Task Model Suitable for Dynamic Load Balancing of Real-Time Applications in NoC-based MPSoCs

Sérgio Johann Filho, Alexandra Aguiar, Felipe Göhring de Magalhães, Oliver Longhi, Fabiano Hessel
Faculty of Informatics - PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil
Email: {sergio.johann,felipe.magalhaes,oliver.longhi}@acad.pucrs.br,
{alexandra.aguiar,fabiano.hessel}@pucrs.br

***Abstract*—Modern embedded systems implemented through Multiprocessor System-on-Chip (MPSoCs) benefit themselves from resources that were previously available solely in general-purpose computers. Currently, these systems are able to provide more features at the cost of an increased design complexity. In this scenario, the applications' behaviour has changed. In the past, the majority of applications showed a static behaviour throughout their entire lifetime. Applications could be divided into tasks and mapped onto processing elements at design time. Currently, the applications' dynamic nature imposes that efficient dynamic load balancing techniques with different task mapping strategies must arise, although a fair static mapping still helps increasing the system overall performance. In this paper we present a task model suitable for dynamic load balancing of real-time applications with special support for Network-on-Chip (NoC)-based MPSoCs that aims to stabilize the system load throughout its lifetime. Results show a reduction in both system stabilization time (mean of 47.62%) and deadline misses (mean of 32.28%) for several benchmarks, compared to classic approaches which employ a centralized migration manager.**

I. Introduction

MPSoCs have become a solid reality in the last few years, enhancing and replacing many uniprocessor-based systems formerly adopted. In this context, in spite of enabling a whole new category of features and applications, the use of multiple processing elements in a single chip introduces new challenges such as an increased architecture complexity.

Along with these new platforms, the applications' behaviour have also changed. Currently, the entire system can be considered as more dynamic since the user himself can upload and request new functionalities which change the system load during its lifetime [1]. In this scenario, the efficient management of the system load considering the available resources and the constraints to be respected is, itself, an enormous challenge.

Besides dealing with dynamic applications, some other common restrictions present in embedded systems still remain. For instance, in many cases there are timing constraints that often require Real-Time Operating Systems (RTOSs) to be nicely met. Still, RTOSs also offer more standard interfaces which allow developers to have easier access to the underlying processing power while respecting applications' timing constraints.

On the hardware point of view, current MPSoC solutions are composed by tens to sometimes hundreds of processing elements and commonly use Network-on-Chip (NoCs) as the interconnection media. NoCs are scalable, modular and provide high communication parallelism and are, therefore, a suitable solution for dense MPSoCs [2], [3], [4].

Furthermore, MPSoCs can be classified according to their processing elements nature as either heterogeneous or homogeneous. While heterogeneous solutions offer several advantages such as reduced energy consumption with a good silicon area tradeoff [5], [6], [7], when compared to homogeneous solutions their design can be considered too complex from the implementation point of view. Therefore, homogeneous solutions usually allow a faster implementation of the design and higher levels of reuse regarding the same architecture for different products [8]. Still, homogeneous systems ease the design space exploration when high level tools are used to provide the mapping of the application tasks.

In this paper we propose a task model suitable for dynamic load balancing through task migration to be used in dense MPSoCs, especially the ones implemented through NoCs. Our model also contemplates real-time constraints in a homogeneous MPSoC aiming to provide a system load stabilization throughout its lifetime. To validate our proposal we implemented the entire load balancing mechanism in an RTOS in order to measure the systems' improvements. Results show a reduction in both system stabilization time (mean of 47.62%) and deadline misses (mean of 32.28%) for several benchmarks, compared to classic approaches which employ a centralized migration manager.

The remainder of the paper is organized as it follows. Section II presents some related work and discusses open topics in the area. The proposed model is presented on Section III, followed by Section IV which briefly describes our implementation. Results are shown on Section V followed by Section VI that concludes the paper.

II. Related Work

Nollet [9] presents a centralized, dynamic resource management scheme through a heuristic to manage task migrations over a NoC. In this system, a centralized OS manages the whole resource allocation. Carvalho [10] investigates the use of different run-time mapping heuristics targeting reduced network congestion on heterogeneous MPSoCs. Singh [11] presents several dynamic mapping heuristics on a heterogeneous MPSoC targeting reduced execution time, energy consumption and network congestion.

