Journal of Systems Architecture 56 (2010) 242–255

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms ☆

Amit Kumar Singh [a,*], Thambipillai Srikanthan [a], Akash Kumar [b,c], Wu Jigang [a]

[a] School of Computer Engineering, Nanyang Technological University, Singapore, 639798
[b] Department of Electrical and Computer Engineering, National University of Singapore, Singapore
[c] Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands

ARTICLE INFO

Article history:
Received 14 October 2009
Received in revised form 26 April 2010
Accepted 27 April 2010
Available online 7 May 2010

Keywords:
Hardware–software co-design
Heterogeneous architectures
MPSoC design
Network-on-Chip (NoC)
Mapping algorithms

ABSTRACT

Efficient run-time mapping of tasks onto Multiprocessor System-on-Chip (MPSoC) is very challenging especially when new tasks of other applications are also required to be supported at run-time. In this paper, we present a number of communication-aware run-time mapping heuristics for the efficient mapping of multiple applications onto an MPSoC platform in which more than one task can be supported by each processing element (PE). The proposed mapping heuristics examine the available resources prior to recommending the adjacent communicating tasks on to the same PE. In addition, the proposed heuristics give priority to the tasks of an application in close proximity so as to further minimize the communication overhead. Our investigations show that the proposed heuristics are capable of alleviating Network-on-Chip (NoC) congestion bottlenecks when compared to existing alternatives. We map tasks of applications onto an 8×8 NoC-based MPSoC to show that our mapping heuristics consistently leads to reduction in the total execution time, energy consumption, average channel load and latency. In particular, we show that energy savings can be up to 44% and average channel load is improved by 10% for some cases.

1. Introduction

Multiprocessor Systems-on-Chip (MPSoCs) are SoCs consisting of multiple processing elements (PEs). MPSoCs are being increasingly used in complex embedded applications in order to meet their ever rising performance requirements as these applications can no longer be supported by a single general purpose processor [17]. Additionally, MPSoCs are the most promising candidate to exploit the rising high level of integration provided by the nanotechnology.

Several MPSoCs have been proposed by academia and industry. Homogeneous MPSoCs [22,38,20], composed of identical PEs are proposed by academia whereas [46,4] by industry. In [46], Intel proposed a homogeneous MPSoC consisting of 80 PEs connected by a NoC where each processor contains two floating-point units. Future MPSoCs are anticipated to contain thousands of PEs in a single die by 2015 because of the advancement in nanometer technology [8]. These MPSoCs can support some applications whereas heterogeneous MPSoCs consisting of different types of PEs can support wider variety of applications. The PEs present in heterogeneous MPSoCs can be instruction set processors (ISPs), specialized PEs like Digital Signal Processors (DSPs), FPGA fabric tiles, dedicated intellectual property cores (IPs) and specialized memories in order to achieve high computation performance along with energy efficiency by exploiting distinct features of different type of PEs to improve the performance. Heterogeneous MPSoCs [30,43] are proposed by academia and [1,19] by industry. In [19], a heterogeneous MPSoC composed by one manager processor and 8 floating-point units is proposed by combined effort of IBM, Sony and Toshiba.

The PEs present in the MPSoCs call for a communication infrastructure, to have proper communication amongst multiple PEs. This communication infrastructure can be bus-based, point-to-point or Networks-on-Chip (NoCs)-based. The NoCs are the future communication infrastructure as they have several advantages over others, such as scalability and shorter wires, which minimizes power consumption [5,14,6].

Mapping applications' tasks onto MPSoC platform can be accomplished at either design-time or run-time. Most of the works in literature present static (design-time) mapping techniques that cover only certain scenarios [22,23,32,9,45]. These mapping techniques find best placement of tasks at design-time and hence these are not suitable for dynamic workloads. Dynamic (run-time) mapping techniques are required for these types of workload, to map them onto the platform resources [30,43,47]. In dynamic approach tasks are loaded into the system at run-time. Task migration can

☆ Some part of this research was presented at 12th Euromicro Conference on Digital System Design, August 2009, pp. 133–140.

* Corresponding author. Tel.: +65 6790 6639; fax: +65 6792 0774.
E-mail addresses: amit0011@ntu.edu.sg (A.K. Singh), astsrikan@ntu.edu.sg (T. Srikanthan), akash@nus.edu.sg (A. Kumar), asjgwu@ntu.edu.sg (W. Jigang).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.04.007

also be used to insert a new task into the system at run-time [30,18,7]. In heterogeneous MPSoC, task migration is used to improve the performance by relocating a task from one PE to another PE when a performance bottleneck is detected or when the workload needs to be distributed more homogeneously. Issues related to the task migration such as the cost to interrupt a given task, saving its context, transmitting all of the data to a new PE and restarting the task in the new PE are discussed in [30,18,7].

1.1. Contributions

This work presents a new task mapping strategy and four new run-time task mapping heuristics based on it. The presented heuristics are applied onto an MPSoC platform, where each processing element (PE) can support more than one task. Multi-task supported PEs are considered to analyze a more realistic platform. Two types of PEs (*Processors and Reconfigurable Areas*) are considered. Processors are used to execute software tasks and reconfigurable areas for hardware tasks. Processors executing more than one task manage the execution of tasks by switching among the tasks after completing one operation of a task, similar to the execution in OSs where the tasks can communicate through some common register or memory space. Reconfigurable areas (RAs) are considered large enough to support more than one hardware tasks in parallel. The new presented heuristics take maximum advantage of the multi-tasking PEs by placing the communicating tasks on the same PE that results in reduced communication overhead between them. These mapped tasks can interact with each other very fast as they are on same PE and do not require any network resources. Our proposed mapping heuristics carefully map the communicating tasks on the same PE at run-time. Additionally, heuristics also try to map the tasks of an application in close proximity within a particular region in order to further reduce the communication overhead between the communicating tasks. Some parts of this research are published in Henkel et al. [41]. In Henkel et al. [41], only RAs are able to support more than one task and two run-time mapping heuristics based on packing strategy are presented. Run-time mapping heuristics reported in literature and heuristics in Henkel et al. [41] do not perform well when applied to the MPSoC platform where each PE is multi-tasking as they are not capable of taking advantage of the multi-tasking PEs to a large extent. To overcome the limitations and drawbacks of these heuristics, the four new mapping heuristics are presented. The new presented heuristics show significant performance improvements when compared to the latest run-time mapping heuristics reported in the literature. The performance metric includes overall execution time, energy consumption, average channel load and average packet latency.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 presents the MPSoC architecture. In Section 4, proposed mapping strategies along with the heuristics based on them are presented. Experimental set-up and the results are presented in Section 5 with Section 6 concluding the paper and providing future directions.

2. Related work

Mapping of tasks onto the MPSoC platform require finding the placement of tasks into the platform in view of some optimization criteria like reducing energy consumption, reducing total execution time and optimizing occupancy of channels. If the MPSoC platform is heterogeneous, then a task binding process is required before finding the placement for a task. The binding process involves defining a platform resource for each task type like instruction set processors for software tasks and FPGA tiles for hardware tasks. Task mapping is accomplished by static (design-time) or dynamic (run-time) mapping techniques.

2.1. Static mapping techniques

Static mapping techniques for NoC-based and bus-based MPSoCs are presented in [35,16,28,24,36,37] respectively, to solve the problem of mapping. Genetic approach in [21,48], Tabu Search in [28,23] and stimulated annealing in [24,22,32] is used. In [16,23,24], energy-aware mapping algorithms are presented. These techniques find fixed placement of tasks at design-time with a well known computation and communication behavior. Therefore, these mapping techniques are not suitable for an adaptive system that changes its configuration over time and requires re-mapping/run-time mapping of applications. Run-time mapping techniques are required for adaptive systems, where the workloads are dynamic (e.g. networking and multimedia applications).

2.2. Run-time mapping techniques for homogeneous MPSoCs

Mapping techniques targeting homogeneous MPSoCs are presented in [12,34,29,10,25]. Chou et al. [12] propose a run-time mapping strategy that incorporates the user behavior information in the resource allocation process; that allows system to better respond to real-time changes and adapt dynamically to user needs. This consideration saves 60% communication energy. Peter et al. [34] present a heuristic algorithm that is distributed over the processors and thus can be applied to systems of random size. Also, tasks added at run-time can be handled without any difficulty, allowing for inline optimization. The mapping results for several example task sets show that the mapping quality achieved by the presented algorithm is within 25% of that of the exact algorithm, for a 3×3 processor array.

Ngouangal et al. [29] describe a run-time mapping technique based on the attraction forces between communicating tasks that tries to place them close to each other in the MPSoC. Briao et al. [10] present dynamic task allocation strategies in MPSoCs based on bin-packing algorithms with task migration capabilities for running soft real-time applications. Two types of algorithms are combined to get better allocation results. Mehran et al. [25] present a Dynamic Spiral Mapping (DSM) heuristic algorithm for 2D mesh topologies where placement for a task is searched in a Spiral path, trying to place the communicating tasks close to each other.

2.3. Run-time mapping techniques for heterogeneous MPSoCs

Some researchers target heterogeneous MPSoCs and have developed run-time mapping techniques for mapping application's tasks onto them. Smit et al. [43] present a run-time task assignment algorithm that maps a task before all other task that needs a scarce resource by taking the availability of resources into account. Efficient heterogeneous multi-core architectures for streaming applications and run-time mapping of these applications onto these multi-core architectures are presented in Smit et al. [42]. Nollet et al. [31] describe a run-time task assignment heuristic for efficiently mapping the tasks on an MPSoC containing FPGA fabric tiles. With the presence of FPGA fabric tiles, algorithm is capable of managing a configuration hierarchy and it improves the task assignment success rate and quality. Holzenspies et al. [15] present a run-time spatial mapping technique consisting of four steps to map the streaming applications onto a heterogeneous MPSoC. The algorithm is implemented on an ARM926 running at 100 MHz and it takes less than 4 ms to run the HIPERLAN/2 example.

