Available online at www.sciencedirect.com

Procedia Computer Science 1 (2010) 1019–1026

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Run-time mapping of multiple communicating tasks on MPSoC platforms

Amit Kumar Singh[a,1], Wu Jigang[a], Akash Kumar[b], Thambipillai Srikanthan[a]

[a]*School of Computer Engineering, Nanyang Technological University, Singapore*
[b]*Department of Electrical and Computer Engineering, National University of Singapore, Singapore*

Abstract

Multi-task supported processing elements (PEs) are required in a Multiprocessor System-on-Chip platform for better scalability, power consumption etc. Efficient utilization of PEs needs intelligent mapping of tasks onto them. The job becomes more challenging when the workload of tasks is dynamic. These scenarios require tasks to be mapped at run-time. This paper presents a run-time mapping technique for efficiently mapping the tasks of applications on the multitasking resources. The technique tries to map the communicating tasks onto the same processing resource and also the tasks of an application close to each other in order to reduce the communication overhead. For an evaluated scenario, the presented technique reduces total execution time by 22%, average channel load by 47% and power dissipation by 48% when compared to state-of-the-art run-time mapping techniques.

Keywords: Multiprocessor System-on-Chip (MPSoC), Network-on-Chip (NoC), Mapping Algorithms.

1. Introduction

The advancement in nanotechnology has made it feasible to develop a complete system on a single chip. The Systems-on-Chip (SoCs) can integrate several processing elements (PEs) towards the development of Multiprocessor Systems-on-Chip (MPSoCs). The PEs need to be connected by an on chip interconnect. Network-on-Chip (NoC) interconnection scheme seems to be used in future architectures as traditional schemes such as shared buses and point-to-point dedicated links are not scalable [1]. These days complex embedded applications are targeting MPSoCs as the performance requirements cannot be achieved by a system based on a single general purpose processor.

A homogeneous MPSoC composed of identical PEs supports only a few applications, whereas a heterogeneous MPSoC composed of different type of PEs can support a wider variety of applications. Recently, Intel [2] and Tilera [3] proposed homogeneous MPSoCs with 80 and 100 PEs respectively, connected by a NoC. IBM, Sony and Toshiba proposed a heterogeneous MPSoC composed of one manager processor and 8 floating-point units [4]. Future MPSoC architectures are anticipated to contain thousands of PEs in a single die by 2015 [5].

Embedded applications like multimedia and networking contain dynamic workload of tasks. This type of workload needs run-time management of tasks as the tasks enter into the system at run-time. Mapping tasks on the MPSoC architecture at run-time needs dynamic mapping techniques.

[*] *Email addresses:* [a]{amit0011,asjgwu,astsrikan}@ntu.edu.sg, [b]akash@nus.edu.sg
[1] Corresponding author

1877-0509 © 2010 Published by Elsevier Ltd.
doi:10.1016/j.procs.2010.04.113

The mapping techniques find the placement of tasks in the architecture in view of some optimization criteria like reducing energy consumption, total execution time etc. Most of the existing mapping techniques consider single task supported PEs in the platform and are optimized for the same. These techniques do not perform well when applied to an MPSoC containing multi-task supported PEs as they are not able to utilize the PEs efficiently. To overcome the drawbacks and limitations of these techniques, we present a new task mapping technique targeting MPSoCs containing multi-task supported PEs. The main goal is to reduce the communication overhead that minimizes congestion in the network. The presented heuristics take maximum advantage of the multi-tasking PEs by carefully mapping the communicating tasks on the same PE that results in reduced communication overhead between them. The tasks on the same PE can communicate faster as they do not need any network resource. The presented heuristic outperforms the existing ones significantly. The evaluated performance metrics include overall execution time, average channel load and power dissipation.

The rest of the paper is organized as follows. Section 2 provides an overview of related work in mapping. Section 3 presents the mapping problem formulation, target architecture and drawbacks of existing mapping strategies. In section 4, we present our proposed mapping technique to overcome the drawbacks of existing ones. The performance of the proposed and existing mapping techniques are evaluated in section 5. Section 6 concludes the paper and provides future directions.

2. Related Work

Task mapping can be accomplished by static (design-time) or dynamic (run-time) mapping techniques depending upon the workload scenarios. Static mapping is suitable for static workloads and the placement for a task is found with its well known computation and communication behavior at design-time. In dynamic workload scenarios, a task might come into picture at run-time, requiring dynamic mapping techniques to map it.