Many of the works in the literature [9], [12], [11], [10] use the concept of a centralized manager to perform migrations or dynamic mapping of tasks. This approach has some advantages such as easy global state management and reduced implementation complexity. The drawbacks are the concentration of the network traffic near a centralized manager, single failure spot and impossibility to perform several migrations or mappings in parallel. Still, as the number of processing elements rises in current MPSoCs, a single manager is not a scalable approach [12]. Other works use a very simple task model where the state is not kept nor considered on dynamic mapping, such as in [1]. Furthermore, other works consider only one task on each processing element such as in [10]. Improvements for these limitations are presented in the proposed model.

III. Proposed Model

A. Problem Definition

Several works [13], [14], [3] present solutions for the task mapping problem using static techniques, which offer optimal solutions for applications that have a profile known at design time. On the other hand, applications with varying characteristics at runtime require solutions that perform the dynamic mapping of tasks, presenting improvements for this type of application over static approaches [15], [8], [10]. The dynamic profile of an application can be the major cause of deadline misses, saturation on network channels and even die-thermal problems. Therefore, dynamic reconfiguration mechanisms can help to reduce these problems and to improve the application execution in this scenario.

In this context, a static mapping is performed on *initial tasks* of the application at design time. Then, the operating system executing on each processing element provides certain services for the dynamic mapping of tasks, adapting changes in the execution profile and optimizing the application execution.

B. Application

The application model proposed in this work includes a set of tasks that execute in several processing elements. Each task has its own parameters as presented following, along with the code and data used to implement its actual functionality.

A task is defined an 9-tuple $t_i =< p_i, uid_i, e_i, d_i, lc_i, pwr_i, cd_i, dt_i, ctx_i >$, where: id_i is the task local identification, uid_i is the task global identification, p_i stands for period of task i; e_i represents the task execution time or capacity; d_i is its deadline; lc_i stands for a communication related list; pwr_i represents the task energy requirements; cd_i, dt_i represent the task code and data segments, respectively; and finally, ctx_i stands for the task context.

Here, p_i, e_i and d_i parameters are defined by the application for each task in the system. The task set executes according a given scheduling policy[1] and the parameters must be informed in abstract time units, named as *tick*[2]. pwr_i, cd_i and dt_i require further characterization, as they depend on the technology, on the algorithm and on the chosen compiler.

Each node executing one or more tasks is called a *partition* and the task set composed of all related tasks executing on several nodes are responsible for implementing the entire application. Still, other applications can be inserted in the system at runtime and also, the task parameters of a given application may change dynamically. Besides, some tasks are fixed, that is, they are assumed to execute on the same processing element and have constant parameters throughout the system lifetime.

Communication among tasks is represented by the 2-tuple $c_{ij} =< t_j, \omega_{ij} >$, where t_j is the target task and ωij represents the data content for this particular communication. As the communication data generated by a single task may be composed by several communications, the total communication data volume is represented by $lc_i = \sum_{i=0}^{n-1} \omega_{ij}$, where n is the number of communications.

C. Architecture

The proposed architecture model represents processing elements connected through queues on a two dimensional, regular 2D-mesh topology NoC, using deterministic XY routing and wormhole packet forwarding. Each node is composed by a processing element, a network router and communication queues.

Thus, a node is defined by an 10-tuple $NPU_c =< ct_c, f_c, ld_c, nt_c, sp_c, o_c, m_c, iq_c, oq_c, v_c >$, where ct_c represents the processing element type; f_c is its clock frequency; ld_c stands for processor load; nt_c is the number of tasks; sp_c represents the scheduling policy; o_c is the operating system overhead, m_c stands for the processor memory size, iq_c and oq_c are the input and output queues sizes, respectively; and v_c is the data volume generated from tasks that execute on the node.

Each communication channel[3] is described by the 3-tuple $v_{ci} =< tc_j, dv_{ij}, h_{ij} >$, where tc_j is the target node; dv_{ij} represents the transmitted data content and h_{ij} is the number of hops between the source and target node, considering XY routing. The total data transmitted from one node to another is expressed by $dv_{ij} = \sum_{u=0}^{t-1} lc_n$, where $t-1$ is the last element of a list composed by the tasks that send data to tasks executing on node j. Still, parameters such as target core and number of hops are characterized after the mapping step. Finally, the link capacity for each node is defined as l_c.

D. Inter-task Communication

Tasks may either enter and leave the system at any time or have their parameters changed dynamically, modifying the initial mapping and its cost. Thus, the inter-task communication must be adapted to allow the communication of non-fixed tasks that can migrate among the nodes.