Faruque et al. [3] present a run-time agent based distributed application mapping technique for large MPSoCs such as 32×64 systems. The approach reduces the monitoring traffic and computational effort. Schranzhofer et al. [39] propose a polynomial-time heuristic algorithm that applies a multiple-step heuristic consisting of initial solutions followed by task remapping algorithms considering power constraints. First, initial solutions for the power-aware scenario are derived based on linear programming relaxation, and then task remapping is performed to improve the solutions. Theocharides et al. [44] demonstrate a run-time, system-level bidding-based task allocation strategy that gives significant performance improvements when compared to a round robin allocation.

Task migration mechanism presented in Nollet et al. [30] uses *task migration points* as a point of reference for migrating a task from one PE to another. Authors in Bertozzi et al. [7] use *checkpoints*, to define the point when a given task can be migrated and in Briao et al. [10] it is based on a *copy* model.

Carvalho and Moraes [11] present heuristics for run-time mapping of tasks in NoC-based heterogeneous MPSoCs. Tasks are mapped on the fly, according to the communication requests and the load in the NoC links. The target MPSoC architecture contains software and hardware PEs, where each PE can support only one task. For the target MPSoC in Carvalho and Moraes [11], efficient run-time mapping techniques based on packing strategy were presented in Singh et al. [40] that show the potential of the packing approach even for the platforms where each PE supports single task and in Singh et al. [41] their potential when the hardware resources modeled to support more than one task. Here, in our target MPSoC, each PE can support more than one task where communicating tasks get mapped on same PE, resulting in reduced communication overhead between them. Mapping heuristics Nearest Neighbor (*NN*) and Best Neighbor (*BN*) presented in Carvalho and Moraes [11] and two run-time mapping heuristics presented in Singh et al. [41] are taken for evaluation and performance comparison with our new proposed mapping heuristics.

3. NoC-based target MPSoC architecture

MPSoC architecture used in this work is an extended version of that used in Carvalho and Moraes [11]. The architecture contains a set of different processing nodes which interact via a communication network [27] as in Fig. 1. In Carvalho and Moraes [11], each processing node was capable of supporting only a single task. In the extended version, each processing node is modeled to support more than one task. The nodes can support either software tasks or hardware tasks. Software tasks execute in instruction set processors (*ISPs*) and hardware tasks execute in reconfigurable areas (*RAs*) or in dedicated IP-cores (*IPs*). Induction of RAs in the platform facilitates flexibility to hardware at a similar level to the software (*ISPs*) programmability. The communication network [27] has a 2D mesh topology that uses wormhole packet switching, handshake control flow, input buffers and deterministic *XY* routing algorithm, where first the packet is transferred in *X*-direction and then in *Y*-direction for transferring packets from one processing node to another processing node. For inter-task communication, message passing protocol is used, similar to that described in Carvalho and Moraes [11].

Among the available processing nodes, one of them is used as the Manager Processor (*M*) that is responsible for *task mapping* (*task binding and task placement*), *task scheduling*, *resource control* and *configuration control*. The configuration overhead results are used to simulate the *configuration control* process [26]. *Task binding* is required before *task placement* if the platform is heterogeneous as described is Section 2. *Task scheduling* uses queue strategy and there are three queues, one for each type (i.e. hardware, software and initial) of task. These tasks are defined in next section. Initial task is the starting task of an application that is mapped first. A task enters into its corresponding queue (hardware, software or initial) if there is no supported free resource in the platform. The task waits in the queue until a resource of same type is not free.

The *M* knows only the initial tasks for each application. Once, the initial tasks are mapped and their execution is started, the communication requests are sent to the communicating tasks at run-time and they are loaded into the MPSoC platform from the *task memory* if they are not already present in the platform. For *resource control*, the resources status is updated at run-time to provide the Manager Processor with an accurate information about the resource occupancy as task mapping decision is taken based on the PEs and NoC use.

4. Proposed mapping strategies

This section describes our proposed mapping strategies and efficient run-time mapping heuristics developed with these strategies. First we introduce some definitions for proper understanding of the mapping strategies and heuristics.

4.1. Definitions

Definitions necessary to explain the mapping strategies and heuristics are as follows:

Definition 1. An *application task graph* is represented as an acyclic directed graph $TG = (T,E)$, where T is set of all tasks of an application and E is the set of all edges in the application. Fig. 2 describes an application having initial, software and hardware tasks along with

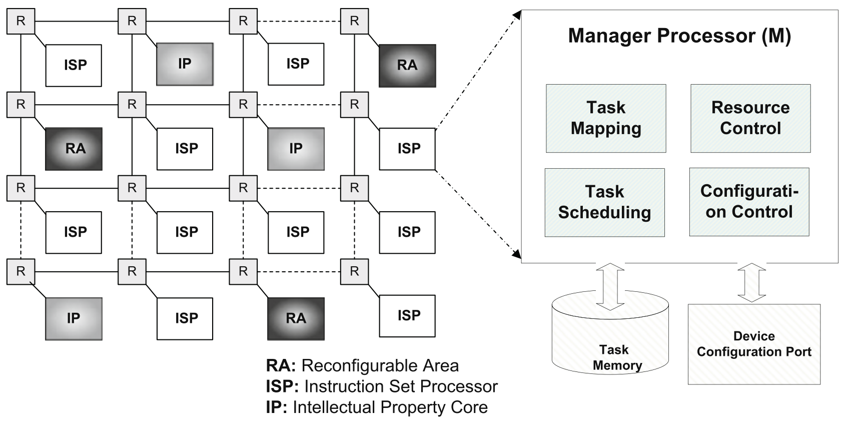

Fig. 1. Conceptual MPSoC architecture.

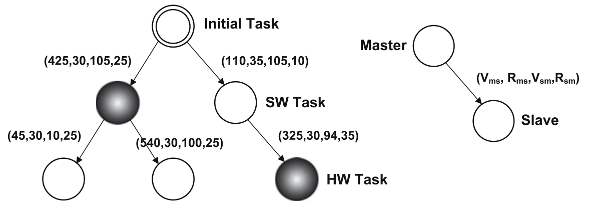

Fig. 2. Application modeling and master-slave.

the edges (E) connecting these tasks and shows a master-slave pair (communicating tasks). A connection (edge) between two tasks defines master-slave pair as in Fig. 2, i.e. a connection contains master and salve tasks. Initial task has no master. A task $t_i \in T$ is represented as (t_{id}, t_{type}, t_{exec}), where t_{id} is the task identifier, t_{type} is the task type (hardware, software, initial) and t_{exec} is the task execution time. E contains all the pair of communicating tasks and is represented as $(mt_{id}, st_{id}, (V_{ms}, R_{ms}, V_{sm}, R_{sm}))$, where mt_{id} represents the master task identifier, st_{id} represents the slave task identifier; V_{ms} and R_{ms} are the data volumes and data rate sent respectively from master to slave (ms); V_{sm} and R_{sm} are the data volumes and data rate sent respectively from slave to master (sm) respectively. The message rates (R_{ms}, R_{sm}) are described as percentage of available link bandwidth. As mentioned XY routing algorithm is used to transmit and receive the messages and both rates are relevant in the model as the path taken by messages may be different.

Definition 2. A NoC-based heterogeneous *MPSoC architecture* is a directed graph $AG = (P,V)$, where P is the set of tiles p_i and $v_{i,j} \in V$ presents the physical channel between two tiles p_i and p_j. A tile $p_i \in P$ is represented as $(p_{id}, p_{add}, p_{type}, p_{tasks}, p_{tasksnum}, p_{cap})$, where p_{id} is the tile identifier, p_{add} is the tile address and is used to receive packets, p_{type} is the tile type (hardware, software, initial), p_{tasks} is the task set mapped on the tile, $p_{tasksnum}$ ($\leqslant p_{cap}$) is the number of tasks mapped on the tile and p_{cap} is the capacity of tile showing the maximum number of tasks it can support. If $p_{tasksnum}$ reaches to p_{cap} then no further task can be mapped on the tile. Each physical channel $v_{i,j} \in V$ keeps the *channel width* information in bits and *available bandwidth usage* (% of available bandwidth) for transmission of data.

Definition 3. The task mapping is represented by function *mpg*: $t_i \in T \mapsto p_i \in P$, that maps a task of an application to a tile in the MPSoC platform.

4.2. Packing strategy

Here, we introduce our packing strategy for efficient mapping of applications onto a NoC-based MPSoC. The packing strategy was proposed in [41] and it has been shown that this strategy gives significant performance improvement when compared to the existing strategies. The strategy is now applied in the MPSoC platform, where each PE can support more than one task.

To map the applications by this strategy, firstly, initial (starting) tasks of applications are mapped as far away as possible while avoiding the edges in a distributed manner as shown in Fig. 3. A clustering approach as shown in Fig. 3 is used to find the placement of initial tasks in a distributed manner so that the new communicating tasks for each application can be mapped close to each other that results in reduced communication overhead. The cluster boundaries are virtual and hence a common region can be shared by tasks of different applications.

After the initial tasks get mapped, new communicating tasks of each application are mapped according to the communication request. To map a requested task, firstly, the task is tried to be mapped at the same node (master task PE) making the request as all the resources can support more than one task. If the task is not supported by the node making the request then it is tried to be mapped on the PEs around the node making the request at hop distance of one. The PEs are searched in sequence of left, down, top and right denoted as 1, 2, 3 and 4, respectively in Fig. 3 for one application in the most bottom-left cluster. This way, first, left and down side PEs are searched to find the placement. Now, if neither left nor down side PE is able to execute the task only then task is tried to be mapped on the top or right side PE according to the above defined sequence. The same strategy is followed from lower to higher hop distances until a free supported PE is found. Each application follows above defined strategy to map the requested tasks on the MPSoC platform resources.