Some static mapping techniques are presented in [6, 7, 8]. Work in [6] uses genetic mapping algorithm and tries to optimize energy consumption. Tabu Search algorithm is adopted in [7], to explore the large search space for finding the placement of tasks. In [8], Simulated Annealing algorithm is employed to solve the mapping problem.

Dynamic mapping techniques are required to find the placement of tasks at run-time. Chou et al. [9] propose a run-time mapping strategy that incorporates user behavior information in the resource allocation process. Communication energy is saved by a large amount compared to the arbitrary task allocation strategies. Nollet et al. [10] describe a run-time task assignment heuristic for mapping the tasks on an MPSoC containing FPGA fabric tiles. The FPGA tiles facilitate for managing a configuration hierarchy that improves the task assignment success rate and quality. Smit et al. [11] present a mapping algorithm to map an application task-graph on a MPSoC at run-time. The algorithm tries to place each task near to its communicating entities in order to save the energy consumption. Faruque et al. [12] present an agent based distributed application mapping approach for large MPSoCs such as 32×64 systems. The approach reduces monitoring traffic and computational effort. H"olzenspies et al. [13] propose a run-time spatial mapping technique consisting of four steps to map the streaming applications on MPSoCs. Briao et al. [14] present dynamic task allocation strategies based on bin-packing algorithms for soft real-time applications. The energy is saved by turning off idle processors and applying Dynamic Voltage Scaling to processors with slack. Mehran et al. [15] suggest a Dynamic Spiral Mapping (DSM) algorithm for mapping an application on an MPSoC arranged in 2-D mesh topology. The placement for a task is searched in a Spiral path, trying to place the communicating tasks close to each other. Carvalho et al. [16] present heuristics for mapping tasks on MPSoCs at run-time. In [17], heuristics are refined to show performance improvements. Tasks are mapped on the fly, according to the communication requests and the load in the NoC links. Each PE in the MPSoC can support only one task. Differently from this, the PEs in our target MPSoC support more than one task. Among all the state-of-the-art heuristics presented in [16], Nearest Neighbor (*NN*) heuristic outperforms when evaluated for total execution time and shows almost similar results for average channel load. We have considered *NN* for evaluation and performance comparison with our proposed mapping technique.

3. Problem Statement

This section first introduces some definitions necessary to understand the mapping problem along with used MP-SoC architecture description, then shortcomings of existing mapping approaches and finally, the problem that we are heading to solve.

An **application task graph** is a directed graph *TG = (T, E)*, where *T* is a set of application tasks and *E* is the set of all edges in the application, connecting the tasks and representing their communication. An edge connecting two tasks of Video Object Plane Decoder (VOPD) application is shown in Figure 1. 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 its type (hardware, software, initial) and t_{exec} is the task execution time. An edge in the edge set *E* contains communication information between two connected tasks and is expressed as (V_{ms}, R_{ms}, V_{sm}, R_{sm}) like in Figure 1, where V_{ms} is the data volume to be sent from task *m* to *s* with injection rate R_{ms}. V_{sm} and R_{sm} have similar meaning for sending the data from task *s* to *m*. Volumes *V* show the flits to be transmitted and rates *R* show the desired bandwidth. The connected tasks are represented as master-slave pair. In Figure 1, *iQuant* is master (m) and *iDCT* is slave (s).

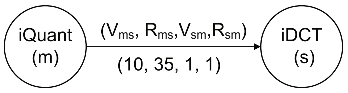

Figure 1: Two tasks of VOPD application and communication between them.

An **MPSoC architecture** is a graph *AG = (P, V)*, where *P* is the set of PEs and *V* represents the on chip communication channels for interconnecting the PEs. A PE is identified by its identifier p_{id} and its type is represented as p_{type}. Each physical channel $v_{i,j} \in V$ keeps the *available bandwidth usage* (% of available bandwidth) for transmitting the data.

In our proposed NoC-based MPSoC architecture, each PE supports more than one task. The PEs are connected in an 8×8 mesh topology by a NoC. Among the available PEs, one is used as *Manager Processor* and is responsible for managing task operations and resources usage, including run-time management of task loads. Task mapping is activated when a mapped task need to communicate with a not yet mapped task at run-time.