In this case, the communication protocol is implemented based in the classic MPI primitives *send()* and *receive()* and

[1]In this work, the Rate Monotonic [16] scheduling policy is assumed.

[2]*Tick* is the minimum scheduling unit. All real-time parameters are represented as *tick* values, and the *tick* itself has a defined time value (e.g. 1 ms).

[3]Here, a communication channel is a path established between source and target routers.

it assumes that at least one of the communicating tasks is not fixed. So, before the actual message is sent from one task to another, a *control message* is sent to the target node communication control task[4], asking for the target task's location. The control task verifies whether the target task is present, has died or migrated to another node, and sends a reply accordingly. If the referenced task is present indeed, the communication happens directly. However, if the task has died, no communication is performed while, if the task has migrated, another control message is sent from the source node to the new target node. The process is repeated until the communication is established. It is important to highlight that and a referenced task can not migrate before the completion of this process.

All control is kept by a task list at each node in order to coordinate the communication control. So, every time a task migrates or dies, a new entry is added to that list. Then, each source task has a local data structure that is updated only during a migration. Still, every time a task returns to its original node, it is removed from the list. In terms of implementation, when local tasks communicate they may use the same message-passing primitives, although, internally, a shared memory approach is adopted.

E. Dynamic Mapping and Task Migration

Each node has, along with its own instance of the operating system, a local task repository. Each node manages its own repository, defined at design time, which can be extended with migrated tasks' code and data. This approach differs from previous works as in [10], where a centralized repository is used. Moreover, the creation, replication or modification of task parameters changes the initial scenario used as a reference for an optimal static mapping. Sometimes a given node does not have enough resources to map and create a task, and an overload situation arises. This overload, considering the proposed model, makes the task partition of a given node either *not schedulable* or generates a *non optimized network traffic* due to bad dynamic mapping. A bad mapping can be optimized at runtime by migration managers, which are also fixed control tasks.

Task migrations are performed by local migration managers and communication control tasks. Initially, the task code is sent to the target node. Next, the task is blocked and its data and context contents are sent to the target. The target node relocates the task code, adds it to its local repository, maps the task and restores the context. The source node receives an acknowledge message, and then it kills the task. All steps are performed during the execution of migration managers and communication control tasks, so execution of other tasks in the system is not delayed. Only the migrated task remains blocked during part of this process.

Distributed migration managers are responsible for verifying each node status. As soon as a node enters an overload situation, the local manager chooses a task to migrate and queries neighbour nodes' local managers. The processor overload situation is calculated by $\sum_{j=0}^{N-1} \frac{e_j}{p_j} \leq N(2^{1/N} - 1)$ for the RM [16] scheduling algorithm and network overload is calculated by $v_c > l_c$. The neighbours reply their status (processor load, network load, number of tasks and free memory), so the querying manager decides the best candidate target to perform the migration. Free memory on node k is calculated such that $fm_k = m_k - (\sum_{i=0}^{N-1}(cd_i + dt_i) + cd_{os} + dt_{os})$.

The choice of which task should be migrated, and the target for the migration can make use of several different heuristics. In this work, a random non-fixed task is chosen and migrated to the closest neighbour with enough free processor and memory to run the task. The target node is chosen using the nearest neighbour algorithm. If no candidates for migration are found, the node remains overloaded and no further migration messages are sent for several rounds[5]. This way, the system remains stable, although overloaded. In the event a neighbour node has enough resources in the future, the overloaded node still has its chance to have one of its tasks migrated.

IV. Model Implementation

We used the HellfireOS[17] RTOS to implement the proposed model as it is fully preemptive, highly configurable and provides POSIX-like interfaces, standard libraries, timers, semaphores, dynamic memory allocation, debug facilities, several scheduling policies and communication drivers.

In terms of architecture, it was first modelled in Register Transfer Level (RTL), and then characterized and implemented on a MIPS-like[6] cycle accurate Instruction Set Simulator (ISS) in [18]. Processing elements, hardware queues and network routers are all emulated on the ISS, and the number of processors, network interface queues size, as well as mesh dimensions and routers internal queues are also configurable.

V. Results

We used an architecture simulator based on the work of [18] to perform our experiments. This simulator consists of several nodes[7], formed by MIPS-like processors with local memory interconnected in a 2D mesh topology NoC, using communication FIFOs and a network interface to connect processors to NoC routers. The simulator implements our MPSoC architecture model, presented in Section III-C, and it has a very close precision to our hardware prototype.

