Journal of Systems Architecture 63 (2016) 80–92

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Hierarchical energy monitoring for task mapping in many-core systems

Guilherme Castilhos[a], Marcelo Mandelli[a], Luciano Ost[b], Fernando Gehm Moraes[a,*]

[a] PUCRS University, Computer Science Department, Porto Alegre 90619-900, Brazil
[b] University of Leicester, Department of Engineering, Leicester, UK

ARTICLE INFO

Article history:
Received 23 July 2015
Revised 21 December 2015
Accepted 27 January 2016
Available online 9 February 2016

Keywords:
Energy-aware task mapping
Monitoring
Load balance
Energy consumption
Many-core systems

ABSTRACT

This work addresses a research subject with a rich literature: task mapping in NoC-based systems. Task mapping is the process of selecting a processing element to execute a given task. The number of cores in many-core systems increases the complexity of the task mapping. The main concerns in task mapping in large systems include (i) scalability; (ii) dynamic workload; and (iii) reliability. It is necessary to distribute the mapping decision across the system to ensure scalability. The workload of emerging many-core systems may be dynamic, i.e., new applications may start at any moment, leading to different mapping scenarios. Therefore, it is necessary to execute the mapping process at runtime to support a dynamic workload assignment. The workload assignment plays an important role in the many-core system reliability. Load imbalance may generate hotspots zones and consequently thermal implications, which may generate hotspots zones and consequently thermal implications. More recently, task mapping techniques aiming at improving system reliability have been proposed in the literature. However, such approaches rely on centralized mapping decisions, which are not scalable. To address these challenges, the main goal of this work is to propose a hierarchical runtime mapping heuristic, which provides scalability and a fair workload distribution. Distributing the workload inside the system increases the system reliability in long-term, due to the reduction of hotspot regions. The proposed mapping heuristic considers the application workload as a function of the consumed energy in the processors and NoC routers. The proposal adopts a hierarchical energy monitoring scheme, able to estimate at runtime the consumption at each processing element. The mapping uses the energy estimated by the monitoring scheme to guide the mapping decision. Results compare the proposal against a mapping heuristic whose main cost function minimizes the communication energy. Results obtained in large systems, up to 256 cores, show improvements in the workload distribution (average value 59.2%) and a reduction in the maximum energy values spent by the processors (average value 32.2%). Such results demonstrate the effectiveness of the proposal.

1. Introduction

Many-core systems have been employed to provide the high demands of performance while maintaining energy efficiency during the execution of concurrent embedded applications (e.g. video compressing, wireless communication standards, gaming). Such systems increase performance by using multiple homogeneous or heterogeneous processors. Many-core systems also integrate memories, dedicated hardware cores, and a communication infrastructure to interconnect the system components, as NoCs (Networks-on-Chip) and buses. Despite the higher design complexity of NoCs, such communication infrastructure offers better scalability, performance and power capabilities when compared to buses [1].

Applications designed to execute in many-core systems may be partitioned into different tasks to execute in different cores, enabling its parallel execution [2]. A task is a set of instructions and data, containing information and constraints for its correct execution in a given core. Additionally, tasks exchange data with other tasks during the execution of the application. The definition in which system core each task will execute is a major issue in the design of many-core systems. In the literature, this issue is defined as *task mapping* [2].

Task mapping decision should be executed at runtime to deal with time-varying workloads caused by the most of the embedded

* Corresponding author. Tel.: +555133538717.
E-mail addresses: guilherme.castilhos@acad.pucrs.br (G. Castilhos), marcelo.mandelli@acad.pucrs.br (M. Mandelli), luciano.ost@leicester.ac.uk (L. Ost), fernando.moraes@pucrs.br, moraesfg@gmail.com (F.G. Moraes).

http://dx.doi.org/10.1016/j.sysarc.2016.01.005
1383-7621/

system applications [3]. Such variations cannot be accurately predicted during design time, such as the scenarios when the system interacts with complex deployment environments or user-driven requests [4]. Runtime approaches (also referred as online or dynamic mapping approaches) require simple and fast mapping solutions since high time-consuming, and high computational algorithms may compromise the system performance. Further, runtime mapping can better lead with other system changes during runtime, such as cores availability and defective cores [2].

The increasing number of cores also requires scalable and hierarchical mapping solutions. Novel systems, with dozens of cores, are already present in the market [5,6] and ITRS roadmap [7] projects systems integrating thousands of cores by the end of the decade. In such systems, a centralized mapping decision compromises the system performance since a single core handles all mapping requests [8]. Also, centralized mapping contributes to increasing NoC congestion around the mapper leading to hotspot zones, which may result in system failures.

Reliability is an important concern related to task mapping, tightly connected to the workload distribution [9–11]. Load imbalance decisions can generate hotspots zones (i.e. peaks of power dissipation) and thermal variations, which affects directly system reliability [9–12]. This issue is worse in many-core systems, increasing power densities and, consequently, system temperature. Further, mapping communicating tasks far from each other result in more data transfer through the system, increasing communication latency and energy consumption. Unusable cores induce mapping of applications onto other system cores, increasing their workload and, consequently, reducing their lifetime.

To develop a hierarchical runtime mapping heuristic aiming a fair workload distribution it is necessary to have available accurate information (e.g. power, energy, temperature) to map the tasks. Reliability, temperature, and lifetime are tightly connected to the consumed energy into the system [13]. Thus, a monitoring scheme should provide energy data to the mapping heuristic. Therefore, the energy monitoring scheme is key for the effectiveness of the mapping heuristic.

The main *goal* of the current work is to propose a new mapping heuristic tackling the following features: runtime execution (dynamic), scalability, and workload distribution. The mapping decisions are guided at runtime by a hierarchical energy monitoring scheme, not requiring application profiling or thermal sensors.

This paper is organized as follows. Section 2 reviews the state-of-art in dynamic mapping heuristics, comparing qualitatively our proposal to the related works. Section 3 details the application model. Section 4 presents the energy model. This model is integrated into the operating system of the processing elements, enabling the energy monitoring at runtime. The hierarchical energy scheme is detailed in Section 5. Section 6 details the mapping heuristic. Section 7 presents results, and Section 8 concludes this paper.

2. State-of-art

Task mapping literature is wide, requiring a taxonomy considering different mapping criteria. Authors in [14,15] classifies the mapping process according to four criteria:

(i) *Target architecture*. Task mapping can be executed in homogeneous (identical processing elements) or heterogeneous (e.g. DSPs, dedicated IPs, accelerators) systems.
(ii) *Number of tasks per PE*: single or multi-task. Single-task assumes only one task assignment per PE while multi-task allows mapping more than one task per PE according to some criteria (e.g. communication, execution time, task deadlines). A multi-task approach can better explore system resources, enabling the execution of an increasing number of applications in parallel.
(iii) *The moment in which it is executed*: design-time or runtime. Design-time approaches are not suitable to dynamic and unpredictable workloads imposed by the execution of different applications. Runtime task mapping enables different applications to be inserted into the system at runtime, enabling dynamic workloads.
(iv) *Mapping management*: centralized or hierarchical. Centralized mapping uses a single core responsible for the overall management, which is suited for small systems due to scalability issues. In a hierarchical approach, the mapping management is distributed in different cores, increasing system scalability and reliability.

This paper focuses on general-purpose many-core systems, able to execute several applications that are unknown in advance. This paper also assumes that underlying applications can be inserted into the system in a non-deterministic way, according to user requirements. The literature contains several runtime-mapping approaches. Table 1 summarizes the reviewed works according to the mapping taxonomy.

Only few works related to multi-task mapping were found in the literature, proposed by Singh et al. [2,3,24] and Mandelli et al. [30]. Multi-task techniques include clustering, which groups tasks to be executed in the same PE. A non-optimized clustering approach may lead to hotspots, reducing system lifetime and accelerating system wear out. Heterogeneous systems may have better performance for specific applications, and homogeneous systems are general-purpose platforms. As industrial examples [5,6], the present work focuses the research in homogeneous architectures. Another important feature is the hierarchical system management approach, as proposed in [25,26,30]. Such approach is scalable and can reduce the mapping algorithm computational effort, increasing system performance.

The literature presents different runtime task mapping approaches to improve system reliability. All reviewed works use a centralized system management approach [9,18,27–29]. Among them, some works [28,29] produce mapping decisions at design time, which are stored in a database and used at runtime. This approach may reduce system performance due to its incapability of dealing with unpredictable system variations. Task mapping approaches proposed in [9,27], employ physical sensors to capture thermal or wear-state condition of cores at runtime. Included sensors provide accurate information to the mapping decision at the cost of the additional system area and energy consumption. Huang et al. [31] use an abstract system to validate the proposed approach, which can produce inaccurate performance results.

The literature presents hierarchical approaches to improve system reliability. However, such approaches use other techniques rather than task mapping [32–34]. Ge et al. [32] propose a task migration approach for thermal balancing. This approach uses thermal sensors, which aggregate hardware costs. Wu et al. [33] present a dynamic frequency scaling for thermal management, which may impose additional hardware costs. Liu et al. [34] also present a thermal management task migration approach, which does not consider performance costs.

Mandelli et al. [30] propose the LEC-DN (Lower Energy Consumption based on Dependencies-Neighborhood) heuristic, a hierarchical mapping approach whose main function is to reduce the communication energy. To minimize communication energy, the LEC-DN heuristic aims to reduce the distance in hops between communicating tasks. When a given task t_i is required to be mapped, this heuristic first analyzes the set of communicating tasks with t_i already mapped. Then, the heuristic approximates t_i to the tasks it has a higher communication volume.