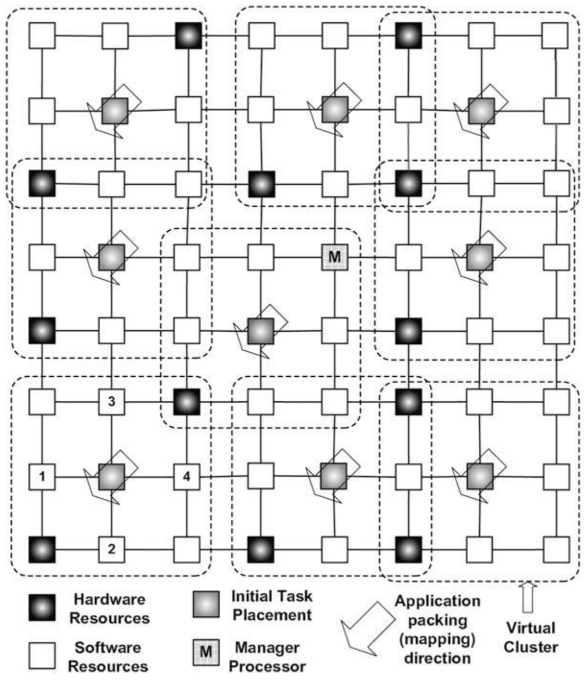

Fig. 3. Initial tasks placement for mapping applications with packing strategy.

With this strategy, first each application tries to map its requested task towards bottom-left (either on left or down PE) side within the cluster hence the PEs present on top-right edge of the cluster may be used by tasks of other applications that are also trying to map their tasks towards bottom-left (either on left or down PE). In this manner if one application is getting mapped then the applications that are tried to be mapped on top-side, right-side or top-right side may get the free resources on the top and right edges of the first application's cluster and tasks of the other applications can be mapped on these resources. This strategy is applied to all the applications to be mapped and most of the applications get the free resources from other application's top-right edge of the cluster. Thus, resource utilization is increased. Additionally, as each platform resource can support more than one task, communicating tasks get mapped on the same resource, resulting in further reduction in communication overhead and making the mapping more compact.

4.3. Packing-based mapping algorithms

This section describes run-time mapping algorithms that are motivated by the packing strategy discussed before. First initial tasks mapping methods are described followed by two run-time mapping heuristics. The initial tasks mapping methods are used to find the placement of initial tasks of applications. After the initial tasks are mapped and start executing, new tasks are requested to be mapped at run-time when communication to them is required. The run-time mapping heuristics are used to find the placement of new requested tasks. These heuristics are light-weight in terms of execution cycles, energy consumption, channel load and packet latency as these reduce the communication overhead on which all the performance metrics are highly dependent.

4.3.1. Initial task mapping

The initial tasks are considered as software tasks so these are mapped onto software processing elements. Initial task mapping has significant impact on the performance of run-time mapping and this can be done in two different ways. In the first method, the initial tasks can be mapped on the first free position found in the network that can support the tasks. This may result the initial tasks to be placed very close to each other. Therefore, when new tasks of different applications are requested to be mapped, the applications have to share the same NoC region, resulting in longer waiting time for a resource to become free for the tasks. This also increases the channel congestion as all the applications are tried to be mapped within a small region. In the second method, a clustering approach as shown in Fig. 3 is adapted to find the placement of initial tasks in a distributed manner as described in Section 4.2. This method reduces the interference between different applications. This work considers the clustering approach.

The Manager Processor (*M*) knows only the initial tasks. It does not know the whole application graphs. When initial tasks start their execution, communication requests are sent to the M to map the slave tasks at run-time. Efficient run-time mapping algorithms are required in order to map these new requested tasks for better performance gain. In next sub-sections our run-time packing-based mapping heuristics are presented.

4.3.2. Packing-based nearest neighbor (PNN)

This algorithm is based on the packing strategy discussed in Subsection 4.2 along with the search space (from lower to higher hop distances) of Nearest Neighbor (*NN*) heuristic and is presented in Fig. 4. In order to map a new requested task, the number of free supported resources in the platform are found. If any resource is available (step 3) then mapping for the requested task is found. First, resources (PEs) at the requesting node position (at zero hop distance; step 4) are selected and evaluated to map the task. If none of the PE can support then PEs at higher hop distances are selected and evaluated until the mapping is found. The search space to select the PEs goes upto the max_hop_count (NoC limit). The selection at each hop distance is done by function ***get_packing_ordered_list*** (hop_distance) (step 5), where PEs are selected according to the packing strategy i.e. in left, down, top and right order. As soon as, a free supported PE is found, the task is mapped onto the PE and selection and evaluation process is stopped (step 8). If there is no free supported PE in the platform for the requested task then the task is entered into its corresponding queue (step 13) and waits until a supported PE becomes free (step 14) by finishing execution of some previously mapped task. The queued task is mapped onto the freed supported PE as and when it is available (step 16). After mapping the requested task, it is entered onto the mapped list (*mpg*) and resources are updated, to have correct resources status for next requested task. The same strategy is followed by each requested task until all the tasks of the application are mapped.

To map multiple applications onto the MPSoC platform, the above described algorithm (PNN) is applied for each application. First, initial tasks of applications are mapped in a distributed manner by the clustering approach as in Fig. 3. Then, new requested tasks from each application are mapped dynamically, by applying Algorithm PNN. The PNN algorithm reduces the communication overhead by mapping the communicating tasks close to each other and sometimes onto the same PE at run-time. Thus, performance improvements are achieved.

```
Input: TG(T,E), AG(P,V) // task $t_i \in T$ ; PE $p_i \in P$ (PE)
Output: mpg (mapping $TG(T,E) \rightarrow AG(P,V)$ )
     // NFR[type]: number of free resource(s) of type type in NoC
 1: Map the initial task ($INI \in T$) at the centre of the cluster;
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if $NFR[ti_{type}]$ != 0 then
 4:     for all hop_distance = 0 to max_hop_count do
 5:       PE_list = get_packing_ordered_list(hop_distance);
 6:       for all PEs ∈ PE_list do
 7:         if $ti_{type}$==$pi_{type}$ AND resource available at $p_i$ then
 8:           Map $t_i$ onto PE $p_i$ and exit to step 17;
 9:         end if
10:       end for
11:     end for
12:   else
13:     insert($t_i$ to Queue($ti_{type}$));
14:     wait until $NFR[ti_{type}]$ != 0;// updated at run-time
15:     release($t_i$ from Queue($ti_{type}$));
16:     Map $t_i$ onto the freed resource at node $p_i$;
17:     insert($p_i$ to mpg); update(resources by mpg);
18:     wait and goto step 3 if new task $t_i \in T$ is requested;
19:   end if
20: end for
```

Fig. 4. Algorithm **PNN**.

4.3.3. Packing-based best neighbor (PBN)

This algorithm is combination of the algorithm PNN and path load computation approach. For each mapping z, path load is computed by Eq. (1), where $r_{ch(i,j)}$ and $r_{ch(j,i)}$ are the rates in the individual channels, from the master to the new requested slave and the rates in the channels in opposite direction.

$$cost_z = \sum r_{ch(i,j)} + \sum r_{ch(j,i)} \tag{1}$$

At each hop distance, after finding the PE list by PNN algorithm (step 5 in PNN), here all the free supported PEs are evaluated unlike the algorithm PNN, where evaluation is stopped when first free supported PE is found. For all the supported PEs, path load is computed and the PE with minimum PL is chosen for final mapping in order to get the best neighbor from the available neighbors. The free supported PE at zero hop distance has zero path load as no channel is involved. The evaluation process is stopped for higher hop distances if mapping is found. The rest of the steps are similar to PNN algorithm.

As this heuristic includes the path load computation, hence it is a congestion aware mapping heuristic that tries to distribute the channel load in the NoC. Thus, in addition to the communication overhead reduction, this heuristic also tries to distribute the channel load more uniformly.

4.4. Novel communication-aware strategy

The existing mapping heuristics reported in the literature and those proposed above do not provide significant performance improvement when applied to the multi-tasking MPSoC platform as they do not map the communicating tasks in a highly communication-aware manner. Thus, they are not able to efficiently utilize the multi-tasking resources of the MPSoC platform.

4.4.1. Problem with existing approach

One possible mapping by Nearest Neighbor (NN) mapping strategy is shown in Fig. 5. The mapping of an application onto the part of the MPSoC platform is depicted. For the shown mapping example, each PE in the platform is assumed to support maximum two tasks. However, in actual, the PEs can support large number of tasks and results for varying number of tasks on each PE are presented in Section 5. For mapping the application task graph, first, the initial task (0) is mapped and other tasks are mapped at run-time when a communication to them is required. When the initial task starts its execution, it requests its communicating tasks (1 and 2) and their mapping is found on the NN basis. As each PE is assumed to support maximum two tasks, so task 1 can be mapped onto the same position as its master (0) and task 2 will be mapped on some neighbor PE. A possible mapping is shown in the Fig. 5. After mapping tasks 1 and 2, they start their execution and their communicating tasks (task 3 and 4 for task1; task 5 and 6 for task2) are requested and mapping is found on NN basis. The possible mapping for tasks 3–6 is shown. In the same manner, tasks 7–9 are requested and mapped. The communication volume is shown in Fig. 2. The communication between the communicating tasks start when they are mapped.

Only one communicating task pair (tasks 0 and 1) gets mapped onto one PE that reduces that communication overhead. All other communicating tasks need to communicate from two different PE, thus there is lot of communication overhead. This communication overhead will be reduced if more communicating pairs are mapped onto the same PE. Next, we discuss the strategy that makes it possible to reduce the communication overhead by a large amount.