On all experiments, the task stack size was set in 2KB, the scheduling policy is Rate Monotonic, the operating core frequency for all nodes is 25MHz (unless otherwise specified) and the time between interrupts (*tick* time) is 10.48ms. The migration managers were configured to run each 104ms and use 10% of processing time. MPSoC size is variable (from

[4]Each node has a communication control task to allow knowing the location of a task.

[5]The algorithm increments by one the time between future queries for neighbour nodes every time no candidate is found. A round consists of an execution iteration of the migration manager.

[6]The architecture is implemented as a subset of the MIPS ISA.

[7]The number of nodes is parameterizable. In the current implementation, up to 256 nodes in a 16x16 mesh can be simulated.

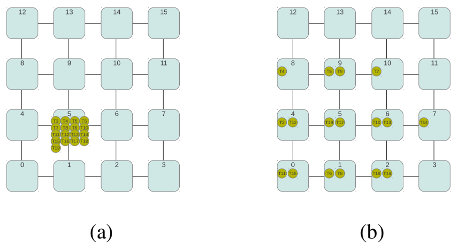
6 to 30 nodes) and each core has a 512KB shared data and instruction memory. Packet size is 64 flits on all tests and the receiving software queue has 16 slots for each core. Operating system and application code were optimized for size and compiled with GCC 4.6.1 for the MIPS target architecture.

A. *Case study: VOPD*

As the first case study, an application described by Murali [3] was used. This application describes the behaviour of several modules that implement a video object plane decoder, where data rates between modules are represented in an Application Task Graph (ATG) as it can be seen on Figure 1. Along with data rates (represented by edges), task parameters are represented in this graph. The first parameter is the task *local id*, the second is the task *unique id* and the last parameters represent the task *period*, *capacity* and *deadline*.

Each task represents an application module, and all tasks were configured to execute each 104ms. Two *ticks* were used as the task capacity, where one is used for processing and the other for communication. This application is composed by 17 tasks. The application modules are implemented as tasks as it follows: *demux* (t_3), *variable length decoder* (t_4), *run length decoder* (t_5), *inverse scan* (t_6), *AC/DC prediction* (t_7), *inverse quantizer* (t_8), *inverse cosine transform* (t_9), *up sample* (t_{10}), *VOP reconstruction* (t_{11}), *padding* (t_{12}), *VOP memory* (t_{13}), *up sample 2* (t_{14}), *reference memory* (t_{15}), *downsample and context calculation* (t_{16}), *arithmetic decoder* (t_{17}), *memory* (t_{18}) e *stripe memory* (t_{19}). Tasks t_4 (*variable length decoder*) and t_{17} (*arithmetic decoder*) define two independent start paths in the pipeline.

Figure 2 depicts the initial mapping (Figure 2(a)) where all application tasks are mapped to one node, corresponding to 340% of processor utilization. After several migrations, the final mapping (Figure 2(b)) is achieved. As it can be seen tasks are kept nearby, so network links are used in an optimized fashion. A maximum of only two application tasks could be mapped per node, because they were configured with tight real-time parameters (20% for each task of the application and 10% for the migration manager). There was only one application mapped to one core initially in this scenario, so only the migration manager of node 5 performed migrations.



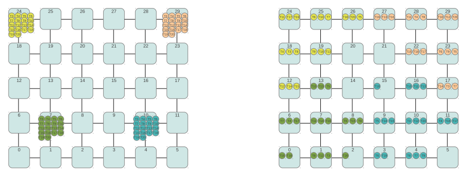
(a) (b)

Fig. 2. VOPD application, distributed managers: initial mapping (a); final mapping (system stabilization) (b)

B. *Case study: MPEG4*

As the second case study, an application described by Milojevic [19] with the behaviour of an MPEG4 encoder was used. The author describes the application as data rates between modules, but in this work we characterized the application with real-time parameters. In this application, tasks were configured to execute each 115ms, and again, two *ticks* were used as task capacity. This application is composed by 18 tasks, and it is described in the ATG graph depicted on Figure 3. The application modules are described by the tasks: *SRAM New Frame* (t_3), *Input Control* (t_4), *FIFO Current MBL* (t_5), *FIFO New MBL* (t_6), *ME* (t_7), *FIFO MV* (t_8), *MC* (t_9), *Error Block* (t_{10}), *Comp Block* (t_{11}), *Texture Coding* (t_{12}), *Texture Block* (t_{13}), *Texture Update* (t_{14}), *SRAM RecFrame* (t_{15}), *Copy Controller* (t_{16}), *Search Area* (t_{17}), *YUV Buffer* (t_{18}), *Quantized MBL* (t_{19}) and *VLC* (t_{20}).

