed	Application	No	No		No	No	Scalability
[4]	Distributed	Application	No	No		No	No	Scalability
[5]	Distributed	Cluster	No	No		Yes	Yes	Detection of HW faults / Energy consumption
[6]	Centralized		Yes	No		No	No	Deadline control
[7]	Centralized		Yes	No		No	No	Resize depth buffer
[8]	Centralized		No	Yes (master PE)	No (code/data)	No	No	Energy consumption
[9]	Centralized		No	Yes (slave PE)	No (code)	No	No	Improve system performance
This work	Distributed	Cluster	Yes, manager PE	Yes (local manager PE)	Yes (code/data/ context)	Yes (sharing resources)	Yes	Scalability / Balancing workload / Energy consumption / Reclustering

The MPSoC is divided in *n* equally sized clusters, as illustrated in Figure 1. PEs may act as: Global Master (GMP), Local Master (LMP) and Slave (SP). SPs are responsible for task execution. Each SP runs a simple operating system, which enables the communication between PEs and multitask execution. Each SP may execute *k* simultaneous tasks. Therefore, each cluster may execute *k**|SP| tasks simultaneously, corresponding to the number of resources that the cluster has.

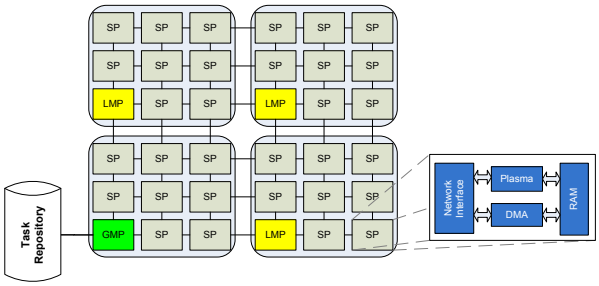

Figure 1 – Clustered architecture for a homogeneous MPSoC.

The LMP is responsible to control the cluster, executing functions such as monitoring, task mapping, deadlines verifications and communication with other LMPs and the GMP. The GMP has all functions of the LMP, and functions related to the overall system management, such as: select in which cluster a given application will be mapped, control the available resources in each cluster, receive debugging and control messages from LMPs. The GMP is the only PE having access to external memory (*task repository*).

3. DISTRIBUTED RESOURCE MANAGEMENT

The distributed resource management assumes an MPSoC divided in *n* regions, named clusters. At system startup, all clusters have the same size. At execution time, if an application does not fit in a given cluster, the LMP of the cluster may request resources to neighbor clusters. The LMP of the cluster monitors the resource availability, migrating tasks that should be in the cluster back to the cluster. Therefore, the cluster size varies dynamically at runtime.

The next sections describe the three main distributed mechanisms: mapping, monitoring, and task migration.

3.1 Distributed Mapping

Figure 2 presents the mapping of a new application in the MPSoC. According to user requests, new applications can be loaded at runtime. This action is represented in the Figure by arrow "*new application*".

The required steps to map a new application include:

1. When the GMP receives an application request, it executes the heuristic "*cluster selection*", which chooses the cluster that can receive the application. The selected cluster is the one with resources to execute the application. If there are no clusters with enough resources, it is chosen the one with the smallest difference between the number of applications tasks and available resource.
2. The GMP reads from the application repository the *application description* (set T), transmitting it to the selected LMP.
3. The LMP maps the initial tasks, i.e., those without dependences to other tasks. This heuristic searches for the SP with the highest number of available resources around it. This increases the probability of the remaining tasks of the application to be mapped close to each other, reducing communicating distance between tasks, and therefore the communication energy.
4. The LMP sends the message "*task allocation request*" to the GMP with the identification of the task to be mapped, its address in the repository, and the address of the SP that will receive the task.
5. The GMP configures its DMA module to transmit the task code to the selected SP. The use of the DMA ensures that the task is transmitted as a burst, reserving the NoC resources for a small amount of time.
6. The task code is transmitted through the NoC, received and stored at the selected SP. The SP will schedule the new task at the end of the task allocation packet reception. In addition, the SP keeps a data structure, named task table, with the address the tasks assigned to it.
7. When the initial task executes a communication with a task that it is not in its task table, a *"task allocation request"* is transmitted to its LMP.
8. The LMP executes the mapping heuristic [10], and steps 9-10-11 are similar to steps 4-5-6.