Table 1
State-of-the-art in dynamic mapping heuristics.

Author / Year	Multi/Mono-task	Architecture model	Management	Optimization Goal
Smit et al. [16] (2005)	Mono-task	Heterogeneous	Centralized	Energy Consumption and QoS requirements
Ngouanga et al. [17] (2006)	Mono-task	Homogeneous	Centralized	Communication volume, computation load
Coskun et al. [18] (2009)	Mono-task	Homogeneous	Centralized	System Reliability
Chou et al. [4] (2010)	Mono-task	Homogeneous	Centralized	Energy Consumption, Internal and external network contention
Hölzenspies et al. [19] (2008)	Mono-task	Heterogeneous	Centralized	Energy consumption and QoS requirements
Al Faruque et al. [8] (2008)	Mono-task	Heterogeneous	**Hierarchical**	Execution time, mapping time and monitoring traffic
Wildermann et al. [20] (2009)	Mono-task	Homogeneous	Centralized	Communication latency, energy consumption
Schranzhofer et al. [21] (2009)	Mono-task	Homogeneous	Centralized	Energy consumption
Lu et al. [22] (2010)	Mono-task	Homogeneous	Centralized	Communication latency and energy consumption
Carvalho et al. [23] (2010)	Mono-task	Heterogeneous	Centralized	Network contention, communication volume
Singh et al. [2][3][24] (2010)	**Multi-task**	Heterogeneous	Centralized	Network contention, communication volume and energy consumption
Kobe et al. [25] (2011)	Mono-task	Homogeneous	**Hierarchical**	Execution time, Communication traffic
Cui et al. [26]	Mono-task	Homogeneous	**Hierarchical**	Communication traffic energy consumption
Hartman et al. [27] (2012)	Mono-task	Homogeneous and Heterogeneous	Centralized	System reliability
Chantem et al. [9] (2013)	Mono-task	Homogeneous	Centralized	System reliability
Bolchini et al. [28] (2013)	Mono-task	Homogeneous	Centralized	Energy consumption and system lifetime
Das et al. [29] (2014)	Mono-task	Homogeneous	Centralized	Application deadlines and system lifetime
Mandelli et al. [30] (2015)	**Multi-task**	Homogeneous	**Hierarchical**	Communication energy reduction
Proposed work	**Multi-task**	**Homogeneous**	**Hierarchical**	**Workload distribution and communication volume**

This paper proposes a task mapping approach that *differs from literature* since it includes all the following characteristics:

- *Executed at runtime.* The proposed approach can better manage time-varying workloads and system changes.
- *Hierarchical mapping approach*. The proposed approach is implemented in a many-core managed in a hierarchical way. Such hierarchical system management improves system scalability by dividing the system into regions, each one with a manager responsible for actions inside it. Further, it reduces mapping decision computational effort, not compromising the system performance.
- *Induces a better system reliability.* The proposed approach aims to improve energy balancing, which is directly related to a better system reliability [9,10].
- *Hierarchical energy monitoring*. The proposed approach does not employ physical sensors in the mapping decision. The energy data is obtained at runtime using a hierarchical monitoring approach.
- *Clock-cycle model for validation*. The proposed mapping approach is validated in a large many-core system (up to 256 processing elements), modeled in clock-cycle RTL SystemC.

3. Application model

An application is modeled as a graph GApp = (T, E), where each vertex $t_i \in T$ represents an application task and each directed weighted edge $e_{ij} \in E$ represents a communication dependence between tasks t_i and t_j. The weight of an edge e_{ij} is denoted by $comm_{ij}$, representing the total data communication volume transferred between application tasks t_i and t_j. Fig. 1 presents an example of an application modeled as a task graph. Applications may be periodic or aperiodic. If the application is periodic (e.g. video decoding), the task graph represents one iteration of the application.

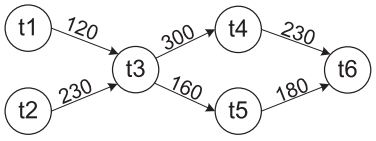


Fig. 1. Application modeled as a task graph GApp = (T, E). Initial tasks: t1, t2. Non-initial tasks: t3, t4, t5, t6.

An application has *initial tasks* (e.g. t_1 and t_2) and *non-initial tasks*. Initial tasks are those that initialize the execution of the application when mapped in the system. Such tasks do not have dependencies on other tasks to start to execute. A task $t_i \in T$ contains a set C_i called communication task list. This set is defined as $C_i = \{(t_j, comm_{ij}); (t_k, comm_{ik}); \ldots (t_n, comm_{in})\}$, where each element is a tuple containing a task t_j that communicates with t_i and the value $comm_{ij}$, corresponding to the total volume transferred between t_i and t_j in both directions (i.e. t_i to t_j and t_j to t_i).

Each application has an *application description* file containing information used to guide mapping decision. Such file contains: (*i*) the application size, which corresponds to the total number of tasks of the application; (*ii*) list of initial tasks; (*iii*) the set C_i for each task t_i, of the application.

All communication between tasks occurs through message passing. Inter-task communication uses send and receive MPI-like primitives.

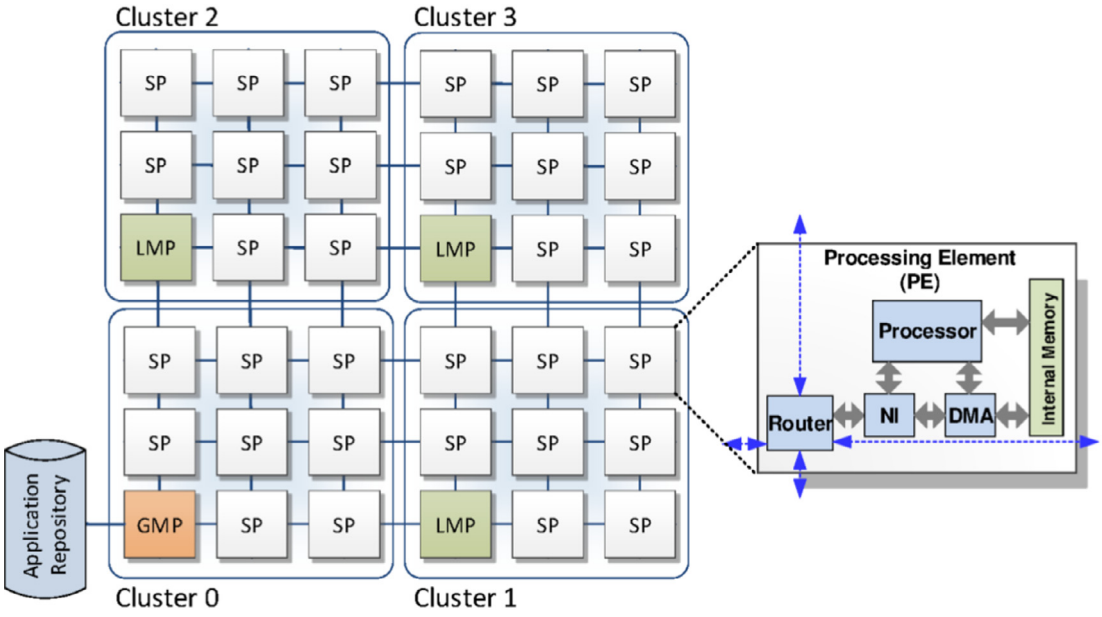


Fig. 2. Example of a 9×9 MPSoC instance, with hierarchical management.

4. Energy model

The energy consumption in a many-core system is mainly due to three components: memory, processors, and NoC (routers and links). The number of memory accesses is identical for the same workload. Therefore, to fairly compare different mapping solutions using the same workload, we consider the energy consumption of both processor and NoC as main metrics.

As described in the literature [35], the energy consumption (*EC*) of a processor pe_i is defined by its static and dynamic consumption. The processor *EC* related to the execution of a given task is a function of the number of executed instructions. In our model, the energy cost of each instruction is determined from a gate-level implementation of the processor, as proposed by Rosa et al. [36].

Each processor pe_i contains an *instruction analyzer* module, which counts the number of executed instructions for different classes at runtime. The set of classes is defined as $C=\{c_0, c_2,\ldots,c_8\}$, with 9 different classes (e.g. arithmetic, logic, branch) [36]. Results show that the error of adopted *instruction analyzer module* varies from 0.06% to 8.05% when compared to a gate-level implementation [36]. The *instruction analyzer* module corresponds to nine instruction counters, included in the control part of the processor. If the hardware of the processor cannot be modified, a sniffer may be added in the address and instruction buses. The instruction counters are specific purpose registers containing the number of executed instructions per class. The instructions per class registers are continuously updated. The area overhead due to this module in the processor corresponds to 6.4%, and in the whole PE it is inferior to 2%.

The processor energy consumption for a given *monitoring period* is obtained according to Eq. 1.

$$E_{processor} = \sum_{i=0}^{8} (energy(c_i) * total\_instructions(c_i)) \quad (1)$$

where: $energy(c_i)$, energy to execute a given instruction belonging to the class c_i, value obtained by simulating the synthesized processor; $total\_instructions(c_i)$, number of executed instructions belonging to the class c_i in the *monitoring* period.