4.4.2. Solution with proposed approach

This strategy maps the requested task by looking the previously mapped task on the PE. The placement for the requested task is searched in increasing hop distances (hop_distance = 0 to max_hop_count) that results in mapping of all the tasks of an application close to each other. After finding the placement (PE) for the requested task, previously mapped tasks at the PE are found. If found PE does not have any previously mapped task then the PE is evaluated for mapping. Otherwise, the previously mapped task(s) are checked to have communication with the requested task. The requested task is mapped at the same position (PE) if they communicate; otherwise it is mapped onto next possible position even if it is supported at that position. The same process is adapted for each requested task. This strategy forces mapping of the communicating tasks onto the same PE if they can be supported and avoids mapping of the non-communicating tasks onto same PE. This process may cause some leaf tasks to occupy the whole PE without sharing it with other tasks if they are not mapped one the same PE as their masters. The leaf tasks don't have any slave that can be requested. But, in dynamic scenarios it may not be able to predict future tasks, so some leaf tasks can occupy the whole PE.

One possible mapping for an application onto the part of MPSoC is depicted in Fig. 6. After mapping initial task (task 0), communicating tasks 1 and 2 are requested. Task 1 is mapped with task 0 and task 2 is mapped onto some other PE, as each PE is assumed to support maximum two tasks. Now, tasks 1 and 2 request their communicating tasks 3,4 and 5,6 respectively. Task 3 is not mapped on the same PE as task 2 (previous strategy) as they are not communicating tasks. One possible mapping could be as shown in Fig. 6. Task 4 is mapped onto a new PE as the available PEs where tasks are previously mapped are not communicating with it. Task 5 is mapped on the same PE as task 2 as these are communicating tasks. Task 6 can be mapped only with task 2 as they are communicating but the PE is full, so it is mapped onto some different PE where no other task is present. Now, tasks 7–9 are requested and gets mapped as shown in Fig. 6.

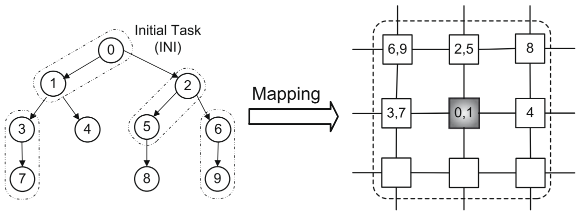

Fig. 6. Possible mapping of an application by the proposed strategy.

This strategy forces the mapping of most of the communicating pairs onto the same PE as shown in Fig. 6. Each of the communicating task pairs (0, 1), (2, 5), (3, 7) and (6, 9) are mapped onto the same PE, resulting in reduced communication overhead between them. We can map the leaf tasks 4 and 8 on the same PE even if they are not communicating in order to increase the resource utilization but in dynamic scenarios it can't be predicted that tasks 4 and 8 are leaf tasks as future tasks are unknown. Also, if they are mapped onto same PE then communication overhead between tasks 1, 4 and tasks 5, 8 will be increased. So, we map them on separate PEs. Thus, communication overhead is greatly reduced by mapping maximum communicating pairs onto the same PE.

4.4.3. Ideal mapping solution

A static mapping approach can provide better solution than dynamic mapping approaches provided the applications' structure and workload of their tasks is known at design-time. The static approach is performed at design-time with well known computation and communication behavior of tasks and resources status, enabling to explore better mapping decisions. However, the dynamic approach is adequate for the scenarios where the applications' structure and their workload is available only at run-time that can not be handled by the static approach. So, if applications are known at design-time, the static approach provides a better solution and thus an ideal mapping solution.

By knowing all the tasks at design-time, most of the communicating pairs can be mapped onto the same PE to reduce the communication overhead. The ideal mapping of an application onto the part of MPSoC is depicted in Fig. 7. The communicating task pairs (0, 2), (1, 4), (3, 7), (5,8) and (6,9) are mapped onto the same PE, providing a better solution than a dynamic mapping strategy

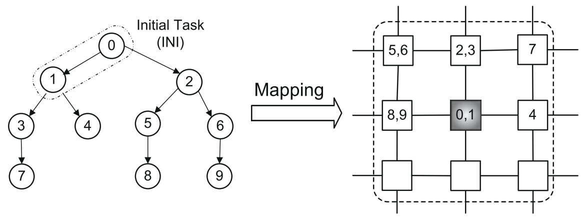

Fig. 5. Possible mapping of an application by nearest neighbor mapping strategy.

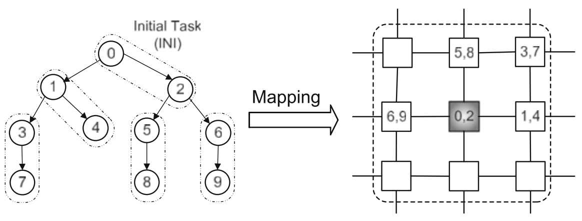

Fig. 7. Ideal mapping of an application with static mapping decision.

where maximum number of communicating pairs are tried to be mapped on the same PE at run-time. However, the ideal static mapping decisions provide better solution, these can not be applied to dynamic scenarios where workload of tasks is unknown at design-time.

4.5. Communication-aware mapping algorithms for multi-tasking platforms

This section details some communication-ware mapping heuristics developed with the new proposed strategy that are very efficient when applied to multi-tasking MPSoC platform.

Applications' mapping is started by first mapping the initial tasks (Fig. 3) as explained in Section 4.3. When the initial tasks start their execution, communicating tasks are requested and their mapping is found.

To find the placement for a requested task, first it is checked if there is any free resource in the platform, having same type (HW, SW or INI) as of requested task. If yes, then placement would be found by scanning the whole platform in increasing hop distances starting from the requesting node (hop_distance = 0) to up to the mac_hop_count (NoC limit). If there is no supported (same type) free resource in the platform then the task is entered into its corresponding queue and waits until a supported resource becomes free. The queued task is mapped on the freed supported resource and the control is transferred to find the placement for another unmapped requested task after updating the resources status. The same strategy is applied for all the tasks whenever they get requested.

4.5.1. Communication-aware nearest neighbor (CNN)

This algorithm is explained in Fig. 8. At each hop distance, PEs list is found by function ***get_basic_ordered_list*** (hop_distance) (step 5), where PEs are selected in top, down, left and right order as in NN heuristic proposed in Carvalho and Moraes [11]. Now, these PEs are evaluated in their selection order, to find the best suitable PE for the requested task. If the selected PE can support the task (step 7) then previously mapped tasks onto the PE are found (step 8). If there is no previous mapped task (i.e. first task onto the PE) (step 13) then requested task is mapped onto the PE (step 14); otherwise previously mapped tasks are checked to have communication with the requested task (step 10). If they are communicating, then only the task is mapped onto the PE (step 11); otherwise next possibility is evaluated, so that the PE can accommodate some another task having communication with the already (previously) mapped one. Just after finding the mapping, the selection and evaluation process is stopped (steps 11 and 14) even if there might be another supported PE at the same hop distance. Resources are updated after mapping in order to have accurate information about their occupancy for other requested tasks. This heuristic maps tasks of each application almost similarly as in Fig. 6.

```
Input: TG(T,E), AG(P,V) // task $t_i \in T$ ; PE $p_i \in P$ (PE)
Output: mpg (mapping $TG(T,E) \rightarrow AG(P,V)$ )
      // NFR[type]: number of free resource(s) of type type in NoC
 1: Map the initial task ($INI \in T$) at the centre of the cluster;
 2: for all unmapped task $t_i \in T$ that is requested do
 3:    if $NFR[ti_{type}]$ != 0 then
 4:       for all hop_distance = 0 to max_hop_count do
 5:          PE_list = get_basic_ordered_list(hop_distance);
 6:          for all PEs ∈ PE_list do
 7:             if $ti_{type}$==$pi_{type}$ AND resource available at $p_i$ then
 8:                Find previously mapped tasks on $p_i$;
 9:                if previous_tasks != NULL then
10:                   if comm[$t_i$][any_previous_task] then
11:                      Map $t_i$ onto PE $p_i$ and exit to step 24;
12:                   end if
13:                else
14:                   Map $t_i$ onto PE $p_i$ and exit to step 24;
15:                end if
16:             end if
17:          end for
18:       end for
19:    else
20:       insert($t_i$ to Queue($ti_{type}$));
21:       wait until $NFR[ti_{type}]$ != 0;// updated at run-time
22:       release($t_i$ from Queue($ti_{type}$));
23:       Map $t_i$ onto the freed resource at node $p_i$;
24:       insert($p_i$ to mpg); update(resources by mpg);
25:       wait and goto step 3 if new task $t_i \in T$ is requested;
26:    end if
27: end for
```

Fig. 8. Algorithm **CNN**.

4.5.2. Communication-aware packing-based nearest neighbor (CPNN)

This algorithm incorporates packing strategy in CNN algorithm. At each hop distance, the selection of PEs is done by function ***get_packing_ordered_list*** (hop_distance) (step 5 in CNN) which returns PEs according to the packing strategy i.e. in left, down, top and right order. All the other steps for this algorithm are same as of CNN.

By choosing left and down side PEs first, the PEs of top and right edge of the cluster (Fig. 3) are intentionally made free so that these can be used by other applications running on top and right side. This way all the applications' tasks are mapped in bottom-left fashion utilizing the PEs on the top and right edge of the cluster from other applications. Thus, resource utilization increases that results in improved performance.

4.5.3. Communication-aware best neighbor (CBN)

This algorithm is combination of the path load computation approach and CNN algorithm and is presented in Fig. 9. Path load computation approach is incorporated by calculating the path load (step 15) for each PE from Eq. (1). In CNN algorithm, at each hop distance, if any evaluated PE is suitable for the requested task then it is selected for mapping and other PEs at the same hop distance are not evaluated. In contrast to CNN, here all the PEs are evaluated (selected temporarily- step 18) and finally, the PE with minimum path load is considered for final mapping. The same strategy is followed at each hop distance until placement for the requested task is found.

This algorithm considers traffic (congestion in channels) while finding the placement for a requested task, hence it is a congestion-aware mapping heuristic that tries to distribute the channel load more homogeneously in the NoC.