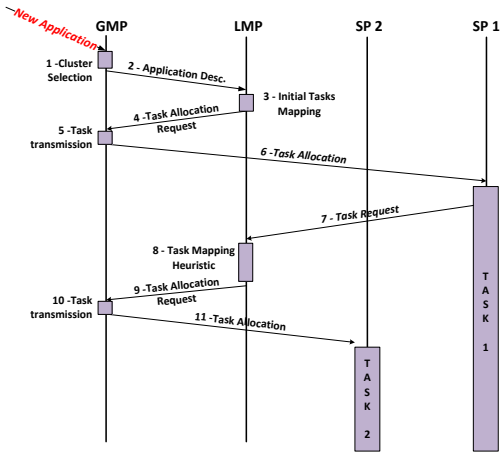

Figure 2 – Protocol to insert new applications into the system.

The step 8 of Figure 2 can fail if the cluster has no available resources. The example in Figure 3 assumes the top-left cluster is requiring a task mapping, and there are no available resources in cluster. In this case, the LMP of the cluster sends a "*loan request*" message, requesting resources to all neighbor clusters LMPs (step 1 of Figure 3).

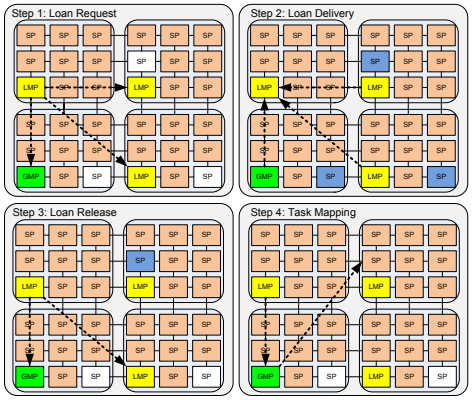

Figure 3 – Protocol to task mapping in neighbor cluster. White SPs (slave PEs) are available PEs.

Then the LMPs that received the "*loan request*" search for available resources in theirs cluster. If there is only one available resource, this resource is reserved to be borrowed; otherwise, if there is more than one available resource, the LMP will reserve the one as close as possible, in number of hops, between the task to be mapped and the source task in the cluster. After the reservation, all neighbor LMPs send a "*loan delivery*" message to the LMP that requested resources, notifying the resource position (step 2 of Figure 3, blue SPs are reserved), if it exist.

The LMP chooses the closest resource from the one that requested the task, sending a "*loan release*" message to all LMPs which were not selected (step 3 of Figure 3). Next, the LMP send a "*task allocation request*" message to the GMP requesting the task mapping on the borrowed resource (step 4 of Figure 3). Therefore, the cluster size increases at runtime, because the borrowed resource is now part of this cluster. This process optimizes the system management, since applications can be mapped in clusters, even if the cluster has no sufficient resources.

3.2 Monitoring

As described in the previous section, a cluster can borrow SPs from neighbor clusters. This increases the hop number between tasks, with the following drawbacks: (*i*) performance degradation of the application, due to its fragmentation; (*ii*) increased data traffic volume in the NoC, (iii) increased communication energy, since it is proportional to the number of traversed hops.

When a given task finishes its execution, its reserved resource is released. Two cases may arise: end of a task mapped in its cluster, or end of a task mapped in a neighbor cluster. In the former case, a message is sent to its LMP notifying the end of task. In the second case, two messages are sent, one to its LMP, and a second one to the cluster where the task is mapped.

Each LMP monitors the resource usage of its SPs. The monitoring is implemented at the task level. When an "*end of task*" is received by a LMP, it verifies if there are tasks that should be mapped locally are mapped in other clusters. If this condition is true, this means that it is possible to optimize the application performance by migrating the task into the cluster. It is verified the Manhattan distance between the task to be migrated with its communicating tasks. If the Manhattan distance decreases, the LMP sends a migration message to the SP holding the task.

The LMPs also monitor the end of applications. When a given application finishes, this action is transmitted to the GMP. The GMP then releases all resources reserved to the application, enabling their use by a new application.

3.3 Task Migration

In [11] a task migration heuristic is detailed, with the following features: (*i*) it is not necessary to have migration checkpoints, i.e. tasks may be migrated at any moment; (*ii*) complete task migration, including context, code and data; (*iii*) in-order message delivery, i.e. tasks communicating with migrated tasks will receive the messages in order they were created.

An important feature of the task migration is the message delivery control. The process to ensure the correct in-order message consumption is to locally store in the operating system the produced messages. When the task is migrated, the produced message remains in the original PE. Therefore, the tasks that communicate with the migrated task still use the task address before migration. Once all messages are consumed, the communication requests are forwarded to the new position and the new position forward the message with the new task address.

4. RESULTS

Results were obtained using three benchmarks: *MPEG*; *multispec* image analysis, evaluate the similarity between two images using different frequencies; *synthetic*. The experiments use the HeMPS MPSoC [12], described in RTL cycle accurate modeling (SystemC). Relevant features of the MPSoC include: 32-bit PE word and 16-bit flit; page size with 16 Kbytes (4,096 works); time-slice: 16,384 clock cycles (amount of time each task is scheduled); NoC router: wormhole packet switching, XY routing, centralized round-robin arbitration.