The NoC *EC* is proportional to the number of transmitted flits at each router port [37]. A gate level description of the NoC is used to determine the energy consumption of the main router components: buffers, internal crossbar and control logic. Eq. 2 gives the energy consumption for a given *monitoring period.*

$$E_{router} = nb\_flits * E_{buffer} + E_{crossbar} + E_{control\_logic} \quad (2)$$

where: *nb_flits* correspond to the number of flits transferred by the router during the *monitoring period;* E_{buffer}, $E_{crossbar}$, and $E_{control\_logic}$ to the energy consumption of the main router components during the *monitoring period*.

Most of the time, the NoC consumes only static power, since the injection rate induced by the processors is typically inferior to 5% (similar injection rate was observed in [38]). Experimental results observed in [37] show that most of the consumed energy comes from processors (roughly 90%). Even if the injection rate is small, it is important to reduce the hop count to reduce the shared resources in the NoC. Increasing the number of shared resources in the NoC may lead to congestion and performance degradation due to increased latency.

Each PE monitors the processor and router energy according to a parameterizable *monitoring period*. The monitoring scheme uses these values to guide the mapping heuristic.

5. Hierarchical monitoring method

The many-core system adopted in this work is a general purpose homogeneous MPSoC in which processing elements (PEs) are interconnected through a NoC. The system uses distributed memory architecture, based on *scratchpad* memories rather than cache memory. The system adopts scratchpad as local storage memories due to its power efficiency and management facilities when compared to cache memories. Further, scratchpad memory is more predictable in terms of access time, and it does not require any coherence protocol, as required by cache-based architectures [39]. The adopted architecture does not contain shared memories.

The MPSoC architecture can be defined as a directed graph $GMPSoC=(PE, L)$. Each vertex $pe_i \in PE$ is a processing element. An *edge* $l_{ij} \in L$ is a NoC link interconnecting pe_i to pe_j. Each PE contains a processor, a local memory, a DMA module, a network interface and a router (Fig. 2). An external memory, named *application repository*, contains the object code of the application tasks to execute in the system.

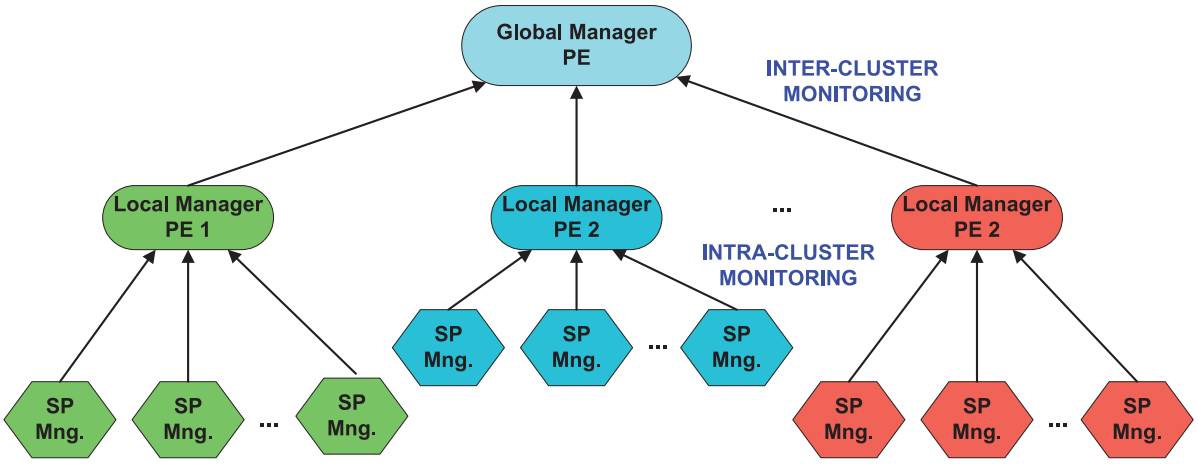


Fig. 3. Hierarchical monitoring method.

The local memory of each PE, which default size is 32 KB, stores the μkernel (simple operating system), the code and data for the tasks assigned to the PE. The local memory is organized into equally sized pages to simplify the memory management. The number of pages in SPs is defined as *SP_PAGES*. While the first page stores the μkernel (9.5 KB), the remaining *SP_PAGES* are used to store the application tasks. If a given task does not fit on one page, the task should be partitioned into smaller tasks. The memory size is a design parameter, being possible to fit this parameter according to the workload to execute in the system.

To enable the hierarchical system management, the system is divided into virtual regions, named *clusters* (Fig. 2) [40]. For this purpose, processing elements may assume one of three roles:

- **Slave Processing Element (SPs)**. SPs execute application tasks. Each SP runs the μkernel, which supports communication between PEs, multitask execution and software interrupts (traps). Each SP can execute MAX_SP_TASKS tasks simultaneously, which corresponds to SP_PAGES -1.
- **Local Manager Processing element (LMP)**. Responsible for cluster control, executing functions such as task mapping, task-migration, and re-clustering (process to requests SPs to neighbor clusters).
- **Global Manager Processing Element (GMP)**. A single PE responsible for the overall system management, such as defining application-to-cluster mapping, controlling external devices accesses (e.g. application repository). Further, the GMP manages one of the system clusters (for example, the bottom left cluster of Fig. 2), executing all functions of an LMP.

The definition of the clusters' size occurs at design time. When the system starts, the GMP handles the clusters' initialization, notifying the LMPs the region they will manage. Then, when an LMP knows the region it will control, it informs all SPs in this region that it will be their manager. This cluster and SPs initialization mechanism provide better system adaptability. For example, run-time re-clustering process enables the modification of the cluster size. The re-clustering process occurs when there are no available SPs inside a cluster to map an application task. The LMP checks the availability of cluster resources when a task is requested to be mapped. If there is no SP available inside the cluster to receive the requested task, an SP is borrowed from neighbor clusters [40]. When the task finishes its execution, the borrowed SP is released to the original cluster.

The proposed hierarchical monitoring approach comprises intra- and inter-cluster monitoring, as illustrated in Fig. 3.

Fig. 4 illustrates the hierarchical monitoring protocol. SPs periodically send monitoring packets to their LMP with the consumed energy of the PE (processor and router), and the LMP updates its energy table. LMPs update the GMP when a task is requested to be mapped, when an application finishes its execution or periodically.

5.1. Intra-cluster monitoring

Intra-cluster monitoring is the process by which each LMP receives information related to the amount of energy each SP has consumed during the *monitoring period,* according to Eqs. 1 and 2. The μkernel periodically computes the energy spent at each SP, transmitting the obtained value to the LMP. Note that the LMPs know the workload (consumed energy) of each SP, which enables the LMPs to execute heuristics to distribute the workload evenly over the time.

This process induces a small amount of traffic in the NoC, being local to each cluster. Also, as the number of SPs in each cluster is small (typically 16), the computational load to treat the monitoring packets in each LMP is small. On the one side, the number of monitoring packets increases with small monitoring periods, overloading the LMP. On the other side, large monitoring periods delay the computation of the consumed energy by the SPs, leading to wrong mapping decisions. Section 7.1 discusses this trade-off evaluating different monitoring periods.

5.2. Inter-cluster monitoring

Inter-cluster monitoring is the process by which the GMP receives the information related to the amount of energy consumed within each cluster. Whenever an LMP to the GMP communication occurs, the cluster energy is inserted in the packet. Such approach avoids overloading the GMP with monitoring messages. Two messages in which the monitoring information is inserted are:

- *NewTask* – the LMP requests an allocation of a new task;
- *AppTerminated* – the LMP reports to the end of a given application. The LMP sends this message when all tasks of a given application finished their execution.

Tasks executing for long periods would not update the GMP, leading to a cluster energy underestimation. Therefore, each LMP notifies the GMP periodically with the consumed energy at each cluster. This inter-cluster monitoring period is larger than the intra-cluster monitoring. Note that the GMP only knows the total energy spent at each cluster, not having a detailed view of the energy distribution.

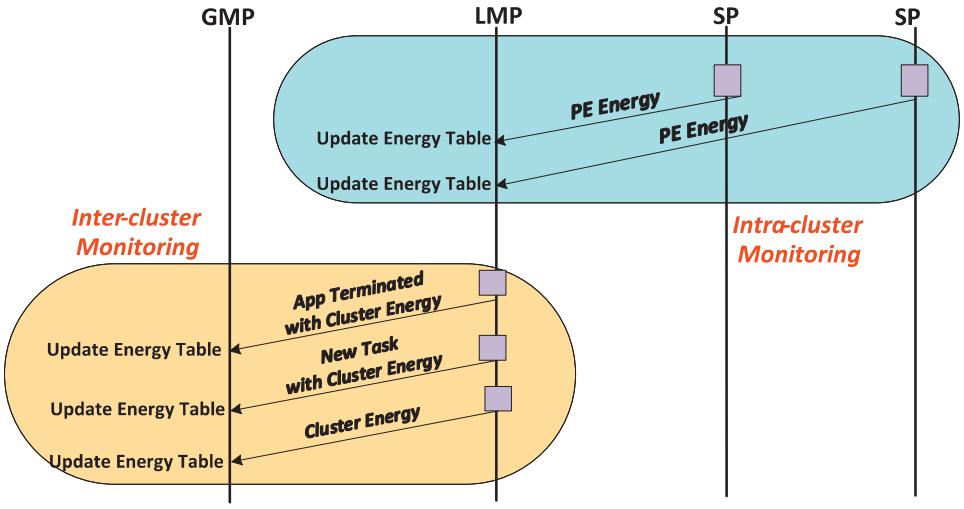


Fig. 4. Hierarchical monitoring protocol.