4.5.4. Communication-aware packing-based best neighbor (CPBN)

This algorithm incorporates packing strategy in CBN algorithm by selecting the PEs by function ***get_packing_ordered_list*** (hop_distance) (step 6) which returns PEs according to the packing strategy. All the other steps for this algorithm are same as of CBN.

This algorithm takes the advantage of the packing strategy and considers traffic as well in order to distribute the loads in the channels more homogeneously.

5. Performance evaluation

Experiments are performed by *co-simulation* in *ModelSim* (*System-C* for applications and *RTL-VHDL* for the NoC) and the results evaluated are *total execution time*, *energy consumption*, *average channel load* and *average packet latency* for applications.

```
Input: TG(T,E), AG(P,V) // task $t_i \in T$ ; PE $p_i \in P$ (PE)
Output: mpg (mapping TG(T,E) → AG(P,V) )
 1: Map the initial task (INI ∈ T) at the centre of the cluster;
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if NFR[$ti_{type}$] != 0 then
 4:     for all hop_distance = 0 to max_hop_count do
 5:       weight = MAX_VALUE; // some large value
 6:       PE_list = get_basic_ordered_list(hop_distance);
 7:       for all PEs ∈ PE_list do
 8:         if $ti_{type}$==$pi_{type}$ AND resource available at $p_i$ then
 9:           Find previously mapped tasks on $p_i$;
10:           if previous_tasks != NULL then
11:             if comm[$t_i$][any_previous_task] then
12:               Map $t_i$ onto PE $p_i$ and exit to step 29;
13:             end if
14:           else
15:             weightTemp = calcChannelLoad(when $t_i$ mapped
                onto $p_i$);
16:             if weightTemp < weight then
17:               weight = weightTemp;
18:               Select node $p_i$ temporarily to map $t_i$;
19:             end if
20:           end if
21:         end if
22:       end for
23:       if weight < MAX_VALUE then
24:         Map $t_i$ onto PE $p_i$ and exit to step 29;
25:       end if
26:     end for
27:   else
28:     Perform steps 20 to 23 from CNN;
29:     insert($p_i$ to mpg); update(resources by mpg);
30:     wait and goto step 3 if new task $t_i \in T$ is requested;
31:   end if
32: end for
```

Fig. 9. Algorithm **CBN**.

5.1. Experimental set-up

Experiments are performed on a simulation platform that is an extended version of that used in Carvalho and Moraes [11]. The extended simulation platform supports more than one task at each node. We have performed the simulation by varying the number of tasks to be supported at each node. However, it is known that larger memory space and reconfigurable area will be required to support more number of software and hardware tasks respectively, so we have performed simulation up to maximum four tasks per node. The memory space required at each node depends on both the number of tasks and size of the tasks to be mapped. The tasks mapped on a node get executed one after another in time multiplexed manner. At each node, the memory space depends on the number of tasks as other mapped tasks need to be stored (kept active) in memory while one is running. For the experimentation, all the tasks are considered of same size so the memory space is governed by the number of tasks (all same size). At the nodes, the memory space or reconfigurable area availability issues are avoided by limiting the number of tasks to be supported by maximum four.

The processing elements (PEs) are modeled using *System-C*. Two different *System-C* threads are used to model the PEs, one for the Manager Processor (M) and another for rest of the PEs as *MPthread* and *TASKthread* respectively. The *MPthread* is responsible for the MPSoC resource management, task mapping, task scheduling and task configuration. This thread contains channels occupation metrics, PEs occupation metrics and scheduling queues to manage system use. The resource metrics are updated at run-time by monitoring the resources status with the help of monitors attached to all the NoC ports. The *TASKthread* is responsible for the task behavior implementation that is described by a configuration file. This file contains execution time and communication rates and these values can be customized.

Each application is modeled as in Fig. 2, with an initial task, hardware tasks and software tasks. The values present on the edges represent the volume and rate of data to be exchanged between the master and slave as explained in definition 1 of Section 4.1. Each task transmits from 200 to 500 packets (data volumes (V) on the edges as in Fig. 2) with size varying from 100 to 400 16-bit flits. After receiving a packet corresponding to some particular task on a PE, the packet is processed for some definite time before starting the processing of next packet corresponding to the same task on the PE. If two tasks are mapped on the receiver PE then processing time of packets corresponding to each task get doubled as the packets are processed in time multiplexed manner. The processing time gets tripled for packets when processed on a PE containing three tasks and so on. The simulation is performed at varying processing time to analyze the computation–communication behavior. Hardware and software tasks allocation time is taken as 1300 and 100 clock cycles respectively [26]. Initial tasks are mapped onto the processors, so the configuration time is the same as that of software tasks.

The evaluated scenarios are:

(i) *Identical tree like applications having all tasks as software*: (Parallel benchmarks have this profile), each having 10 tasks, where one task is taken as initial (starting task) and rest 9 as software tasks.
(ii) *Identical tree like applications having hardware and software tasks*: Each having 10 tasks, where one task is taken as initial task with different combination hardware/software tasks.
(iii) *Random applications having hardware and software tasks*: Random generated applications using *Task Graph For Free* (TGFF[13]). Each have one initial and random number of hardware/software tasks (varying from 4 to 9).

In the first two scenarios, simulation is performed with injection rate varying from 5% to 20% (% usage of available channel bandwidth) and in third it is random from 5% to 30%.

The NoC is modeled in VHDL [27], in an 8×8 2D-mesh topology. NoC is responsible for data transfer between the tasks. As handshake protocol is used to transfer the data therefore each flit is transmitted in two clock cycles, limiting the available channel bandwidth to 50% of its capacity.

For the scenario (i) evaluation, 8×8 NoC-based homogeneous MPSoC is taken, where all the PEs are processors. For evaluating scenarios (ii) and (iii), 8×8 NoC-based heterogeneous MPSoC is taken with 52 nodes as processors and 12 nodes as reconfigurable areas. In all the scenarios, one software node is used for the Manager Processor (M) that is considered to support a single task.

The number of simultaneously running applications (initial tasks) are varied according to the processing capability of the platform that gets increased when number of tasks to be supported at each PE is increased. This variation is required to utilize all the platform resources, otherwise some resources might be just idle and doing nothing. For the considered MPSoC platforms containing 2, 3 or 4 tasks supported PEs, the experiments are performed by taking 10, 14 and 18 applications respectively.

The initial task placement is done by a clustering approach, where the processing capability of a cluster is determined by the non-shared PEs within the cluster. The processing capabilities of clusters are stored in advance. The applications to be mapped are sorted in descending order by the number of tasks in them, before the actual mapping starts. The initial tasks of the sorted applications are mapped on the center of the clusters sorted in their decreasing processing capability. Thus, an application containing more number of tasks is tried to be mapped into a more processing capability cluster and an application containing relatively less number tasks into a less processing capability cluster, resulting in better resource utilization. This approach is not useful for the scenarios where all the applications contain same number of tasks like scenario (i) and (ii). Scenario (iii) contains applications with varying number of tasks in them so the approach is useful.

5.2. Experimental results

The number of applications executing at a time is increased with the processing capability of the platform by dividing the NoC into corresponding number of clusters. The mapping heuristics NN &BN proposed in Carvalho and Moraes [11], PNN, PBN, CNN, CPNN, CBN, CPBN and the ideal static mapping (ISM) decision are evaluated on three different platforms that contain PEs supporting two, three or four tasks.

5.2.1. Total execution time

It is the time taken to finish the execution of all the applications to be mapped on the platform. It comprises of mapping time (time to find the placement), configuration time, communication time, waiting time (when no free resource in the platform) and computation (processing) time amongst which communication time dominates. The computation of a packet corresponding to a task mapped on a PE starts just after its receiving and the computation is finished before receiving the next packet for the same task. So, if the computation time is less than the time interval between receiving two consecutive packets corresponding to the same task on the PE then the computation time will get absorbed within the communication time. Thus, the computation time should be large enough in order to contribute to the total execution time.

New proposed mapping heuristics map the maximum number of communicating pairs onto the same PE, resulting in reduced communication overhead and the traffic in channels (channel congestion) that gets generated when communicating pairs communicate from different PEs. The reduced traffic decreases the communication overhead of other communicating tasks that communicate from different PEs. Thus, total execution time is reduced.

Our analysis in time complexity shows that all the heuristics have time complexity of same level and is of $O(C)$, where C is the number of processing elements in the NoC. All the heuristics execute almost in similar time with some improvements in some cases.

Fig. 10 shows average execution time required for the first simulated scenario at different platforms when heuristics NN, CNN, BN, CBN and ISM are employed. The average for each heuristic is taken after executing it at varying injection rate. A couple of observations can be made from the Fig. 10. First, the proposed CNN and CBN perform better than NN and BN respectively, at each platform and thus are scalable for platforms containing PEs to support even higher number of tasks. Second, the largest gain for CNN and CBN over NN and BN respectively is witnessed for 3 tasks/PE platform. For this platform, CNN and CBN show an average gain of 46.07% and 46.30% when compared to NN and BN respectively. Third, ISM outperforms all the heuristics at each platform as the mapping decision is taken at design-time with a global view of the platform resources and takes maximum advantage from the task graph structure. Communication-aware heuristics CPNN and CPBN also get similar improvements over PNN and PBN respectively.

5.2.2. Energy consumption

Energy is required when a packet needs to be transmitted from source PE to destination PE and then to process the packet at the destination PE after it is received. The energy required in transmission and processing are referred as communication and computation energy respectively.