Task **mapping** is represented by function $mpg : t_i \in T \longmapsto p_i \in P$, that maps a task of the application to a PE in the MPSoC architecture.

One possible mapping of an *application task graph* on part of the *MPSoC architecture* by applying the nearest neighbor (NN) run-time mapping heuristic is shown in Figure 2. For demonstration, each PE is assumed to support maximum three tasks. First, the initial task (0) is mapped and other tasks are requested to be mapped at run-time when a communication to them is required. The placement for the requested task is searched in increasing hop distances (0 to max_hop_count). When the initial task starts executing, it requests its communicating slave tasks (1 & 2) and their mapping is found on the nearest possible neighbor PE. As each PE is assumed to support three tasks, so the requested tasks (1 & 2) can be mapped onto the same position as of task 0 (hop_distance = 0). After mapping tasks 1 and 2, they start their execution and their slave tasks (tasks 3 & 4 for task 1; tasks 5 & 6 for task 2) are requested and their mapping is found by the NN. The possible mapping for tasks 3, 4, 5 and 6 is shown. Now, when these tasks (3, 4, 5 & 6) start executing, their slave tasks (7, 8 & 9) are requested and their mapping is found. In the same manner, task 10 gets requested and mapped when task 7 starts its execution. A possible mapping for all the requested tasks of the application by NN heuristic is shown in Figure 2. The communication between the communicating tasks start when they are mapped.

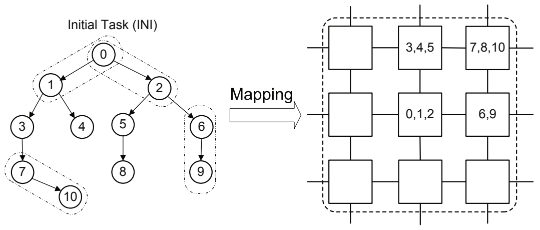

Figure 2: Mapping by an existing mapping heuristic.

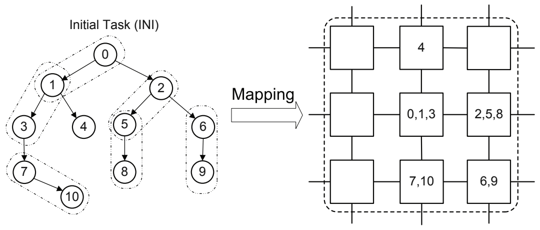

Figure 3: Mapping by the proposed technique.

Communicating task pairs (0, 1), (0, 2), (6, 9) and (7, 10) gets mapped onto the same PE. Thus, communication overhead gets reduced. But, rest of the communicating pairs need to communicate from different PEs, thus a lot of communication overhead still remains. This overhead can be further reduced if more communicating pairs can be mapped onto the same PE. Next, we discuss our proposed strategy that maps the communicating pairs systematically so that most of them are mapped on the same PE.

4. Proposed Mapping Technique

This section details our proposed mapping technique. The technique is explained with the help of an example application.

The mapping is started by the initial (starting) task of the application. Each application's initial task is mapped by using a clustering approach that divides the NoC into clusters or regions. Each cluster (Figure 3) supports only one initial (INI) task that is placed at the center of the cluster. So, resource sharing between applications is possible, but its occurrence is minimized. The boundaries of the clusters are virtual, so an application might use parts of different clusters if required.

When the initial task starts execution, communicating tasks are requested. For each unmapped requested task, the placement is found by scanning the whole network (hop_distance = 0 to max_hop_count) with the help of the Algorithm 1. If there is no supported resource in the platform, the task is entered into its corresponding type of queue and waits for a resource to become free that can execute the task (step 33). As and when a supported resource becomes free, the queued task is released from the queue and is mapped onto the resource (step 34).

The technique is applied on the same application as used before to see its potential over the existing techniques. The technique is explained with Algorithm 1. Unlike the Nearest Neighbor (NN) strategy as described before, here the placement for requested tasks is not found as and when they come into picture. Out of all the requested slave tasks from some particular task (master or requesting task), placement for the first one is found and others are stored in a queue named Application Queue (AQ). For a requested task (step 1), first its master task is found (step 2) then it is checked if the found master task is same as of the master of previous requested task and the present requested task is not a leaf task (step 3). If the checked condition (step 3) is satisfied, the requested task is entered into the AQ (step 4).