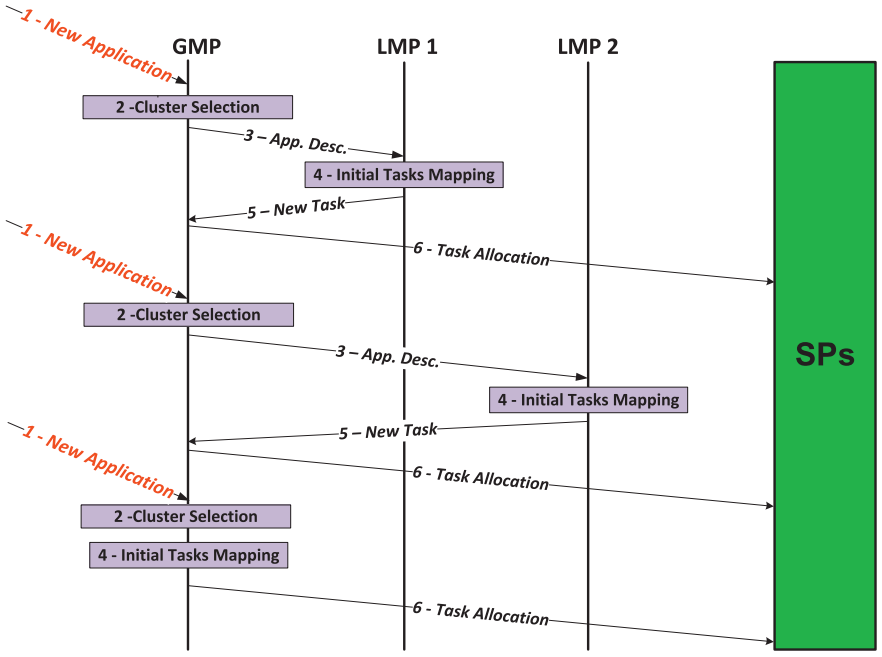


Fig. 5. Cluster selection and initial task mapping protocol.

6. Hierarchical task mapping

The mapping of the set of tasks $T=\{t_1, t_2,..., t_n\}$ of GApp onto the set $SP=\{sp_1, sp_2,..., sp_k\}$ of GMPSoC is defined by the mapping function: $T \rightarrow SP$, where $\forall t_i \in T, \exists\ sp_j \in SP$. The hierarchical task mapping is divided into three main steps. (1) *cluster selection*, define a cluster to map a required application; (2) *initial task mapping*, select SPs to map the application initial tasks inside the cluster; (3) *non-initial tasks mapping*, select SPs to map the non-initial tasks.

The GMP receives from the external world requisitions to execute new applications in the system ('1 – New application', Fig. 5). The GMP verifies if the system has available resources to map the application. If there are no available resources, the application is scheduled to be mapped later. Otherwise, the GMP selects a cluster to map the required application ('2 – Cluster Selection', Fig. 5). The heuristic to select a cluster is presented in Section 6.1.1. Once a given cluster is selected, the GMP obtains the *application description* (Section 4) from the application repository, transmitting it to the selected cluster LMP ('3 – App. Desc.', Fig. 5). The LMP of the selected cluster receives and stores the application description. Then, such LMP verifies the application description to obtain the initial tasks of the application. Next, the LMP map the initial tasks inside the cluster ('4 – Initial Tasks Mapping', Fig. 5). The mapping of initial tasks starts the application execution. Section 6.1.2 presents the heuristic to map the initial tasks. After selecting an SP to receive an initial task, the LMP sends a message to the GMP with the service *task allocation request* ('5 – NewTask', Fig. 5). Such message requests the allocation of the initial task object code in the selected SP. This happens since the GMP is the only PE with access to the application repository. Then, the GMP obtains the task object code from the application repository and transmits it to the selected SP ('6 – Task Allocation', Fig. 5). The SP will schedule the new task at the end of the "*task allocation*" packet reception. Also, the LMP keeps a data structure, named *task table*, with the address of all mapped tasks in the cluster.

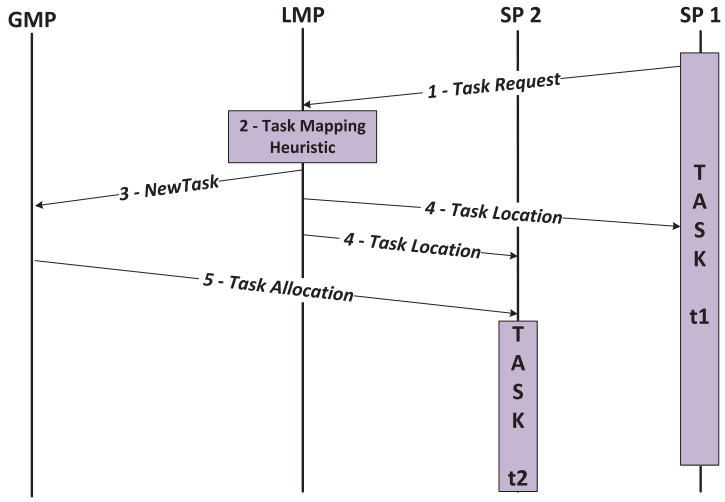


Fig. 6. Non-initial task mapping protocol.

Consider in Fig. 5 the third application insertion. This situation illustrates a scenario where the selected cluster is the one managed by the GMP itself. In this case, the GMP also executes the initial task mapping algorithm.

As explained before, the mapping of non-initial tasks occurs whenever a given task t_i needs to communicate with a non-mapped task t_j. Suppose the example of Fig. 6, where task t_1, mapped on SP_1, needs to communicate with a non-mapped task t_2. In this case, task t_1 requests the mapping of t_2 to its cluster LMP by sending a Task Request packet message ('1 – Task Request', Fig. 6). The LMP receives the task request and executes a mapping heuristic to select an SP to map task t_2 ('2 – Task Mapping Heuristic, Fig. 6). The mapping algorithm, described in Section 6.1.3, selects SP_2 to map task t_2. Next, the LMP request the mapping of task t_2 on SP_2 to the GMP by sending a "Task Allocation Request" service packet ('3 – NewTask' Fig. 6). The LMP also uses a "Task Location" service packet to inform to SP_1 the location of t_2, and to SP_2 the location of task t_1 ('4 – Task Location', Fig. 6). These locations are stored in the SPs task tables. Finally, the GMP obtains task t_2 object code from the application repository and transmits it to SP_2 ('5 – Task Allocation', Fig. 6).

Algorithm 1
Cluster selection heuristic, executed in the GMP.

```
Input: application size app.size
Output: selected_cluster
1.  selected_cluster ← −1
2.  selected_cluster_energy ← +∞
3.  if available_resources(system) >= APP.size then
4.    for each cluster c_k in the system
5.      if available_resources(c_k) >= APP.size and cl_energy (c_k)<
          selected_cluster_energy then
6.        selected_cluster ← c_k
7.        selected_cluster_energy ← cl_energy (c_k)
8.      end if
9.    end for
10.   if selected_cluster=−1 then
11.     for each cluster c_k in the system
12.       if cl_energy (c_k)< selected_cluster_energy then
13.         selected_cluster ← c_k
14.         selected_cluster_energy ← cl_energy (c_k)
15.       end if
16        end for
17    end if
18.   end if
19. return selected_cluster
```

6.1. "Heat" mapping heuristic

This section describes the proposed HEAT (**H**ierarchical **E**nergy-**A**ware **T**ask) mapping heuristic. This heuristic makes a trade-off between workload distribution (processor and router energy) and communication volume reduction. The heuristic uses the following definitions:

- **Definition 1**: *application size* (app.size) corresponds to the number of tasks of the application to be mapped.
- **Definition 2**: MAX_SP_TASKS is the number of tasks a given SP may execute simultaneously (SP_PAGES - 1).
- **Definition 3**: *available_resources* corresponds to the number of *resources* (a resource is a page in the memory) that do not have a task mapped on it. This information may refer to the whole system, *available_resources*(*system*), or to a given cluster c_k, *available_resources*(c_k).
- **Definition 4**: *available*(sp_i) returns *true* if sp_i is available to receive a new task, otherwise *false*. An SP is available when the number of tasks mapped on it is smaller than MAX_SP_TASKS.
- **Definition 5**: *empty SP* is an SP with no tasks mapped on it. Therefore, an empty SP can receive MAX_SP_TASKS tasks.
- **Definition 6:** *TE* is the total consumed energy by a given SP, corresponding to the energy (E_i) consumed by all already executed tasks and the tasks that are currently being executed on this processor. The router energy consumption is also accounted in the TE value. Monitoring packets transmits the *TE* value of each SP to the corresponding LMP.

6.1.1. Cluster selection

This heuristic computes the consumed energy of each cluster c_k, *cl_energy*(c_k), using data sent by the monitoring packets. Then, the cluster with the smallest *cl_energy*(c_k) is selected. This procedure avoids mapping an application in a high overloaded cluster, which improves the workload distribution. Algorithm 1 presents the pseudo-code of the cluster selection heuristic.