The communication energy depends on the number of bits to be transmitted, the number of links to be traversed between both the PEs and energy required in transmitting one bit through one link. The transmitted bits are calculated by multiplying number of packets by the average packet size in bits. Here, number of packets is considered as data volume V_{ms} (Fig. 2) and average packet size as ten flits each of 16 bits denoted as P_{size}, when transferred from master to slave. As communication takes place from slave to master also, so total bits include the number of packets transferred from slave to master as well and the number is considered as V_{sm} (Fig. 2) having same average packet size. The number of links to be traversed between the source and destination PE is calculated from the Manhattan distance ($\Delta X_{ms} + \Delta Y_{ms}$) between the PEs as XY routing algorithm is used. The energy required to transmit one bit through each link is considered as E_{Lbit} [33]. The communication energy is estimated as product of number of bits to be transmitted, the number of links to be traversed between source and destination PE and the energy required to transmit one bit through one link, for each master-slave pairs from Eq. (2).

$$E_{comm} = \sum[(V_{ms} + V_{sm}) \times P_{size} \times (\Delta X_{ms} + \Delta Y_{ms}) \times E_{Lbit}] \quad (2)$$

The computation energy depends on the number of bits to be processed on the receiver PE, time required to process each received bit and power needed to process the bit. The bits to be processed are the same that were transmitted from some source PE and are calculated by multiplying number of packets (V_{ms}) by the average packet size (P_{size}), when received by slave. The total bits for each master-slave pair includes the bits to be processed on the master PE ($V_{sm} \times P_{size}$) as well, when received by master and sent by slave. The time required to process each bit is calculated by dividing the time taken to process each packet (t_{comp}) by the average packet size (P_{size}). The value of t_{comp} is provided by a configuration file. The power needed to process the bits on a PE is estimated from the power efficiency of Tile64 processor [2]. In [2], power efficiency is varied from 15 to 22 W when all the 64 PEs operate at 700 MHz simultaneously. The power is scaled for one

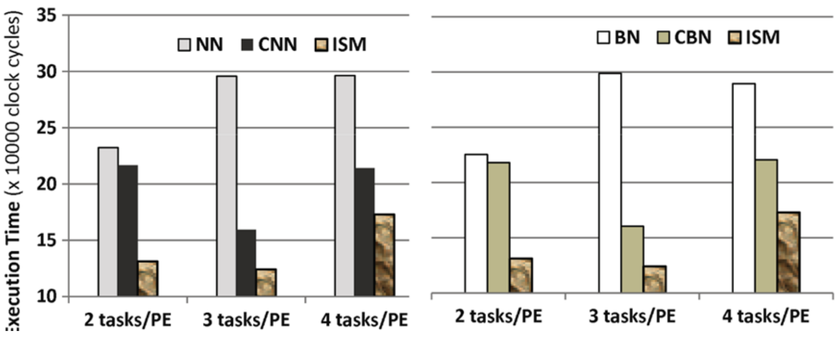

Fig. 10. Execution time for NN, CNN, ISM and BN, CBN, ISM heuristics at different platforms.

PE operating at 25 MHz and is referred as PE_{power}. The scaling is done for 25 MHz as the NoC [27] also operates at 25 MHz and its very reasonable for the PEs to operate at the same frequency. An average power dissipation of 20W is considered while scaling is performed. The computation energy is estimated as product of the number of bits to be processed, time required to process one bit and power needed to process the bit, for each master-slave pairs from Eq. (3).

$$E_{comp} = \sum \left[(V_{ms} + V_{sm}) \times t_{comp} \times PE_{power} \right] \quad (3)$$

Total energy consumption is estimated as the sum of communication and computation energy from Eq. (4). Our proposed mapping strategy reduces the distance between source and destination PE by placing the communicating tasks onto the same PE, where bits can be exchanged very easily through some common memory or register without the need of much communication energy. Thus, total energy consumption is greatly reduced.

$$E_{total} = E_{comm} + E_{comp} \quad (4)$$

Fig. 11 shows energy consumption (E_{total}) for all the simulated scenarios in different platforms when heuristics PNN and CPNN are employed. A number of observations can be made from the Fig. 11. First, CPNN always performs better than PNN and maximum gain of CPNN over PNN is witnessed for scenario-1 at each platform. At 4 tasks per PE platform, CPNN shows an improvement of 44.27% over PNN. Second, the energy consumption for PNN and CPNN is minimum for scenario-2 at each platform when compared with other scenarios. Similar behavior is shown by other proposed communication-aware heuristics too.

5.2.3. Average channel load

The average channel load represents the NoC use. It is calculated by looking the loads in all the channels at a fixed clock cycle interval until all the applications finish execution. The load in the channels depends on the communication overhead between the tasks and the traffic produced by the communicating tasks while communicating from different nodes. The communication overhead and the traffic produced is reduced by the proposed mapping heuristics by mapping maximum number of communicating pairs onto the same processing node. Thus, average channel load is significantly reduced when proposed heuristics are employed.

Fig. 12 plots average channel load for all simulation scenarios at varying injection rates, for 3 tasks per PE platform. Similar behavior is shown for other platforms too. The average channel load increases with communication rate as more traffic gets generated in the channels with increase in the communication rate. CPBN reduces the average channel load for all the scenarios when compared to PBN. CPBN shows an average gain of 10.67% over PBN. Other proposed heuristics bring about almost similar improvements.

5.2.4. Average packet latency

The average packet latency depends on the congestion in the path and the distance between the source and destination PE on which communicating tasks are mapped. Proposed mapping algorithms try to map the maximum communicating task pairs onto the same processing node. The latency for the packets of the communicating tasks mapped on the same PE is reduced very much as the packets can be easily exchanged on the PE without needing any channel but we have not considered latency for these packets while calculating the average packet latency. However, mapping the communicating tasks onto the same PE reduces congestion in channels that helps in reducing packet latency for other tasks communicating from different PEs. Thus, proposed mapping algorithms reduce the average packet latency and the improvements for different evaluated scenarios are shown in Table 1.

Table 1 shows the latency results for all simulated scenarios for two tasks per PE platform. Communication-aware mapping heuristics CNN, CPNN, CBN and CPBN reduce the average packet latency when compared to NN, PNN, BN and PBN respectively. Improvements are shown as *% Gain by Communication-aware Algorithms* in the last row of Table 1. Improvements are not significant as packets for the communicating tasks mapped on the same PE are not considered. ISM performs better than all other heuristics for scenario-1 and scenario-2 but can not be applied to scenario-3 as applications structure and their workload is random and not known at design-time. Other evaluated platforms show almost similar results.

5.2.5. Effect of computation–communication ratio

The computation–communication ratio (CCR) is estimated as the ratio of desired computation time (in cycles) and desired communication time (in cycles) for all the packets from the following equation, where $t_{computation}$ and $t_{communication}$ are the desired computation and communication time for individual packets.

$$CCR = \sum \left[t_{computation} \right] \div \sum \left[t_{communication} \right] \quad (5)$$

The number of packets to be transferred (communicated) and processed (computed) remains same as all the transferred packets need to be processed at some PE. Since, every packet is considered identical, each has the same desired computation time ($t_{computation}$) that is provided through a configuration file, and the same desired communication time ($t_{communication}$) that is calculated for NoC [27] operating at 25 MHz for a given injection rate (% usage of available bandwidth). Thus, CCR simply reduces to the ratio of $t_{computation}$ and $t_{communication}$ due to the same number of identical packets. The value of $t_{communication}$ remains fixed for a given rate. Therefore, in order to get varying values of CCR, different values of $t_{computation}$ (clock cycles) are provided through the configuration file.

The total execution time mainly consists of computation and communication time. The computation of a packet starts just after it is received on a PE and is finished before the next packet is re-

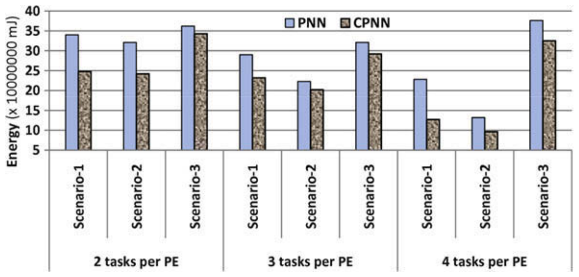

Fig. 11. Energy consumption for PNN and CPNN heuristics in different platforms for all scenarios.

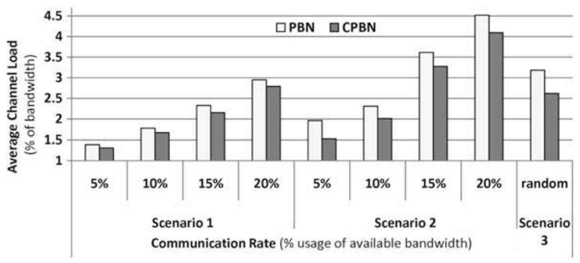

Fig. 12. Average channel load for PBN and CPBN heuristics for all simulation scenarios.

Table 1
Average packet latency for all simulated scenarios for the presented communication-aware heuristics, existing heuristics and ideal static mapping (ISM) mapping decision. Gain by communication-aware heuristics are shown.

Scenarrios	Rate (%)	Average packet latency (clock cycles)								
		NN	CNN	PNN	CPNN	BN	CBN	PBN	CPBN	ISM
Scenario 1	5	127	120	120	118	118	114	115	113	110
	10	226	220	225	225	216	216	217	216	211
	15	326	313	316	313	316	309	314	309	303
	20	431	421	425	417	419	405	414	405	411
Scenario 2	5	118	113	123	120	116	109	118	116	105
	10	216	215	220	215	214	214	215	213	212
	15	312	309	320	310	310	310	311	307	303
	20	409	404	429	406	410	406	407	403	397
Scenario 3		273	270	278	273	278	275	287	280	NA
% Gain by communication-aware algorithms		–	2.17%	–	2.40%	–	1.63%	–	1.50%	

ceived. So, the computation time should be greater than the time interval between receiving of two consecutive packets in order to contribute to the total execution time.