By following above conditions, mapping for the first requested task from a master is found and other requested tasks from the same master gets entered into the AQ. This facilitates in mapping of most of the communicating pairs on the same PE that might not happen if all the requested tasks are tried to be mapped at the same time. As the platform PEs are modeled to support multiple tasks, so the requested slave task can get mapped on the same PE as of the master task (step 9), reducing the communication overhead. After the mapped task starts executing, its unmapped communicating (slave) tasks are requested. Similarly as before, the placement for the first requested task is found and others are placed in the same AQ. The placement could be same as of the requesting task as PEs support multiple tasks, further reducing the communication overhead. Thus, many communicating pairs can be mapped on the same PE. The same process continues until the communicating tasks from one branch of the application are supported at the same PE. If not, the requested task is entered into the AQ and one of its previously queued task is taken out to find its mapping by NN. The mapping is found by scanning the NoC in increasing hop distances (1 to max_hop_count) from its requesting task (step 12). For the queued tasks, out of the many possible mapping positions, one that does not have any previously mapped task (step 17) is chosen so that its unmapped communicating tasks would be mapped on the same PE, to reduce the communication overhead. The scanning stops when the placement is found. The same process as described above continues for the requested tasks by the mapped queued task. Additionally, while finding the mapping for the queued task, it is not entered into the AQ again to avoid the unnecessary overhead of queuing and finding placement for it many times. For this, *comingFrmAppQ* variable is set while finding mapping for a queued task. Also, leaf tasks are not entered into the AQ (step 11) and are mapped even if not supported at the same PE as of its requesting task to avoid the unnecessary overhead of queuing and finding its mapping again. If a requested task does not gets mapped on the same PE as of its master and does not follow any of above conditions (step 23) then also the task is entered into the AQ (step 24). After queuing at this point, it is checked if there is any previously queued task (step 25) to start the mapping of previous queued tasks (step 26). The same strategy is followed by all the requested tasks. When there is no task to be requested and there is a task in the AQ, the mapping is started with the queued task.

Algorithm 1 Smart Nearest Neighbor (SNN)

```
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))
    // NFR[type]: number of free resource(s) of type type in NoC
 1: for all unmapped task $t_i \in T$ that is requested do
 2:   mstTID = Finding master of the requested task from buffer storage;
 3:   if mstTID == oldmstTID AND $t_i$ is not a leaf task then
 4:     scheduleToAppQ($t_i$, mstTID); oldmstTID = mstTID;
 5:     wait and go to step 1 if new task is requested;
 6:   else
 7:     if NFR[$ti_{type}$] != 0 then
 8:       if at requesting (hop = 0) PE $ti_{type}$==$pi_{type}$ & resource available at $p_i$ then
 9:         Map $t_i$ onto the requesting PE $p_i$ and exit to step 35;
10:       else
11:         if $t_i$ is a leaf task OR comingFrmAppQ then
12:           for all hop_distance = 1 to max_hop_count do
13:             PE_list = get_PE_list(hop_distance);
14:             for all PEs ∈ PE_list do
15:               if $ti_{type}$==$pi_{type}$ AND resource available at $p_i$ then
16:                 prev_tasks = Find previously mapped tasks on $p_i$;
17:                 if (prev_tasks == NULL & comingFrmAppQ) OR $t_i$ is leaf then
18:                   Map $t_i$ onto PE $p_i$ and exit to step 35;
19:                 end if
20:               end if
21:             end for
22:           end for
23:         else
24:           scheduleToAppQ($t_i$, mstTID);
25:           if queueAppNtids > 1 then
26:             leaveFrmAppQ($t_i$, mstTID); comingFrmAppQ = 1; go to 11: to map $t_i$;
27:           else
28:             go to step 1 for next requested task $t_i \in T$;
29:           end if
30:         end if
31:       end if
32:     else
33:       insert($t_i$ to Queue($ti_{type}$)); wait until NFR[$ti_{type}$] != 0;
34:       release($t_i$ from Queue($ti_{type}$)); Map $t_i$ onto the freed resource at node $p_i$;
35:       insert($p_i$ to mpg); update(resources by mpg);
36:       if mapped $t_i$ is leaf task AND queueAppNtids > 0 then
37:         Perform step 26;
38:       else
39:         go to step 1 for next requested task $t_i \in T$;
40:       end if
41:     end if
42:   end if
43: end for
```