The heuristic in Algorithm 1 first verifies if the system has available resources to map the application (line 3). If there are no sufficient resources in the system, the application is scheduled to be mapped later. The first loop (lines 4–9) analyzes all clusters that have available resources to map the application, selecting the one with the smallest accumulated energy. If there are no clusters with available resources to map the application, a cluster with the smallest accumulated energy is selected, regardless the number

			123			
		66	178	280		
	114	200	80	109	77	
120	210	120	200	110	350	327
	124	156	85	413	95	
		149	123	189		
			102			

Fig. 7. Hypothetical example of *region_energy*.

Algorithm 2
First phase of the initial tasks mapping, executed in the LMPs.

```
Input: n_hops
Output: selected_sp
1.  selected_sp ← -1
2.  selected_region_energy ← +∞
3.  for each SP sp_i in the cluster
4.    if available(sp_i) and region_energy(sp_i, n_hops)<
        selected_region_energy then
5.      selected_sp ← sp_i
6.      selected_region_energy ← region_energy(sp_i, n_hops)
7.    end if
8.    end for each
9. return selected_sp
```

Algorithm 3
Second phase of the initial tasks mapping, executed in the LMPs.

```
Input: SP_address, n_hops   // SP_address is the selected_sp address obtained
    in the 1st phase
Output: selected_sp
1.  selected_sp ← −1
2.  selected_sp_energy ← +∞
3.  // Get all neighbors of selected_sp within a distance n_hops
4.  neighbors_list ← neighbors(SP_address, n_hops)
5.  while all SPs in the cluster not evaluated and selected_sp=−1 do
6.    for each SP sp_i in neighbors_list
7.      if available(sp_i)=true and TE(sp_i) < selected_sp_energy then
8.        selected_sp ← sp_i
9.        selected_sp_energy ← TE (sp_i)
10.     end if
11.   end for
12.   if selected_sp=−1 then
13.     n_hops ← n_hops +1
14.     neighbors_list ← neighbors(SP_address, n_hops)
15.   end if
16. end while
17. return selected_sp
```

of available resources (lines 11–16). Note that the application is mapped in the MPSoC *iff* the system has available resources for the application.

This heuristic aims to distribute the energy homogeneously when a new application arrives in the system. In the long-term, this procedure avoids hotspots, and processors stressed over the time.

6.1.2. Initial tasks mapping

The initial tasks mapping heuristic searches a region with smallest consumed energy in the cluster. The search space is limited by the parameter *n_hops*, obtained from sqrt(|$PE_{cluster}$|/2), where |$PE_{cluster}$| is the number of PEs in the cluster. The reasoning of this procedure is to map communicating tasks near to each other, in a set of PEs with the smallest accumulated energy.

This heuristic divides the initial task process into two phases. The first phase selects an SP with the smallest *region_energy* to receive an initial task. A second phase is executed when the application has more than one initial task. In such phase, it is created a set with all SPs up to *n* hops from the selected SP, selecting the SP of this set with the smallest TE (Definition 6).

The function *region_energy(sp_i, n_hops)* returns the average TE from the set containing sp_i and all SPs up to *n_hops* hops from sp_i. Fig. 7 shows a hypothetical example using a 7×7 cluster, where sp_i is the central SP $sp_{central}$ (in green); and *n_hops* is 3 hops. In Fig. 7, the numbers inside each rectangle represent the TE of each SP. The value of *region_energy($sp_{central}$, 3)* corresponds to 64, since: (*i*) inside a region 3 hops far from $sp_{central}$ there are 25 SPs; (*ii*) the sum of the TEs of the SPs in this area is equal to 4100; (*iii*) the average TE in this area is equal to $4100/25 = 64$.

Suppose a hypothetical example of an application with two initial tasks: t_i and t_j. The first initial task t_i is mapped in $sp_{central}$ of Fig. 7. For the mapping of the t_j is defined a region 3 hops from $sp_{central1}$, as delimited by the numbered SPs in Fig. 7. Then, the SP with the smallest TE in this region is selected to map t_j. In the example, such SP has TE equal to 66.

The pseudo-code of the first phase of the initial tasks mapping heuristic is detailed in Algorithm 2. The main loop (lines 3–8) selects an SP (*selected_sp*) with the lowest *region_energy*. This procedure ensures that application's tasks that will be mapped later will be assigned closer to the selected SP and in SPs with a lower accumulated energy.

If the application has only one initial task, the SP chosen by the heuristic of Algorithm 2 is selected to execute the task. Otherwise, the heuristic presented in Algorithm 3 is executed for each non-mapped initial task. In line 4 it is created a set *neighbors_list* with all SPs up to *n_hops* from *selected_sp* computed in the previous phase. The loop between lines 6-11 selects an available SP from the *neighbors_list* with the smallest TE. If there is no available SP inside the list, the search space increases 1 hop (lines 12-15), until visiting all SPs of the cluster (line 5).

6.1.3. Non-initial task mapping

Suppose a non-initial task t_i is required to be mapped. The HEAT heuristic evaluates the set $C(t_i)$, and creates a bounding box containing all t_i communicating tasks mapped within the cluster. Then, such bounding box is increased in one hop offering a large search space. The cluster boundaries limit the search space. Fig. 8 illustrates the mapping search space in the cluster. This heuristic selects the SP inside the bounding box with the lowest TE. This heuristic makes a trade-off between workload balancing and communication volume reduction. The heuristic selects the SP inside the bounding box with the lowest TE.

Algorithm 4 describes the algorithm used to select an SP to receive a non-initial task t_i. The heuristic creates a list with all tasks communicating with t_i already mapped onto the SPs of the cluster (line 3). In the sequel, it is defined a bounding box rectangle (line 4), with all mapped communicating tasks. This bounding box is increased by one hop (line 5), offering a larger search space to map t_i. A list with candidate SPs is created (line 7). The available SP in the list with the smallest TE is selected (lines 8–13). If no SP can be selected, the bounding box is increased by one hop (lines 14–16). This process continues up to find an SP or to visit all SPs of the cluster.

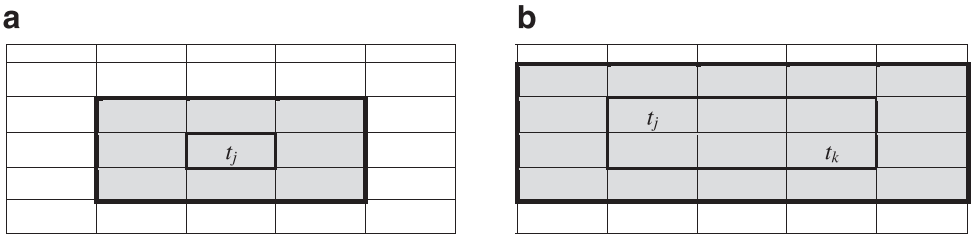


Fig. 8. Non-initial task mapping search space. (a) search space when one communicating tasks is already mapped (t_i). (b) search space when more than one communicating task is already mapped (t_i and t_k).

Algorithm 4
Mapping of non-initial tasks, executed in the LMPs.

```
Input: t_i, set C(t_i)
Output: selected_sp
1.  selected_sp ← −1
2.  selected_sp_energy ← +∞
3.  MC(t_i)← mapped_tasks(C(t_i))  // all tasks communicating with t_i
    already mapped
4.  bounding_box ← area(MC(t_i))
5.  increase(bounding_box, 1)
6.  while all SPs in the cluster were not evaluated and selected_sp=−1
    do
7.    neighbors_list ← search_SPs(bounding_box)
8.    for each SP sp_i in neighbors_list
9.      if available(sp_i)=true and TE(sp_i)< selected_sp_energy then
10.       selected_sp ← sp_i
11.       selected_sp_energy ← TE(sp_i)
12.     end if
13.   end for
14.   if selected_sp=−1 then
15.     increase(bounding_box, 1)
16.   end if
17. end while
18. return selected_sp
```

Algorithms 3 and 4 may return –1, meaning that the cluster has no available SP to receive the task. In this situation, the μkernel borrows an SP from a neighbor cluster (process named *reclustering*), mapping the task in the borrowed SP.

7. Results

The experiments were executed in the reference MPSoC, using a clock cycle accurate model described in SystemC. Each SP can execute up to 2 simultaneous tasks, scheduled by the μkernel. The main cost function of the proposed mapping heuristic, *HEAT*, is the energy distribution, as previously discussed.

The reference mapping heuristic is the LEC-DN [30]. The LEC-DN heuristic considers the dependencies between all communicating tasks, using as the main cost function the minimization of the communication energy in the NoC. To minimize the communication energy, this heuristic uses the communication volume between tasks since the number of transmitted flits defines the communication energy. This heuristic is selected as the reference since its cost function is the one adopted in most NoC-based systems: minimize the communication energy.

Chantem et al. [9] use as part of their heuristic the largest task first (LTF) algorithm to slow down the wear process on the cores as much as possible. LTF is an energy-aware heuristic that attempts to balance spatially the system load in a non-increasing order of energy consumption and assign them to the core with the least total energy consumption. Once a task is assigned to a core, the core total energy consumption is updated. This heuristic does not divide the system into clusters, and the whole application is mapped at the moment it is required. LTF is also compared against the proposed heuristic, but not used as the reference because it is centralized and not consider in its cost function the communication energy.