Fig. 13 shows the total execution time for mapping algorithms NN and CNN at varying CCR when applied to scenario (i) at an injection rate of 5% (% usage of available bandwidth). The execution time behavior is shown for 2 tasks per PE platform. Other platforms also show almost similar behavior. It is clear that CNN always performs better than NN. The gain by CNN over NN vary for different CCR. The gain behavior for different platforms is described subsequently.

Fig. 14 shows gain (%) in total execution time for mapping algorithm CNN over NN when applied to the scenario (i) for different platforms at varying CCR. The gains shown are for the injection rate of 5%. A couple of observations can be made from the Fig. 14. For 2 tasks per PE platform, the gain is constant for some initial values of CCR. For these CCR values, the total execution time for both the algorithms (NN and CNN) remain fixed as different values of computation time (vary with CCR) for each packet gets absorbed in the time interval between receiving of consecutive packets. The constant gain is due to the communication time saving by employing the CNN algorithm to map most of the communicating task pairs on the same PE. The communication time is saved as the packets for most of the communicating tasks are processed on the same PE without sending to any other PE. With further increment in CCR, we see continuous gains up to some CCR values and then a falling trend. The initial continuous gain is very drastic as increase in computation time adds to total execution time very much for NN when compared to the CNN. The NN gets affected much as the computation times for most of the packets are not getting absorbed between the time interval of receiving consecutive packets whereas in CNN, most of the packets are exchanged on the same PE. Therefore, the total execution time for CNN is not affected much as the computation time for the packets of the communicating tasks mapped on the same PE gets absorbed within the communication and computation time of other tasks' packets processed in parallel. The falling trend at higher values of CCR is obtained when the computation time starts adding to the total execution time significantly, for the packets of the communicating tasks mapped on the same PE too for the CNN heuristic. In the falling trend region, for CNN, the computation time does not get absorbed within the computation and communication time of other parallely processed packets and thus the total execution time gets affected by all the packets for both the heuristics.

For 3 tasks per PE platform, the gain falls for some initial values of CCR and then shows a similar trend as in 2 tasks per PE platform. For initial values of CCR, increase in computation time starts affecting total execution time for the CNN heuristic as the computation time for the packets of communicating tasks mapped on the same PE is not getting absorbed within the communication and computation time of other tasks' packets due to longer time required to process the packets. The time required in processing gets longer as the packet for a particular task is processed for some fixed clock cycles in time multiplexed manner along with packets of other tasks mapped on the same PE and here more number of tasks get mapped on a PE. However, the total execution time for NN remains same for smaller values of CCR as computation time for these values of CCR does not add

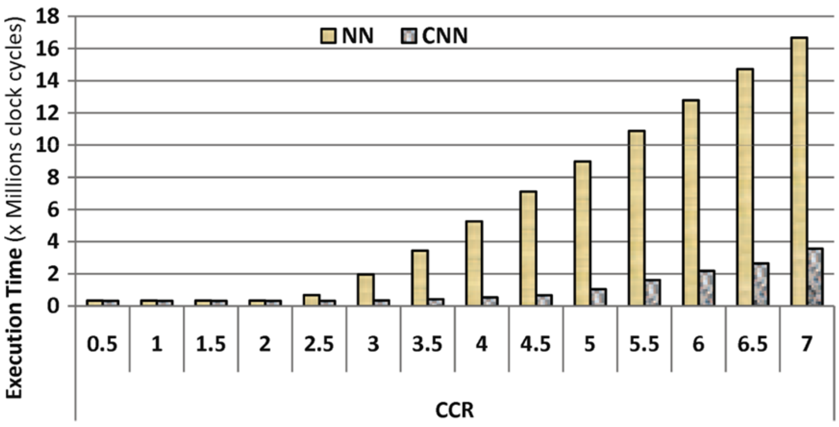

Fig. 13. Total execution time for NN and CNN at varying CCR for two tasks per PE platform.

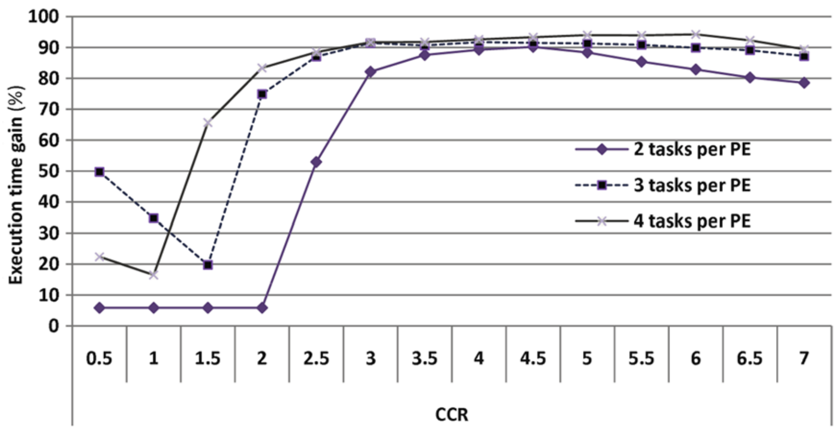

Fig. 14. Improvement in total execution time for CNN over NN at varying CCR for different platforms.

to the total execution time due to the same reason as in 2 tasks per PE platform. Thus, a falling trend is obtained for some initial values of CCR. The reason for the similar trend after some values of CCR is same as explained for 2 tasks per PE platform. The drastic gain starts relatively at lower CCR values as compared to the 2 tasks per PE platform due to more tasks getting mapped on the PEs. For 4 tasks per PE platform, a similar trend is obtained with drastic gain starting relatively more early due to the similar reason.

The initial gain for 3 tasks per PE platform is higher than other platforms. The almost flat region in mid values of CCR is longer for platforms supporting more number of tasks. Other heuristics also show similar behavior.

5.2.6. Clustering vs. non-clustering approach

The clustering approach benefits the evaluated metrics only in the third simulation scenario where applications contain varying number of tasks. In the clustering approach, first, applications are sorted by the number of tasks within them and then initial tasks of applications are mapped at the center of the clusters sorted by their processing capability as explained in experimental set-up section. In non-clustering approach, applications are not sorted and their initial tasks are mapped at any random location.

Fig. 15 shows gain obtained by the clustering approach over the non-clustering approach for average channel load, energy consumption, average packet latency and total execution time at the two tasks per PE platform for different mapping algorithms. Average channel load and energy consumption is improved by around 15% with some improvement in packet latency and execution time. Almost similar behavior is obtained at other evaluated platforms too.

6. Conclusions and future work

This paper describes a new mapping strategy where placement for a task is found by looking at previously mapped tasks onto a processing element (PE) in the multi-tasking MPSoC platform. We have relied on this mapping strategy to propose four run-time mapping heuristics. A simulation platform has been extended to support the mapping of more than one task on each PE, which can be either a CPU or a reconfigurable hardware (RH) block.

We show that the ideal static mapping solution can lead to performance improvement. However, the ideal static mapping has been shown to improve the overall performance only when all the applications and their workloads are known prior to the mapping process. However, this does not cater for realistic scenarios in which the run-time characteristics call for dynamic mapping strategies.

Based on our investigations all the proposed heuristics have been evaluated using an 8×8 NoC-based MPSoC platform. We clearly demonstrate that the newly presented heuristics can consistently provide for notable reduction in the communication overhead. The potential of mapping adjacent communicating tasks and those of the same application on to the same PE whenever possible has contributed to the overall reduction in the communication overhead. We have investigated different scenarios depending on performance metrics of interest and they show that improvement in the total execution time can be up to 90% when the packet execution time is increased.

In future, we plan to evaluate real-time benchmarks on the MPSoC platform and devise techniques for task migration when performance bottlenecks are identified at run-time.

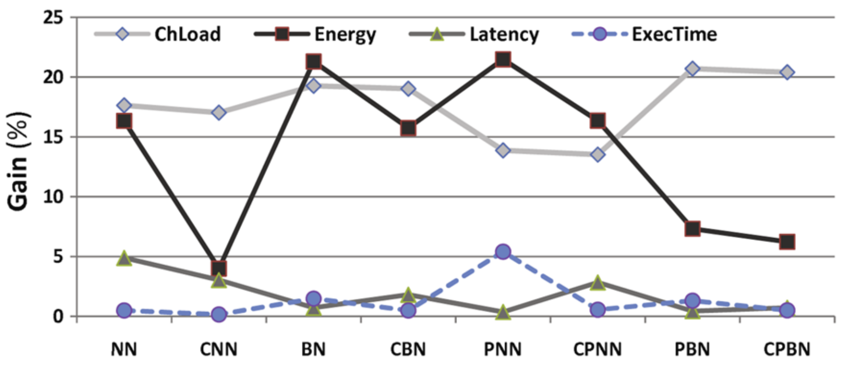

Fig. 15. Improvements of clustering approach when different mapping algorithms are employed.

Acknowledgments

We thank Mr. Alok Prakash for discussions and contributions to the paper [41] and special thank to Dr. Ewerson Carvalho for providing the basic simulation environment used in Carvalho and Moraes [11].