One possible mapping of an application on part of the MPSoC using the proposed mapping technique is depicted in Figure 3. Here also, each PE is assumed to support maximum three tasks and the application is same to compare the final mapping with NN mapping strategy. After mapping initial task (task 0), communicating slave tasks 1 and 2 are requested. Placement for the task 1 is found and task 2 is placed in the AQ. Task 1 gets mapped with task 0 as each PE supports multiple tasks. When task 1 starts executing, it requests its communicating tasks 3 and 4. Task 3 gets mapped on the same PE as of its requesting task (task 1) and mapping for task 4 is also found as it is a leaf task that should not be queued according to the strategy described above. Now when task 3 starts execution, task 7 gets requested. Task 7 can't be mapped with task 3 as the PE supporting it already has three tasks mapped on. So, task 7 is entered into the AQ and previously queued task 2 is considered for mapping. According to the strategy, it needs to be mapped at a PE without any previous task so that its unmapped communicating tasks would be mapped with it. A possible mapping for task 2 is shown in Figure 3. For its communicating tasks 5 and 6, task 5 is mapped with it and task 6 is entered into the AQ. After mapping the task 5, its communicating task 8 is requested and gets mapped on the same PE. Since, there is no task to be requested, the queued task 7 is considered for mapping and should be mapped on a PE without any previous task. A possible mapping for it and its unmapped communicating task 10 is shown. After mapping task 10, queued task 6 is considered for mapping. Placement for task 6 with its unmapped communicating task 9 is shown.

This strategy maps most of the communicating pairs onto the same PE as shown in Figure 3. Communicating task pairs (0, 1), (1, 3), (2, 5), (5, 8), (7, 10) and (6, 9) get mapped on the same PE. Thus, communication overhead is greatly reduced as compared to the previous strategy where only few pairs were mapped on the same PE. We switch to find mapping of queued tasks without traversing all the tasks of a branch in order to start the parallel execution of tasks from another branches as well. This switching avoids long waiting for the tasks of another branch that can be executed in parallel and decreases chances of mapping the communicating tasks far from each other. All these considerations facilitate in finishing the application's tasks execution faster.

5. Performance Evaluation

Experiments are performed in co-simulation (*SystemC* for applications and *RTL-VHDL* for NoC [18]) by *ModelSim* Simulator. The NoC is arranged in 2D-mesh topology and it's parameters include input buffers, wormhole packet switching, 16-bit flit width and deterministic XY routing. *SystemC* is used to model the PEs with two *SystemC-threads*. One thread for the Manager Processor (MP) named *Mthread* is responsible for resource and task (mapping, scheduling) management. Second thread for rest of the PEs named *TASKthread* describes task behavior implementation that is specified in a configuration file containing execution time and communication rates for each task.

We evaluated 20 identical tree like applications (parallel benchmarks have this profile) each having 10 tasks and injection rate is varied from 5 to 20% (% usage of available bandwidth). Each application is modeled as in Figure 2, consisting of one initial (starting) and rest as software tasks. An 8×8 NoC-based MPSoC is considered, where all the PEs are processors. One PE is used for the MP and rest 63 PEs as software resources. All the processors except those supporting the MP and initial tasks are considered to support multiple tasks. Initial tasks act as manager of applications and have overhead for managing other tasks, so the supported processors are considered to support only one task. The PEs reserved for initial tasks are defined aiming to uniformly distribute them over the system area. The experiment has been performed at 2 tasks/processor, which is easily extendible to more number of tasks. The number of simultaneously running applications was varied and best results were obtained at 10 applications.

5.1. *Total Execution Time*

The total execution time is the overall time to execute all the applications. The overall time includes mapping, configuration, computation, waiting (when system does not have a supported resource) and communication time. Out of all these times, communication time dominates, which is required to transfer packets from one PE to another. This dominating time highly depends on the communication overhead that is greatly reduced by our proposed strategy. Thus, overall execution time gets reduced.

Figure 4.(a) shows total execution time when Nearest Neighbor (NN) and Smart Nearest Neighbor (SNN) heuristics are employed. The SNN heuristic outperforms NN at all the communication rates and an average improvement of 22.83% is achieved.