Five benchmarks, described in C language, are used: (*i*) DTW - Digital Time Warping (DTW), with 10 tasks; (*ii*) MPEG decoder, with 5 tasks; (*iii*) DJK - Dijkstra, with 6 tasks; (*iv*) SYN1, synthetic application, with 12 tasks, which emulates the communication behavior of an MPEG4 full decoder; (*v*) SYN2, synthetic application, with 12 tasks, that emulates the communication behavior of VOP (Video Object Plane) decoder application.

Experiments are conducted using the scenarios presented in Table 2. Scenarios 1–5 correspond to a many-core system with 64 PEs, executing a large number of tasks – from 250 to 1,000. Scenarios 1 and 2 contain a mix of applications while scenarios 3–5 have identical applications. Scenarios with identical applications are expected to generate mapping solutions with a balanced workload distribution. Scenarios 6 and 7 contain 256 PEs. The goal of these scenarios is to present the effectiveness of the proposed approach for large systems. The last column of Table 2 corresponds to the average number of tasks per SP. Scenarios with larger values in this column correspond to heavier workloads, favoring the proposed heuristic to produce a better workload distribution along the time.

7.1. Monitoring period evaluation

Table 3 evaluates the consumed energy at each cluster, varying the monitoring period. With a small intra-cluster monitoring period, the number of monitoring packets increases, overloading the LMP. In such a case, several monitoring packets are delayed, and the LMP takes decisions with current and past data (i.e. some SPs were not updated since the monitoring packets were not treated), leading to wrong mapping decisions. On the other side, with large monitoring periods, SPs may receive new tasks since the energy consumption was not yet updated. With an intermediate monitoring period, all monitoring packets are received and treated, without incurring in the long updating problem induced by long monitoring periods. Observe the DIFF row, which corresponds to the difference between the maximum and minimum consumption between clusters. The monitoring periods 1 ms/3 ms lead to the better load distribution among the clusters.

Table 4 evaluates different performance parameters for different monitoring periods. Scenario 1 was selected because it has a set of different applications, and an important workload to execute (780 tasks). The results in this Table shows:

- *Workload distribution* (lines 1–3). The energy standard deviation between SPs drops from 119 to 31 µJ, while the maximum energy consumption drops from 432 to 234 µJ. Also, using LEC-DN several processors do not execute user tasks (*min SP consumption* line) while in the proposed heuristic all SPs execute user tasks.
- *Execution time* (line 4). Small reduction. Next section discusses this result, evaluating all scenarios.

Table 2
Characteristics of the evaluated scenarios.

Scenario	MPSoC Size	Cluster Size	Applications	Total number of tasks	Number of tasks per SP
1	8 × 8 (60 SPs)	4 × 4	20 × MPEG, 20 × DJK, 20 × SYN1, 20 × SYN2, 20 × DTW	780	13
2			10 × MPEG, 10 × DJK, 10 × SYN1, 10 × SYN2, 10 × DTW	390	6.5
3			50 × MPEG	250	4.17
4			100 × DTW	1000	16.67
5			100 × MPEG	500	8.33
6	16 × 16 (240 SPs)	4 × 4	20 × MPEG, 20 × DJK, 20 × SYN1, 20 × SYN2, 20 × DTW	780	3.25
7			40 × MPEG, 40 × DJK, 40 × SYN1, 40 × SYN2, 40 × DTW	1560	6.5

Table 3
Evaluation of the monitoring period, for scenario 1. TE: total energy consumed in the cluster (μJ). STDEV: standard deviation related to the consumed energy by the SPs in the cluster (μJ). DIFF: difference between the maximum and minimum consumption between clusters.

			HEAT - Monitoring period varying the intra/inter periods									
	LEC-DN		0.25 ms / 3 ms		0.5 ms / 3 ms		1 ms / 3 ms		2 ms / 3 ms		4 ms / 8 ms	
	TE	STDEV	TE	STDEV	TE	STDEV	TE	STDEV	TE	STDEV	TE	STDEV
CL 0	2086	130	4247	46	3818	51	**2607**	**30**	2609	56	2567	34
CL 1	2245	114	2512	28	2215	31	**2479**	**22**	2196	31	2412	22
CL 2	2508	99	2408	37	2434	33	**2433**	**36**	2541	40	2788	31
CL 3	2470	127	1676	26	2083	15	**2476**	**33**	2390	27	2592	42
DIFF	422		2571		1735		**174**		413		376	

Table 4
Evaluation of the monitoring period, for scenario 1, considering the total system energy, standard deviation between SPs and clusters, maximum and minimum energy consumption by SPs, and the execution time.

	LEC-DN	HEAT - Monitoring period varying the intra/inter periods				
		0.25 ms / 3 ms	0.5 ms / 3 ms	1 ms / 3 ms	2 ms / 3 ms	4 ms / 8 ms
STDEV all SPs (μJ)	119	72	58	**31**	41	34
Max SP consumption (μJ)	432	390	372	**234**	269	249
Min SP consumption (μJ)	0.33	66	98	**111**	88	68
Execution time (ms)	243	260	234	**234**	240	233
Total System Energy (μJ)	9310	10842	10549	**9996**	9736	10358
N# of flits (10^6)	10.443	18.815	16.655	**15.666**	15.539	14.887

- *Energy consumption* (line 5). Increases, because more SPs execute user task. Next section discusses this result, evaluating all scenarios.
- *NoC traffic* (line 6). Increases, because the proposed heuristic reduces the CPU sharing to improve the workload distribution. Next section discusses this result, evaluating all scenarios.

The current work adopts 1 and 3 ms as the intra- and inter-cluster monitoring periods respectively. These values are adopted because they present the best tradeoff between workload distribution and energy consumption.

7.2. Workload distribution

Fig. 9 presents the workload distribution for scenario 1 (similar results are observed for the other scenarios), where each rectangle contains the total energy consumed by each SP (processor and router). The manager PEs are not included in the result because they do not execute user applications. As illustrated in Fig. 9(a), the LEC-DN produces an unbalanced workload distribution with several "hot" processors, spending more than 300 µJ. The "hot" processors are placed in the center of the clusters, in such a way to reduce the distance between communicating tasks, and hence minimize the communication energy. On the other side, the HEAT mapping (Fig. 9(b)) produces a uniform energy distribution.

Fig. 10 presents the workload distribution histograms for scenarios 1 and 7, considering the number of SPs per energy interval. From the first histogram, Fig. 10(a), it is possible to observe the non uniform load distribution produced by the heuristic that minimizes only the communication energy – LEC-DN. For scenario 1, 23 SPs consume less than 100 µJ, 15 SPs consume more than 240 µJ, and 22 SPs consume in the interval 100–240 µJ. The proposed HEAT heuristic has all 60 SPs consuming between 100 and 240 µJ, showing its ability to distribute the workload along the time. A similar distribution is observed for scenario 7.

Table 5 evaluates all scenarios, with summarized results. Fig. 11 plots results normalized to LEC-DN. The results in this Table shows:

- *Average consumed energy per SP*. Considering that the workload applied for both mapping heuristics is the same for each scenario, a small variation is expected. Excepting scenario 4 (it executes a computation intensive application – DTW), the proposed HEAT heuristic increases the average number of executed instructions by 8.7%. This is explained by the fact more processors are assigned to execute tasks, leading to additional μkernel instructions execution. When a given processor is not executing any task, it enters in a hold state, dissipating only static power.
- *Total system energy*: this column considers the energy consumed by the processors and the routers. As the number of executed instructions increased, the proposed HEAT heuristic increased the consumed energy in average by 4.4% (worst-case: 12.3%, scenario 7). Note that the total energy consumption does not increase in the same proportion to the SPs because the static energy is accounted.
- *Workload distribution* (column STDDEV). This is the *main cost function* of the HEAT mapping. All scenarios presented expressive improvement in the workload distribution. As mentioned in the experimental setup, scenarios with identical applications (3–5) present the smaller standard deviation values. A smaller

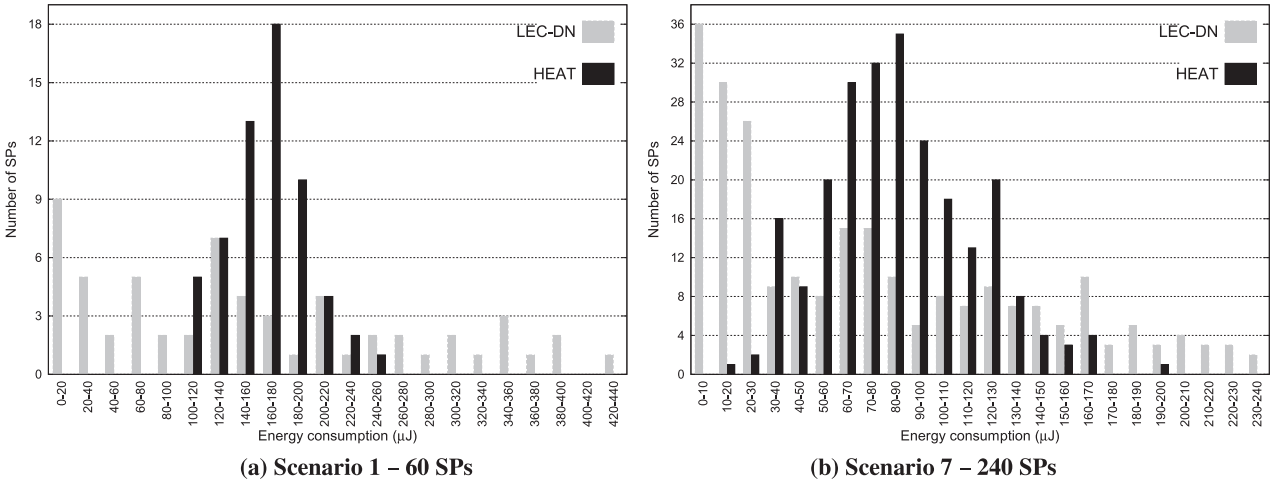
(a) LEC-DN mapping

31	184	206	133	14	242	150	29
138	**376**	**347**	255	261	**399**	**391**	146
32	205	168	167	17	**327**	171	114
LMP	39	132	98	LMP	79	75	56
0	6	203	144	8	235	91	19
125	**432**	298	73	140	**350**	**343**	142
0	260	313	61	18	**314**	213	133
GMP	112	59	0	LMP	35	133	71