4.1 Total Execution Time

Table 2 presents the execution time normalized w.r.t the centralized management in a 12x12 MPSoC, with an MPSoC load equal to 75%. As can be observed, the distributed management leads to a total execution time reduction. The smaller reduction observed in the MPEG benchmark is due to its periodicity feature. The reduction in the total execution time reduction comes from: (i) several PEs execute the task mapping in parallel; (ii) each manager treats a smaller number of control packets (mapping request, end of task, monitoring events) compared to the centralized approach. This reduces the manager load and the traffic in the NoC.

A relevant issue is the cluster size. Is there an optimal cluster size? Even with few clusters, as two, the total execution time is reduced, due to the parallelism of the management tasks. A large number of clusters use too much MPSoC resources, reducing the number of applications that can run simultaneously. From Table 2, a cluster size with 18 (6x3) or 16 (4x4) PEs represents a good trade-off between execution time reduction and resources reserved for management.

Table 2. Total execution time normalized w.r.t. the centralized management (cluster sized 12x12), in a 12x12 MPSoC instance, with an MPSoC load equal to 75%.

Cluster Size	Nb of Clusters	Benchmark		
		MPEG	Synthetic	Multispec
12x12	1	1,00	1,00	1,00
12x6	2	0,94	0,78	0,77
6x6	4	0,90	0,67	0,63
6x4	6	0,88	0,58	0,71
6x3	**8**	**0,86**	**0,57**	**0,56**
4x4	**9**	**0,88**	**0,58**	**0,52**
3x3	16	0,87	0,54	0,49

In a smaller MPSoC the number of simultaneous tasks is smaller compared to a lager MPSoC. Therefore, the management load is smaller. In such a case, the partitioning of a smaller MPSoC is less effective than a large MPSoC. This point out to a second issue: when a centralized or distributed management should be used? From the Authors results, MPSoCs starting with 64 PEs, with four 4x4 clusters, presents significant reduction in the execution time, as observed in Table 2.

Summarizing the evaluation of the total execution: (*i*) distributed management is effective for large MPSoCs, i.e., more than 64 PEs; (*ii*) a cluster size with 16-18 PEs represents a good trade-off between total execution time and PEs dedicated to management; (*iii*) the distributed management reduced the augmentation of the total execution time when increasing the system load.

4.2 Hop Number

The evaluation of the average hop number is a key parameter to evaluate the mapping quality. A small hop number between tasks favors QoS, since communication tasks are mapped closer to each other, and reduces the communication energy. Higher values of hop number on the other side penalize the performance of

applications, since disturbing traffic may interfere in the communication.

Table 3 presents statistical data related to the hop number between tasks, for two benchmarks and three cluster sizes. In both benchmarks the centralized management (cluster size 12x12) presented a higher average hop number than the distributed management. The mapping in the centralized management potentially could generate better solutions than the distributed management, because the mapper has a global view of the MPSoC. This is true when the load applied to the MPSoC is small. In such cases, both management techniques present similar results in terms of hop number. In scenarios with a higher load (Table 3), the number of continuous regions using centralized management reduces, increasing in this way the hop number. On the other side, the use of clusters favors the usage of the MPSoC resources, keeping continuous regions even under higher loads applied to the MPSoC. The results presented in Table 3, are also related to the total execution time (previous section). The higher is the average number of hops, higher is the total execution time.

The standard deviation observed in the centralized management is high (2.17 and 7.98), meaning that some applications have their tasks mapped far from each other. The distributed management presents a smaller standard deviation, from 0.31 to 1.65. This means that most applications were mapped in a continuous region, inside the cluster, favoring composability, an important feature for QoS.

Table 3. Hop number between tasks, in a 12x12 MPSoC, load equal to 75%.

Cluster Size	Nb of Clusters	MPEG (5 tasks)			
		min	avg	max	Std dev
12x12	1	4	5.14	14	2.17
6x3	8	4	4.05	6	0.31
4x4	9	4	4.45	8	1.15
Cluster Size	Nb of Clusters	Synthetic (6 tasks)			
		min	avg	max	Std dev
12x12	1	2	9.66	38	7.98
6x3	8	2	5.65	10	1.51
4x4	9	2	6.00	10	1.65

4.3 Reclustering

This Section evaluates monitoring with task migration. Two scenarios were evaluated: (*i*) *main application* - MA_{pp} (evaluated application) with disturbing applications, without task migration; (ii) MA_{pp} with disturbing applications and two task migrations (tasks E and D).