Our analysis in time complexity shows that both NN and SNN heuristics have time complexity of same level that is of O(C), where C is number of PEs in the NoC. The heuristics execute almost in similar time.

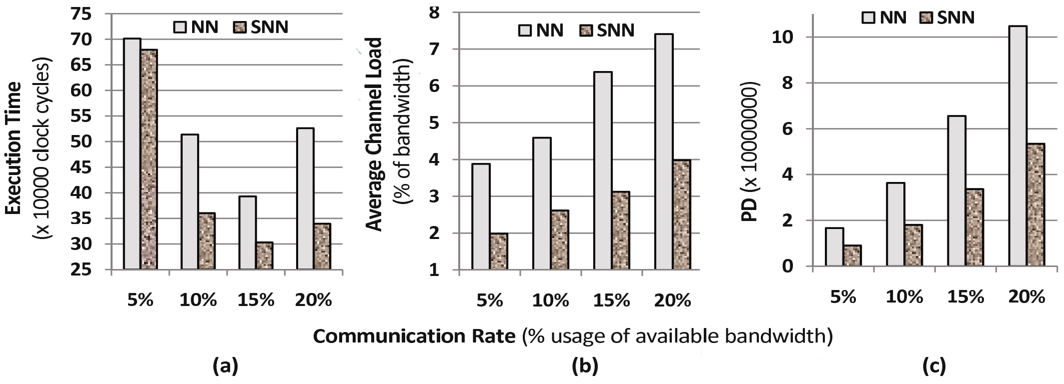

Figure 4: Overall Execution Time, Average Channel Load and Power Dissipation at different communication rates for NN and SNN mapping heuristics.

5.2. Average Channel Load

It represents the NoC use and is calculated by looking loads in all the channels at some fixed clock cycle interval until execution of all the applications is finished. The loads depend on traffic produced in channels while two tasks communicate from different PEs. The proposed strategy maps maximum communicating pairs on the same PE resulting in reduced traffic production in channels.

Figure 4.(b) plots average channel load for NN and SNN heuristics at varying rates. At all the rates, the SNN heuristic reduces the average channel load significantly when compared to NN. An average gain of 47.38% is achieved by SNN over NN.

5.3. Power Dissipation

Power is needed to transfer packets from source PE to destination PE on them communicating tasks are mapped. The amount of power depends on the distance between both the PEs and the number of bits to be transferred. The bits are calculated by multiplying the number of packets to average packet size and are considered as data volume V_{ms} and rate R_{ms} (Figure 1) respectively, when transferring from master to slave. As communication happens from slave to master as well, so total bits include multiplication of V_{sm} and R_{sm} (Figure 1) too. The distance between source-destination pair is considered as Manhattan distance ($\Delta X_{ms} + \Delta Y_{ms}$) as XY routing algorithm is used. The power dissipation is estimated as the product of number of bits to be transferred and distance between source-destination pair by Equation 1, for all master-slave pairs.

$$PD = \sum [(V_{ms} \times R_{ms} + V_{sm} \times R_{sm}) \times (\Delta X_{ms} + \Delta Y_{ms})] \tag{1}$$

Our strategy places the communicating tasks on the same PE. The bits can be easily exchanged through some common register or memory without needing communication channels that need lot of energy. The distance between source-destination pairs is also reduced as they become the same PE by placing the communicating task pairs on it. Thus, power dissipation is greatly reduced by reducing the energy consumed in communication.

Third graph in Figure 4 plots power required by NN and SNN heuristics at different communication rates for the evaluated application scenario. The power is clearly proportional to the rate. The SNN heuristic achieve an average gain of 48.38% over NN.

6. Conclusions

This paper describes an efficient run-time mapping technique for MPSoC platforms consisting of multi-task supported PEs. The technique tries to map maximum communicating task pairs of an application on the same PE to reduce the communication overhead. The communicating tasks mapped on the same PE can communicate faster as they don't need to communicate through channels of the NoC. The performance of the technique is evaluated to map the applications onto an 8×8 NoC-based MPSoC. Evaluated performance metrics total execution time, average channel load and power dissipation show significant improvement as they heavily depend on the communication overhead. Our future scope includes evaluation of real-time benchmarks on the MPSoC platform and to incorporate task migration when a performance bottleneck is detected in order to improve the performance.