(b) HEAT mapping

122	196	212	163	135	162	141	165
228	147	177	153	212	216	128	131
116	111	143	203	155	130	234	159
LMP	185	113	164	LMP	140	176	192
197	148	198	142	134	153	153	181
193	190	138	123	130	165	177	153
225	171	150	224	159	206	194	158
GMP	185	163	160	LMP	156	157	203

Fig. 9. Workload distribution for scenario 1. Each rectangle is an SP, with the consumed energy in μJ.



(a) Scenario 1 – 60 SPs **(b) Scenario 7 – 240 SPs**

Fig. 10. Histogram related to the energy distribution for scenarios 1 and 7 (*x*-axis: energy interval, *y*-axis: number of SPs for each interval).

Table 5
Evaluation of the 5 scenarios, considering the monitoring periods equal to 1 ms/3 ms.

Scenario	Avg. consumed energy per SP (μJ)		Total System Energy (μJ)		STDEV Energy - all SPs (μJ)		MAX Energy - all SPs (μJ)		Execution time (ms)		N# of flits (10^6)	
	LEC-DN	HEAT	LEC-DN	HEAT	LEC-DN	HEAT	LEC-DN	HEAT	LEC-DN	HEAT	LEC-DN	HEAT
1	155	167	11922	12412	119	31	432	234	243	234	10.443	15.666
2	77	83	6036	6444	63	29	217	158	130	133	5.330	8.023
3	37	39	3007	2975	43	17	152	78	68	59	2.159	2.870
4	66	64	4523	4414	34	10	101	84	65	64	4.473	5.135
5	73	81	5921	6064	79	21	304	130	134	115	4.259	5.797
6	36	41	11816	1281	36	25	141	126	69	66	12.979	17.708
7	73	86	23711	26622	64	31	238	192	134	139	26.366	36.843
HEAT/LEC-DN:		**+8.7%**		**+4.4%**		**−59.2%**		**−32.2%**		**−4.4%**		**+37.2**

reduction is observed in scenario 6 because the load applied to it is lighter (smaller number of tasks per PE as shown in the last column of Table 2).

- *Maximum energy*. This result is a parameter related to the system reliability. The average reduction of the maximum consumption per SP is 32.2% (best-case: −57.2%, scenario 5).
- *Execution time*. Even if the goal is not to reduce the execution time, the average reduction in the execution time is 4.5%. This result is explained by the fact that more processors execute tasks, reducing the processor sharing induced by the LEC-DC heuristic.
- *Traffic in the NoC* (column *N# of flits*). This column measures the number of flits (10^6) transferred through the NoC. As expected, LEC-DN, reduces the traffic in the NoC because the communication energy is the main goal of this heuristic. The proposed HEAT heuristic increased the number of transferred flints in average by 37.2% (worst case: 50.5%, scenario 2).

The column "all SPs STDDEV" of Table 5 reflects the cost function of the proposed heuristic: workload distribution. The energy is evenly distributed in the systems, with an important reduction in the number of hotspots, as shown in Fig. 9(b) and column "all

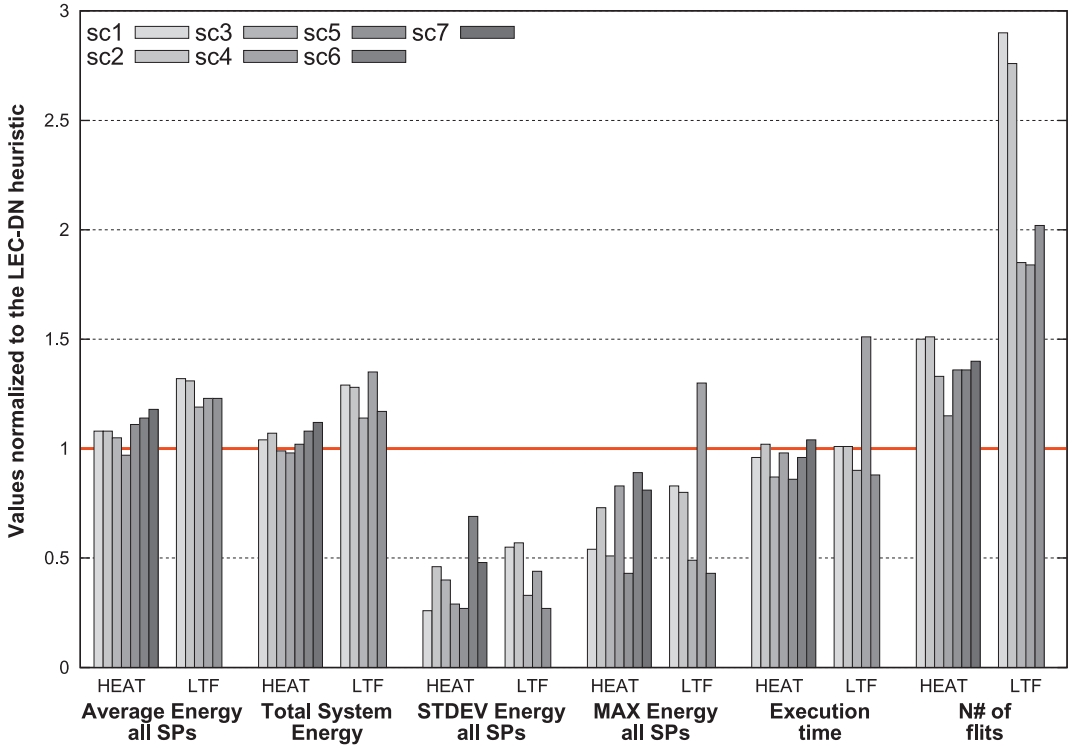


Fig. 11. Comparison of the proposed HEAT (scenario 1 to 7) and LTF (scenario 1–5) heuristics, normalized to the LEC-DN heuristic.

SPs MAX". The column "N# of flits" reflects the traditional cost function of mapping heuristic: reduction of the NoC traffic. Even if the communication energy is reduced, processors are overload, compromising in the long term the system reliability.

Finally, Fig. 11 compares the proposed HEAT and LTF heuristics (both heuristics use as cost function the energy consumption as main metric), normalized to the LEC-DN mapping. The behavior of the proposed HEAT heuristic was previously discussed, using as reference Table 5. The LTF heuristic presents a *similar trend*: higher energy consumption (up to 38%), better workload distribution (*all SPs STDDEV)*, similar execution time (excepting scenario 4), and a larger number of flits transmitted in the NoC.

The LTF heuristic presents worse results than the HEAT heuristic for two main reasons. The first one is related to its centralized approach: one single PE to make mapping decisions (this explains why scenarios 6 and 7 for LTF are not presented in Fig. 11). The second issue is the fact the *only* energy is considered to take mapping decisions. The number of hops between communicating tasks increases, leading to an excessive increase in the number of flits transferred through the NoC (almost 3 times). Note that LTF in scenario 4 increased the maximum SP utilization and the execution time (51%). This scenario has a computation intensive benchmark, resulting in tasks from different applications sharing the same PE, increasing the execution time.

8. Conclusion and future works

The features included in the HEAT mapping include scalability, runtime execution, workload distribution. The hierarchical management of the mapping approach, which comprises three steps, ensures scalability. The workload distribution is ensured by the energy monitoring approach, which guides the mapper to select the processors less used.

The proposed HEAT mapping achieved a better workload distribution, with minimal impact to energy consumption, and reduction in maximum processor energy. The NoC usage increases, being an expected result because the application tasks use more processors to execute the same job. An important feature of the proposal is its distributed nature, using several manager processors to map the tasks. Comparing our approach to a centralized approach, with a similar cost function, we observed that a centralized approach increases the total consumed energy and spread the tasks, increasing the NoC traffic. Consequently, this works enforces important features to consider in mapping heuristics: hierarchy, monitoring and multi-objective cost function (in our proposal accumulated energy and distance among communicating tasks).

Future works include to: (1) integrate of a lifetime model to evaluate MTTF; (2) include a temperature model to guide the mapping; (3) extend the mapping heuristic to cope with power constraints (i.e. limit the usage of processors according to a power budget assigned to the system); (4) couple the approach to a DVFS approach acting over PEs when a given power constraint is violated.

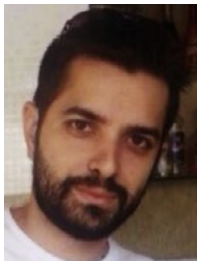
Acknowledgments

The Author Fernando Moraes is supported by CNPq - projects 472126/2013-0 and 302625/2012-7, and FAPERGS - project 2242-2551/14-8.