In this scenario, four instances of this application were mapped to individual nodes on a 6x5 MPSoC. Each task uses 18.18% of processing time, so a maximum of three application tasks could be allocated per node. Each application instance has a 327% of processor utilization, so the nodes are clearly overloaded. Figure 4 represents the initial (Figure 4(a)) and final mappings (Figure 4(b)). Independent pipelines are mapped to nodes 7, 10, 24 and 29. After migrations, tasks from independent pipelines are kept close to each other, improving the application deadlines along with network utilization.



(a) (b)

Fig. 4. MPEG4 application, distributed managers: initial mapping (a); final mapping (system stabilization) (b)

Figures 5 and 6 depict processor load over time for two different migration manager approaches. The first one uses centralized control of migrations and the second one uses the proposed distributed migration managers. In the first scenario, migrations are performed between *ticks* 100 and 1380. In the second scenario, migrations are performed between *ticks* 100 and 700. On both scenarios 60 migrations are performed. After *tick* 700 (Figure 6) the application has all its tasks mapped, and an iteration of each pipeline occurs in 2306ms.

As it can be seen, the distributed managers performed migrations faster than a centralized one. Although several messages had to be exchanged among managers in the distributed approach, a centralized manager has to be updated every time a node has a load change. This generates a large amount of messages between the centralized manager (master) and other nodes (slaves), introducing a large overhead considering the proposed task model.

C. *Experiments*

Synthetic and real world applications were tested, and a summary of the experiments is shown on Tables I and II. Along with different applications the MPSoC size and number