Figure 4(a) shows the initial mapping. Tasks D and E of the MA_{pp} were mapped in neighbor clusters, not in the cluster that contains the manager PE of the MA_{pp} (top-left cluster). These two tasks were mapped in neighbor clusters due to absence of resources in the cluster managing MA_{pp}. When tasks belonging to the disturbing application finish, the manager of the cluster verify if exists tasks running in other clusters. In this case, the LMP sends migration requests to SPs running these tasks. In Figure 4(b) tasks D and E were migrated to cluster managing MA_{pp}. This results in a smaller number of hops between the tasks of MA_{pp}, with an improvement in its execution performance.

The MA_{pp} is periodic, with a pipeline behavior, repeating each task for parameterizable number of iterations. The iteration time of task F stabilizes in 10,000 clock cycles. With task migration, as expected, during migration the iteration time of task F increases, because the application is stalled since the data is not consumed by tasks D and E. After migration, the iteration time of task F stabilizes in 8,600 clock cycles. Such improvement came from the hop number reduction between MA_{pp} tasks.

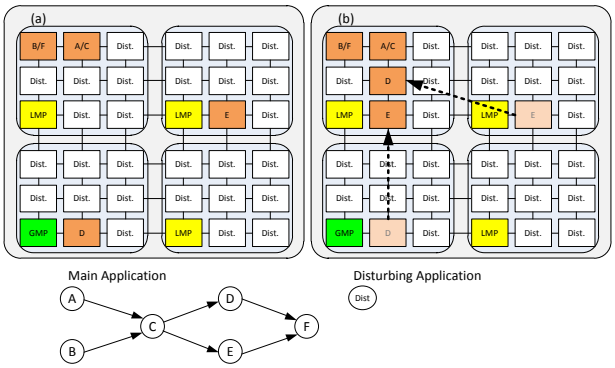

Figure 4 – Experimental setup to task migration.

The total execution time for both scenarios was 2,772,692 and 2,698,812, in clock cycles. This corresponds to a reduction of 2.67% in the total execution time, considering two task migrations. Therefore, even if task migration momently increases the execution time, the final result is an improvement in the overall performance.

5. CONCLUSIONS AND FUTURE WORKS

This work proposed a distributed resource management for NoC-based MPSoCs with dynamic cluster sizes. The proposal divides the system in fixed sized cluster at startup that may be resized according to a reclustering protocol. Compared to a centralized system, the proposed management technique reduced the distance among tasks, resulting in an important reduction in the total execution time. In addition, it was shown that monitoring coupled to task migration is an effective adaptive method to improve the system performance.

Future works include the measurement of performance metrics, as latency and communicating energy. Even if they are proportional to the hop number, its evaluation is important to corroborate the distributed management approach. It is also important to add in the monitoring system QoS parameters, as throughput, to adapt applications dynamically according the the system load.

6. REFERENCES

[1] Kobbe, S., et al. "*DistRM: Distributed Resorce Management for On-Chip Many-Core Systems*". In: ISSS11, 2011.

[2] Shabbir, A., et al. "*Distributed Resource Management for Concurrent Execution of Multimedia Applications on MPSoC Platforms*". In: SAMOS, 2011, pp. 132-139.

[3] Fattah, M., et al. "*Exploration of MPSoC Monitoring and Management Systems*". In: ReCoSoC, 2011.

[4] Al Faruque, M.A., Krist, R., and Henkel, J. "*ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication*". In: DAC, 2008.

[5] Stan, A., Valachi, A., and Bêrleanu, A. "*The design of a run-time monitoring structure for a MPSoC*". In: ICSTCC, 2011, 4p.

[6] Matos, D., et al. "*Monitor-Adapter Coupling for NoC Perfomance Tuning*". In: ICECS, 2010, pp. 193-199

[7] Bertozzi, S., et al. "*Supporting task migration in multi-processor systems-on-chip: a feasibility study*". In: DATE, 2006, pp. 15-20.

[8] Goodarzi, B., and Sarbazi-Azad, H. "*Task Migration in Mesh NoCs over Virtual Point-to-Point Connections*". In: Euromicro, 2011, pp.463-469.

[9] Almeida, G.; et al. "*Evaluating the Impact of Task Migration in Multi-Processor Systems-on-Chip*". In: SBCCI, 2010, pp. 73-78.

[10] Mandelli, M.; et al. "*Multi-task dynamic mapping onto NoC-based MPSoCs*". In: SBCCI, 2011, pp 191-196.

[11] Moraes, F., et al. "*Proposal and Evaluation of a Task Migration Protocol for NoC-based MPSoCs*". In: ISCAS, 2012, pp. 644-647.

[12] Carara, E., et al. "*HeMPS - a Framework for NoC-based MPSoC Generation*". In: ISCAS, 2009, pp. 1345-1348