References

[1] J. Henkel, W. Wolf, S. Chakradhar, On-chip networks: A scalable, communication-centric embedded system design paradigm, in: VLSID '04: Proceedings of the 17th International Conference on VLSI Design, IEEE Computer Society, Washington, DC, USA, 2004, p. 845.

[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, N. Borkar, An 80-tile 1.28tflops network-on-chip in 65nm cmos, in: Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International, 2007, pp. 98–589. doi:10.1109/ISSCC.2007.373606.

[3] First 100-core processor with the new tile-gx family, http://www.tilera.com/news_&_events/press_release_091026.php (Oct, 2009).

[4] M. Kistler, M. Perrone, F. Petrini, Cell multiprocessor communication network: Built for speed, IEEE Micro 26 (3) (2006) 10–23. doi:http://dx.doi.org/10.1109/MM.2006.49.

[5] S. Borkar, Thousand core chips: a technology perspective, in: DAC '07: Proceedings of the 44th annual Design Automation Conference, ACM, New York, NY, USA, 2007, pp. 746–749. doi:http://doi.acm.org/10.1145/1278480.1278667.

[6] D. Wu, B. M. Al-Hashimi, P. Eles, Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems, in: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, IEEE Computer Society, Washington, DC, USA, 2003, pp. 90–95.

[7] S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, A methodology for mapping multiple use-cases onto networks on chips, in: DATE '06: Proceedings of the conference on Design, automation and test in Europe, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006, pp. 118–123.

[8] H. Orsila, T. Kangas, E. Salminen, T. D. Hämäläinen, M. Hännikäinen, Automated memory-aware application distribution for multi-processor system-on-chips, J. Syst. Archit. 53 (11) (2007) 795–815. doi:http://dx.doi.org/10.1016/j.sysarc.2007.01.013.

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

[10] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, H. Corporaal, Run-time management of a mpsoc containing fpga fabric tiles, IEEE Trans. Very Large Scale Integr. Syst. 16 (1) (2008) 24–33. doi:http://dx.doi.org/10.1109/TVLSI.2007.912097.

[11] L. Smit, G. Smit, J. Hurink, H. Broersma, D. Paulusma, P. Wolkotte, Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture, in: Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on, 2004, pp. 421–424. doi:10.1109/FPT.2004.1393315.

[12] M. A. Al Faruque, R. Krist, J. Henkel, Adam: run-time agent-based distributed application mapping for on-chip communication, in: DAC '08: Proceedings of the 45th annual Design Automation Conference, ACM, New York, NY, USA, 2008, pp. 760–765. doi:http://doi.acm.org/10.1145/1391469.1391664.

[13] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, G. J. M. Smit, Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc), in: DATE '08: Proceedings of the conference on Design, automation and test in Europe, ACM, New York, NY, USA, 2008, pp. 212–217. doi:http://doi.acm.org/10.1145/1403375.1403427.

[14] E. W. Bri ao, D. Barcelos, F. R. Wagner, Dynamic task allocation strategies in mpsoc for soft real-time applications, in: DATE '08: Proceedings of the conference on Design, automation and test in Europe, ACM, New York, NY, USA, 2008, pp. 1386–1389. doi:http://doi.acm.org/10.1145/1403375.1403709.

[15] A. Mehran, A. Khademzadeh, S. Saeidi, DSM: A Heuristic Dynamic Spiral Mapping algorithm for network on chip, IEICE Electronics Express 5 (13) (2008) 464–471.

[16] E. Carvalho, F. Moraes, Congestion-aware task mapping in heterogeneous mpsocs, in: System-on-Chip, 2008. SOC 2008. International Symposium on, 2008, pp. 1–4. doi:10.1109/ISSOC.2008.4694878.

[17] A. K. Singh, W. Jigang, A. Prakash, T. Srikanthan, Efficient heuristics for minimizing communication overhead in noc-based heterogeneous mpsoc platforms, in: RSP '09: Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping, IEEE Computer Society, Washington, DC, USA, 2009, pp. 55–60. doi:http://dx.doi.org/10.1109/RSP.2009.18.

[18] F. Moraes, N. Calazans, A. Mello, L. Möller, L. Ost, Hermes: an infrastructure for low area overhead packet-switching networks on chip, Integr. VLSI J. 38 (1) (2004) 69–93. doi:http://dx.doi.org/10.1016/j.vlsi.2004.03.003.