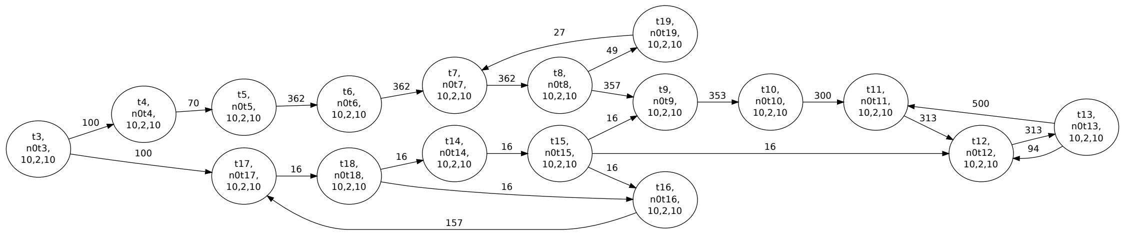


Fig. 1. VOPD application task graph

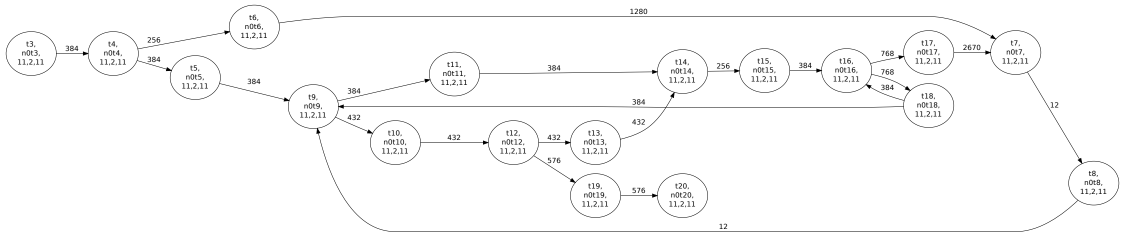


Fig. 3. MPEG4 application task graph

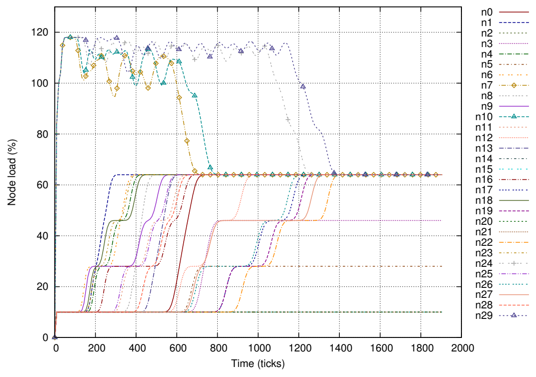


Fig. 5. MPEG4 application, node load over time: centralized manager

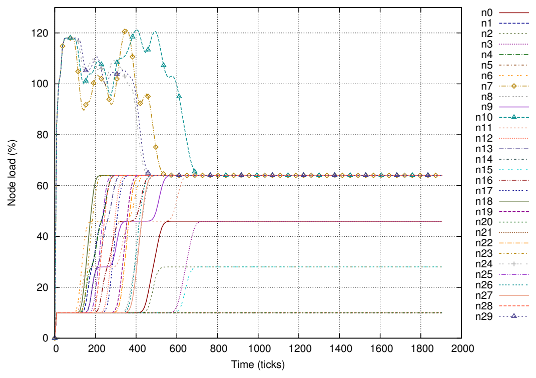


Fig. 6. MPEG4 application, node load over time: distributed managers

of tasks was varied, so a large set of benchmarks was used. Improvements on stabilization time and deadline misses are relative to the centralized manager approach. CM stands for *centralized manager* and DM for *distributed managers*.

The applications used as benchmarks are characterized as it follows. *Synthetic app 1* is composed by 18 tasks, and another one is mapped at run-time. This is a relatively complex application, where task parameters are varied and there is a great dependence among tasks. Tasks are mapped to 4 nodes initially. *Synthetic app 2* is an even more complex application, composed by 36 tasks (plus 2 mapped at run-time) of varied real-time parameters and great dependence among tasks. Tasks on this application are mapped to 8 nodes initially. Applications *Synthetic app 3* and *Synthetic app 4* are simpler applications, composed by tasks with similar real-time parameters and less task dependence. MJPEG, VOPD and MPEG4 are typical benchmarks.

Table I presents results concerning situations where several nodes are overloaded. As it can be seen, the distributed approach has a great advantage for this type of application, presenting reductions on stabilization time (a mean of 47.62%) and deadline misses (a mean of 32.28%). It can be observed that the proposed approach is highly scalable as the number of processors and tasks increase, along with the probability of overload situations in the same system.

Table II presents results concerning situations where just one node in the system is overloaded. For applications with a reduced number of tasks, there is no advantage in the distributed approach. As the number of tasks increases, the distributed managers approach presents an increasing advantage in stabilization time (a mean of 12.38%) and reduction of deadline misses (a mean of 7.83%).

VI. Conclusions

Recent applications use the computing power of MPSoCs by extending their features at runtime. This kind of application introduces new challenges, and efficient mechanisms are required to adapt hardware resources to the application needs. This paper proposes a model that can be used to describe such complex applications and adapt the available resources dynamically in a fast and efficient manner. Results show that a distributed approach may be more efficient for dynamic task

TABLE I
MIGRATION MANAGER EXPERIMENTS, MULTIPLE OVERLOADED NODES

Application	MPSoC size	Tasks	Migrations		Stabilization time			Deadline misses		
			CM	DM	CM	DM	Improvement	CM	DM	Improvement
Synthetic app 1	3x2	18+1	5	5	175	90	48.57%	8	5	37.50%
Synthetic app 2	4x4	36+2	11	10	275	160	41.81%	14	6	57.14%
Synthetic app 3	3x2	12	5	6	100	75	25.00%	10	9	10.00%
Synthetic app 3	4x4	16	8	8	210	65	69.04%	20	12	40.00%
Synthetic app 3	6x5	28	14	14	290	65	56.66%	33	24	27.27%
Synthetic app 4	3x2	27	9	9	240	205	14.58%	8	8	0.00%
Synthetic app 4	4x4	36	12	12	310	90	69.35%	13	8	38.46%
Synthetic app 4	6x5	63	20	20	275	90	67.27%	18	12	33.33%
MJPEG	3x2	8	4	4	115	90	21.73%	32	29	9.37%
VOPD	6x5	68	56	56	1200	400	66.66%	2459	1080	56.07%
MPEG4	6x5	72	60	60	1280	600	53.12%	2383	1288	45.95%

TABLE II
MIGRATION MANAGER EXPERIMENTS, SINGLE OVERLOADED NODE

Application	MPSoC size	Tasks	Migrations		Stabilization time			Deadline misses		
			CM	DM	CM	DM	Improvement	CM	DM	Improvement
Synthetic app 3	6x5	4	2	2	65	65	0.00%	2	2	0.00%
Synthetic app 4	6x5	9	3	3	95	90	5.26%	3	3	0.00%
MJPEG	3x2	4	2	2	45	45	0.00%	0	0	0.00%
VOPD	4x4	17	15	15	505	350	30.69%	361	311	13.85%
MPEG4	4x4	18	15	15	520	385	25.96%	363	271	25.34%

mapping than a centralized approach and also more scalable for larger MPSoCs.