References

[1] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, IEEE Comput. 35 (1) (January, 2002) 70–78.
[2] A. Singh, et al., Mapping on multi/many-core systems: survey of current and emerging trends, in: DAC, 2013, p. 10.
[3] A.K. Singh, et al., Communication-aware heuristics for runtime task mapping on NoC-based MPSoC platforms, J. Syst. Archit.: EUROMICRO J. 56-7 (Jul 2010) 242–255.
[4] C-L. Chou, R. Marculescu, Runtime task allocation considering user behavior in embedded multiprocessor networks-on-chip, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 29 (1) (2010) 78–91.
[5] Intel, The Intel® Xeon Phi™ Coprocessor, Intel, 2012.
[6] Tilera Corporation, Tile-GX Processor Family, Tilera Corporation, 2010.
[7] International Technology Roadmap for Semiconductors. Accessed in: http://www.itrs.net/reports.html. February 2013.

[8] M.A. Faruque, et al., ADAM: runtime agent-based distributed application mapping for on-chip communication, in: DAC, 2008, pp. 760–765.
[9] T. Chantem, et al., Enhancing multicore reliability through wear compensation in online assignment and scheduling, in: DATE, 2013, pp. 1373–1378.
[10] Z. Wang, et al., System-level reliability exploration framework for heterogeneous MPSoC, in: GLSVLSI, 2014, pp. 9–14.
[11] J. Henkel, et al., Reliable on-chip systems in the nano-era: lessons learnt and future trends, in: DAC, 2013, pp. 1–10.
[12] Meyer, B.; et al. "Cost-effective lifetime and yield optimization for NoC-based MPSoCs". In: ACM Transactions on Design Automation Electronic Systems, vol. 19(2), 2014
[13] D. Kramer, W. Karl, A scalable monitoring infrastructure for self-organizing many-core architectures, in: DSD, 2012, pp. 42–49.
[14] M.G. Mandelli, L.C. Ost, A.M. Amory, F.G. Moraes, Multi-task dynamic mapping onto NoC-based MPSoCs, in: SBCCI, 2011, pp. 191–196.
[15] L.C. Ost, M.G. Mandelli, G.M. Almeida, L.S. Moller, L.S. Indrusiak, G. Sassatelli, P. Benoit, M. Glesner, M. Robert, F.G. Moraes, Power-aware dynamic mapping heuristics for NoC-based MPSoCs using a unified model-based approach, ACM Trans. Embed. Comput. Syst. 12 (3) (2013) 1–22.
[16] L.T. Smit, J.L. Hurink, G.J.M. Smit, Runtime mapping of applications to a heterogeneous SoC, in: SoC, 2005, pp. 78–81.
[17] A. Ngouanga, G. Sassatelli, L. Torres, T. Gil, A. Soares, A. Susin, A contextual re-sources use: a proof of concept through the APACHES platform, in: DDECS, 2006, pp. 42–47.
[18] A.K. Coskun, et al., Dynamic thermal management in 3D multicore architectures, in: DATE, 2009, pp. 1410–1415.
[19] P.K.F. Hölzenspies, J.L. Hurink, J. Kuper, G.J.M. Smit, Runtime spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSOC), in: DATE, 2008, pp. 212–217.
[20] S. Wildermann, T. Ziermann, J. Teich, Run time mapping of adaptive applications onto homogeneous NoC-based reconfigurable architectures, in: FPT, 2009, pp. 514–517.
[21] A. Schranzhofer, J.-J. Chen, L. Thiele, Dynamic power-aware mapping of applications onto heterogeneous MPSoC platforms, IEEE Trans. Ind. Inf. 6 (4) (2010) 692–707.
[22] S. Lu, C. Lu, P. Hsiung, Congestion- and energy-aware runtime mapping for tile-based network-on-chip architecture, in: Proceedings of Frontier Computing. Theory, Technologies and Applications, 2010, pp. 300–305.
[23] E. Carvalho, N. Calazans, F. Moraes, Dynamic task mapping for MPSoCs, IEEE Des. Test Comput. 27-5 (Oct 2010) 26–35 Set.
[24] A.K. Singh, Efficient heuristics for minimizing communication overhead in NoC-based heterogeneous MPSoC platforms, in: RSP, 2009, pp. 55–60.
[25] S. Kobbe, L. Bauer, D. Lohmann, W. Schroder-Preikschat, J. Henkel, DistRM: distributed resource management for on-chip many-core systems, in: CODES+ISSS, 2011, pp. 119–128.
[26] Y Cui, W Zhang, H Yu, Decentralized agent based re-clustering for task mapping of tera-scale network-on-chip system, in: ISCAS, 2012, pp. 2437–2440.
[27] A. Hartman, et al., Lifetime improvement through runtime wear-based task mapping, in: CODES+ISSS, 2012, pp. 13–22.
[28] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, B. Veeravalli, Runtime mapping for reliable many-cores based on energy/performance trade-offs, in: DFT, 2013, pp. 58–64.
[29] A. Das, et al., Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs, in: DATE, 2014, pp. 1–6.
[30] M. Mandelli, L. Ost, G. Sassatelli, F. Moraes, Trading-off system load and communication in mapping heuristics for improving NoC-based MPSoCs reliability, in: ISQED, 2015, pp. 392–396.
[31] L. Huang, et al., Lifetime reliability-aware task allocation and scheduling for MPSoC platforms, in: DATE, 2009, pp. 51–56.
[32] Y. Ge, et al., Distributed task migration for thermal management in many-core systems, in: DAC, 2010, pp. 579–584.
[33] Y.-K. Wu, et al., Distributed thermal management for embedded heterogeneous MPSoCs with dedicated hardware accelerators, in: ICCD, 2011, pp. 183–189.
[34] Z. Liu, et al., Task migrations for distributed thermal management considering transient effects, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 23 (2) (2015) 397–401.
[35] R. Jejurikar, C. Pereira, R. Gupta, Leakage aware dynamic voltage scaling for real-time embedded systems, in: DAC, 2004, pp. 275–280.
[36] F. Rosa, L. Ost, T. Raupp, F. Moraes, R. Reis, Fast energy evaluation of embedded applications for many-core systems, in: PATMOS, 2014, pp. 1–6.
[37] A. Martins, D. Silva, G. Castilhos, T. Monteiro, F. Moraes, A method for NoC-based MPSoC energy consumption estimation, in: ICECS, 2014, pp. 427–430.
[38] Y. Kao, M. Yang, S. Artan, H. Chao, CNoC: high-radix clos network-on-chip, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 30 (12) (2011) 1897–1910.
[39] C Villavieja, Y Etsion, A. Ramirez, N. Navarro, FELI: HW/SW support for on-chip distributed shared memory in multicores, in: Euro-Par, 2011, pp. 282–294.
[40] G. Castilhos, M. Mandelli, G. Madalozzo, F. Moraes, Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes, in: ISVLSI, 2013, pp. 153–158.



Guilherme Castilhos received the M.Sc. degree (2012) in Computer Science from the Pontifical Catholic University of Rio Grande do Sul (PUCRS). He is currently a PhD student at the same University, and an associate professor at UNISC (Universidade de Santa Cruz do Sul). His main research interests include Multiprocessor Systems on Chip (MPSoC), power management techniques, and networks on chip networks (NoCs).

Marcelo Grandi Mandelli received the M.Sc. degree (2011) and Ph.D. degree (2015) in Computer Science from the Pontifical Catholic University of Rio Grande do Sul (PUCRS). He is currently an associate professor at UNISC (Universidade de Santa Cruz do Sul). From 2013 to 2014, he made a PHD internship at LIRMM laboratory (Montpellier, France). His main research interests include Multiprocessor Systems on Chip (MPSoC), electronic system level design (ESL), and networks on chip networks (NoCs).

Luciano Ost is currently assistant professor at the University of Leicester. Dr. Ost received his Ph.D. degree in computer science from PUCRS, Brazil in 2010. During his PhD, Dr. Ost worked as invited researcher at the Microelectronic Systems Institute of the Technische Universitaet Darmstadt. After the completion of his doctorate degree, he worked as a research assistant and then as assistant professor at the University of Montpellier in France, until joining the University of Leicester. His main research interests include adaptive and reliable multi/many-core embedded systems.

Fernando Gehm Moraes received the Electrical Engineering and M.Sc. degrees from the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1987 and 1990, respectively. In 1994 he received the Ph.D. degree from the Laboratoire d'Informatique, Robotique et Microélectronique de Montpellier (LIRMM), France. He is currently at PUCRS, where he has been an Associate Professor from 1996 to 2002, and Full Professor since 2002. From 1998 to 2000 he joined the LIRMM as an Invited Professor for 3 months each year. He has authored and co-authored 25 peer refereed journal articles in the field of VLSI design, comprising the development of networks on chip and telecommunication circuits. One of these articles, "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip", is cited by more than 500 other papers. He has also authored and co-authored more than 200 conference papers on these topics. He has advised 24 M.Sc. and 6 Ph.D. works. His primary research interests include Microelectronics, FPGAs, reconfigurable architectures, NoCs (networks on chip) and MPSoCs (multiprocessor system on a chip). SBC, SBMICRO and IEEE Senior Member.