References

[1] Chip Multiprocessor Watch, 2008. <http://view.eecs.berkeley.edu/wiki/Chip_Multi_Processor_Watch>.
[2] Tile64 Processor, 2008. <http://www.tilera.com/products/TILE64.php>.
[3] M.A. Al Faruque et al., Adam: run-time agent-based distributed application mapping for on-chip communication, in: Proceedings of the DAC, 2008, pp. 760–765.
[4] S. Bell et al., Tile64tm processor: a 64-core soc with mesh interconnect, in: IEEE International Solid-State Circuits Conference, 2008, pp. 88–598.
[5] L. Benini, G. De Micheli, Networks on chips: a new soc paradigm, Computer 35 (1) (2002) 70–78.
[6] D. Bertozzi, L. Benini, Xpipes: a network-on-chip architecture for gigascale systems-on-chip, Circ. Syst. Mag. IEEE 4 (2) (2004) 18–31.
[7] S. Bertozzi et al., Supporting task migration in multi-processor systems-on-chip: a feasibility study, in: Proceedings of DATE, 2006, pp. 15–20.
[8] S. Borkar, Thousand core chips: a technology perspective, in: Proceedings of DAC, 2007, pp. 746–749.
[9] M. Branca et al., Evolutionary algorithms for the mapping of pipelined applications onto heterogeneous embedded systems, in: Proceedings of the Gen. and Evolutionary Comp., 2009, pp. 1435–1442.
[10] E.W. Briao et al., Dynamic task allocation strategies in mpsoc for soft real-time applications, in: Proceedings of DATE, 2008, pp. 1386–1389.
[11] E. Carvalho, F. Moraes, Congestion-aware task mapping in heterogeneous mpsocs, in: International Symposium on SoC, November 2008, pp. 1–4.
[12] C.-L. Chou, R. Marculescu, User-aware dynamic task allocation in networks-on-chip, in: Proceedings of DATE, 2008, pp. 1232–1237.
[13] R.P. Dick et al., Tgff: task graphs for free, in: Proceedings of Workshop on Hardware/Software Codesign, 1998, pp. 97–101.
[14] J. Henkel et al., On-chip networks: a scalable, communication-centric embedded system design paradigm, in: Proceedings of VLSI Design, 2004, p. 845.
[15] P.K.F. Hölzenspies et al., Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc), in: Proceedings of DATE, 2008, pp. 212–217.
[16] J. Hu, R. Marculescu, Energy- and performance-aware mapping for regular noc architectures. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 24(4) (2005) 551–562.
[17] A. Jerraya et al., Guest editors' introduction: multiprocessor systems-on-chips, Computer 38 (7) (2005) 36–40.
[18] H. Kalte, M. Porrmann, Context saving and restoring for multitasking in reconfigurable systems, in: Proceedings of FPL, 2005, pp. 223–228.
[19] M. Kistler et al., Cell multiprocessor communication network: built for speed, IEEE Micro 26 (3) (2006) 10–23.
[20] A. Kumar et al., Multiprocessor systems synthesis for multiple use-cases of multiple applications on fpga, ACM Trans. Des. Autom. Electron. Syst. 13 (3) (2008) 1–27.
[21] T. Lei, S. Kumar, Algorithms and tools for network on chip based system design, in: Proceedings of Integrated Circuits and Systems Design, 2003, p. 163.
[22] L.-Y. Lin et al., Communication-driven task binding for multiprocessor with latency insensitive network-on-chip, in: Proceedings of ASP-DAC, 2005, pp. 39–44.
[23] S. Manolache et al., Fault and energy-aware communication mapping with guaranteed latency for applications implemented on noc, in: Proceedings of DAC, 2005, pp. 266–269.
[24] C. Marcon et al., Time and energy efficient mapping of embedded applications onto nocs, in: Proceedings of ASP-DAC, 2005, pp. 33–38.
[25] A. Mehran et al., A heuristic dynamic spiral mapping algorithm for network on chip, IEICE Electron. Exp. 5 (13) (2008) 464–471.
[26] L. Möller et al., A NoC-based Infrastructure to Enable Dynamic Self Reconfigurable Systems, 2007.
[27] F. Moraes et al., Hermes: an infrastructure for low area overhead packet-switching networks on chip, Integr. VLSI J. 38 (1) (2004) 69–93.
[28] S. Murali et al., A methodology for mapping multiple use-cases onto networks on chips, in: Proceedings of DATE, 2006, pp. 118–123.
[29] A. Ngouanga et al., A contextual resources use: a proof of concept through the apaches' platform, in: Proceedings of IEEE Des. and Diagnostics of Electronic Circuits and Sys., 2006, pp. 42–47.
[30] V. Nollet et al., Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles, in: Proceedings of DATE, 2005, pp. 234–239.
[31] V. Nollet et al., Run-time management of a mpsoc containing fpga fabric tiles, IEEE Trans. VLSI Syst. 16 (1) (2008) 24–33.
[32] H. Orsila et al., Automated memory-aware application distribution for multi-processor system-on-chips, J. Syst. Archit. 53 (11) (2007) 795–815.
[33] J.C.S. Palma et al., Mapping embedded systems onto nocs: the traffic effect on dynamic energy estimation, in: Proceedings of Integrated Circuits and System Design, 2005, pp. 196–201.
[34] Z. Peter et al., A decentralised task mapping approach for homogeneous multiprocessor network-on-chips, Int. J. Reconfig. Comp., 2009.
[35] C.-E. Rhee et al., Many-to-many core-switch mapping in 2d mesh noc architectures, in: Proceedings of IEEE International Conference on Computer Design, 2004, pp. 438–443.
[36] M. Ruggiero et al., Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip, in: Proceedings of DATE, 2006, pp. 3–8.
[37] M. Ruggiero et al., A fast and accurate technique for mapping parrallel applications on stream-oriented mpsoc platforms with communication awareness, Int. J. Parallel Progr. 36 (1) (2008) 3–36.
[38] N. Saint-Jean et al., Hs-scale: a hardware-software scalable mp-soc architecture for embedded systems, in: Proceedings of ISVLSI, 2007, pp. 21–28.
[39] A. Schranzhofer et al., Power-aware mapping of probabilistic applications onto heterogeneous mpsoc platforms, in: Proceedings of Real-Time and Embedded Technology and Applications Symposium, 2009, pp. 151–160.
[40] A.K. Singh et al., Efficient heuristics for minimizing communication overhead in noc-based heterogeneous mpsoc platforms, in: Proceedings of International Symposium on Rapid System Prototyping, pp. 55–60, 2009.
[41] A.K. Singh et al., Mapping algorithms for noc-based heterogeneous mpsoc platforms, in: Proceedings of the Euromicro Symposium on DSD, 2009, pp. 133–140.
[42] G.J. Smit et al., Multi-core architectures and streaming applications, in: Proceedings of Interntional Workshop on System Level Interconnect Prediction, 2008, pp. 35–42.
[43] L. Smit et al., Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture, in: FPT, 2004, pp. 421–424.
[44] T. Theocharides et al., Towards embedded runtime system level optimization for mpsocs: on-chip task allocation, in: Proceedings of Great Lakes Symposium on VLSI, 2009, pp. 121–124.
[45] L. Thiele et al., Mapping applications to tiled multiprocessor embedded systems, in: Proceedings of Application of Concurrency to System Design, 2007, pp. 29–40.
[46] S. Vangal et al., An 80-tile 1.28tflops network-on-chip in 65 nm cmos, in: IEEE Internattional Solid-State Circuits Conference, 2007, pp. 98–589.
[47] F. Wronski, E. Brião, F. Wagner, Evaluating energy-aware task allocation strategies for MPSOCs, in: IFIP Distri. and Para. Embedded Sys., Springer, 2006.
[48] D. Wu et al., Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems, in: Proceedings of DATE, 2003, p. 10090.

Amit Kumar Singh received his Bachelor degree in Electronics Engineering from Indian School of Mines, Dhanbad, India, in 2006. He worked with HCL Technologies, India for year and half before joining Nanyang Technological University (NTU), Singapore, in 2008. Currently, he is working with Centre for High Performance Embedded Systems (CHiPES), School of Computer Engineering, NTU, Singapore as a research student towards the completion of his PhD. His research interests include NoC-based MPSoC design, run-time mapping algorithms.

Srikanthan joined Nanyang Technological University (NTU), Singapore in June 1991. At present, he holds a full professor and joint appointment as Director of a 100 strong Centre for High Performance Embedded Systems (CHiPES). He founded CHiPES in 1998 and elevated it to a university level research Centre in February 2000. He has also served as founding Director of the Intelligent Devices and Systems (IDeAS) cluster for 2 years (2005–2007). His research interests include design methodologies for complex embedded systems, architectural translations of compute intensive algorithms, computer arithmetic and high-speed techniques for image processing and dynamic routing. He has published more than 250 technical papers including 60 journals in IEEE Transactions, IEE Proceedings and other reputed international journals. His services as a key consultant to embedded systems industry, both locally and internationally are continually being sought for. He was awarded the Public Administration Medal (Bronze) on 2006 National Day in recognition of his contributions to education in Singapore.

Akash Kumar received the B.Eng. degree in Computer Engineering from the National University of Singapore (NUS), Singapore, in 2002. He received the joint Master of Technological Design degree in embedded systems from NUS and the Eindhoven University of Technology (TUe), Eindhoven, The Netherlands, in 2004, and received the joint PhD degree in Electrical Engineering in the area of embedded systems from TUe and NUS, in 2009. In 2004, he was with Philips Research Labs, Eindhoven, The Netherlands, where he worked on Reed Solomon codes as a Research Intern. From 2005 to 2009, he was with TUe as a PhD student. Since 2009, he has been with the Department of Electrical and Computer Engineering, NUS, currently as a Visiting Fellow. He has published over 25 papers in leading international electronic design automation journals and conferences. His current research interests include analysis, design methodologies, and resource management of embedded multiprocessor systems.

Wu Jigang received the B.Sc. degree in computational mathematics from Lanzhou University, China in 1983, and doctoral degree in computer software and theory from University of Science and Technology of China (USTC) in 2000. He was with the Department of Computer Science of Lanzhou University, China from 1983 to 1993, as an assistant professor followed by lecturer. He was with the Department of Computer Science and Engineering of Yantai University, China from 1993 to 2000, as a lecturer followed by associate professor. He was with the Centre for High Performance Embedded Systems, School of Computer Engineering, Nanyang Technological University, Singapore from 2000 to 2009, as a postdoctoral fellow followed by research fellow. He joined the School of Computer Science and Software, Tianjin Polytechnic University, China from 2009 as a full professor. He has published more than 100 technical papers including journals in IEEE Transactions, IEE Proceedings and other reputed international journals. His research interests include in reconfigurable VLSI design, hardware/software co-design, parallel computing and combinatorial search.