REFERENCES

[1] G. Marchesan Almeida, G. Sassatelli, and P. Benoit, “An adaptive message passing mpsoc framework,” *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–21. [Online]. Available: http://hindawi.com/journals/ijrc/2009/242981.pdf

[2] J. Hu and R. Marculescu, “Energy- and performance-aware mapping for regular noc architectures,” *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 24, no. 4, pp. 551–562, 2005.

[3] S. Murali and G. De Micheli, “Bandwidth-constrained mapping of cores onto noc architectures,” in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, ser. DATE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 20 896–. [Online]. Available: http://portal.acm.org/citation.cfm?id=968879.969207

[4] C.-L. Chou and R. Marculescu, “Incremental run-time application mapping for homogeneous nocs with multiple voltage levels,” in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS ’07. New York, NY, USA: ACM, 2007, pp. 161–166. [Online]. Available: http://doi.acm.org/10.1145/1289816.1289857

[5] D. Lyonnard *et al.*, “Automatic generation of application-specific architectures for heterogeneous multiprocessor,” in *DAC’2001 - Design Automation Conference*. New Orleans, EUA: ACM Press, Jun 2001.

[6] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, “Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip,” in *DATE ’03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10986.

[7] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, “Low cost task migration initiation in a heterogeneous mp-soc,” in *DATE ’05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 252–253.

[8] C.-L. Chou and R. Marculescu, “User-aware dynamic task allocation in networks-on-chip,” in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE ’08. New York, NY, USA: ACM, 2008, pp. 1232–1237. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403675

[9] V. Nollet, T. Marescaux, P. Avasare, and J.-Y. Mignolet, “Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles,” in *DATE ’05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 234–239.

[10] E. L. de Souza Carvalho, N. L. V. Calazans, and F. G. Moraes, “Dynamic task mapping for mpsocs,” *IEEE Design and Test of Computers*, vol. 27, pp. 26–35, 2010.

[11] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, “Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms,” *J. Syst. Archit.*, vol. 56, pp. 242–255, July 2010. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2010.04.007

[12] A. Ngouanga, G. Sassatelli, L. Torres, T. Gil, A. Soares, and A. Susin, “A contextual resources use: a proof of concept through the apaches’ platform,” *Design and Diagnostics of Electronic Circuits and Systems*, vol. 0, pp. 42–47, 2006.

[13] H. Orsila, T. Kangas, E. Salminen, T. D. Hamalainen, and M. Hannikainen, “Automated memory-aware application distribution for multi-processor system-on-chips,” *J. Syst. Archit.*, vol. 53, pp. 795–815, November 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1282861.1282883

[14] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner, “Time and energy efficient mapping of embedded applications onto nocs,” in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC ’05. New York, NY, USA: ACM, 2005, pp. 33–38. [Online]. Available: http://doi.acm.org/10.1145/1120725.1120738

[15] A. Mehran, A. Khademzadeh, and S. Saeidi, “Dsm: A heuristic dynamic spiral mapping algorithm for network on chip,” *IEICE Electronics Express*, vol. 5, no. 13, pp. 464–471, 2008.

[16] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: exact characterization and average case behaviour,” *IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.

[17] A. Aguiar, S. F. Johann, F. G. Magalhaes, T. D. Casagrande, and F. Hessel, “Hellfire: A design framework for critical embedded systems’ applications,” in *ISQED ’10*. IEEE, 2010, pp. 730–737.

[18] S. F. Johann, A. Aguiar, C. A. M. Marcon, and F. P. Hessel, “High-level estimation of execution time and energy consumption for fast homogeneous mpsocs prototyping,” in *RSP ’08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 27–33.

[19] D. Milojevic, L. Montperrus, and D. Verkest, “Power dissipation of the network-on-chip in a system-on-chip for mpeg-4 video encoding,” *2007 IEEE Asian SolidState Circuits Conference*, pp. 392–395, 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4425713