Efficient Task Mapping in Multi-tasking Heterogeneous MPSoC Platforms

Amit Kumar Singh, Wu Jigang, Alok Prakash, Thambipillai Srikanthan, Douglas Maskell
Centre for High Performance Embedded Systems, Nanyang Technological University, Singapore
{amit0011, asjgwu, alok0001, astsrikan, asdouglas}@ ntu.edu.sg

Abstract - Real-time requirements of applications can limit the number of tasks executing in a Multiprocessor System-on-Chip (MPSoC) platform. This demand for efficient mapping algorithm that is capable of maximizing computation performance. This paper introduces a multi-tasking heterogeneous MPSoC platform along with two run-time mapping algorithms to maximize the performance by minimizing the communication overhead. The proposed algorithms have been shown to be capable of upholding the proximity of tasks in order to reduce the communication overhead by mapping the adjacent hardware tasks onto the same reconfigurable processing element whenever possible. The proposed approach has been shown to consistently alleviate Network-on-Chip (NoC) congestion bottlenecks, to maximize the performance. Based on our evaluations, we show that the new mapping algorithms are capable of reducing average channel load, total execution time and latency when compared to state-of the-art run-time mapping techniques reported in the literature.

Keywords: ***Multiprocessor System-on-Chip (MPSoC) design, Hardware-Software Co-design, Heterogeneous Architectures, Network-on-Chip (NoC), Mapping Algorithms***

I Introduction

The computation complexity of embedded applications and their performance requirements have increased significantly and these can no more be supported by a single general purpose processor, inevitably requiring high performance computing platforms. These platforms can be created by integrating several embedded processors on a single chip towards the development of a Multiprocessor System-on-Chip (MPSoC). MPSoC platforms [1] [2] are solutions to these complex applications to meet the ever rising performance requirements.

The advancement in nanotechnology will enable to integrate thousands of processing elements in a single die by 2015 [3]. In order to achieve high computation performance along with energy efficiency, the MPSoCs need to contain several type of processing elements, ranging from instruction set processors to application specific architectures [4].

The PEs present in the MPSoC platform demands a communication infrastructure in order to have proper communication amongst different processing elements in the platform. Network-on-Chip (NoC) [5] is used as the communication infrastructure as they are highly scalable and support parallelism unlike buses and point-to-point communication. NoCs can integrate several type of processing elements like general purpose processors, specialized processing elements such as Digital Signal Processors, FPGA fabric tiles, dedicated IP-cores and specialized memories on a single chip towards the development of a MPSoC [4] [6].

Today's complex applications need to be supported by MPSoC platforms containing different type of processing elements (PEs). Homogeneous MPSoCs [7] [8] composed of identical PEs can't support many applications whereas, heterogeneous MPSoCs [4] [6], composed of different type of PEs support wide variety of applications. Heterogeneous MPSoCs are solutions for most of the applications to meet the performance constraints.

Mapping applications onto MPSoC platform can be accomplished at either design-time or run-time. Design-time (Static) mapping techniques proposed in [9] [10] [11] find best placement of the tasks at design-time with a well known computation and communication behavior. They are not suitable for dynamic workloads where number of simultaneously running tasks may exceed the available platform resources. Run-time mapping techniques are required for dynamic workloads [12]. Task migration [13] is also used for run-time application mapping by inserting a new task into the system at run-time. At run-time, when a performance bottleneck is detected, the task is migrated from one PE to another, to distribute the workload more homogeneously, in order to improve the performance.

This work presents a multitasking NoC-based heterogeneous MPSoC platform and two new run-time mapping heuristics based on our strategy called as packing strategy, for efficiently mapping applications onto the MPSoC platforms. Proposed mapping heuristics maps the tasks of an application in a systematic manner by clearly choosing a PE for a task and is described in detail in section IV. The performance of the mapping heuristics is evaluated for dynamic workloads. State-of-the-art run-time mapping heuristics presented in [21] consider single task supported processing elements (PEs) in the platform and do not perform well when applied to different scenarios. The heuristics developed by our packing strategy consider multitasking resources in the MPSoC platform and give better performance compared to state-of-the-art mapping heuristics in [21]. Our MPSoC platform consists of hardware/software resources to execute hardware/software tasks. Hardware resources (*Reconfigurable Area*) are considered large enough to support more than one task in parallel. On the other hand software resources are considered to support only one task so as to avoid any memory bottlenecks. Memory space is required to store the software's (software tasks). At run-time adjacent (communicating) hardware tasks of an application get mapped on same hardware node, reducing the communication overhead between the tasks. Additionally, our strategy maps the communicating tasks close to each other so as to further minimize the communication overhead. We have evaluated our mapping heuristics and state-of-the-art run-time mapping heuristics presented in [21]

for our target MPSoC architecture and performance matrices such as average channel load, total execution time and latency are compared in order to show the performance improvements.

The rest of the paper is organized as follows: Section II presents the related work. Section III describes the MPSoC architecture. In Section IV, we present our packing strategy. Section V describes the mapping algorithms. Experimental setup and the results are presented in Section VI with Section VII concluding the paper along with future directions.

II. Related Works

The academia and companies have developed several MPSoCs and many are on the way to be released [1]. These MPSoCs are customized for different target market like general purpose computing, scientific computing and embedded computing according to their computation requirement.

Most of the work in literature present static mapping techniques [9] [10] [11] in order to map applications onto the MPSoC platforms but these are not adequate for an adaptive system that changes its behavior over time. These mapping techniques consider fix computation and communication behavior of a task to find its best possible placement. Run-time mapping techniques are required for adaptive systems that contain dynamic workload.

Nollet et al. [14] present a run-time task assignment heuristic that performs fast and efficient task assignment in an MPSoC. The MPSoC platform contains FPGA fabric tiles along with other processing elements. The FPGA fabric tiles facilitate configuration hierarchy that improves the task assignment success rate and quality. Theocharides et al. [15] demonstrate a run-time, system-level bidding-based task allocation strategy for generic MPSoC architectures. The allocation strategy gives significant performance improvements when compared to a round robin allocation, in popular MPSoC applications. Briao et al. [16] 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. In order to save energy, the system turns off idle processors and applies Dynamic Voltage Scaling to processors with slack. Smit et al. [17] present multi-core architectures and run-time mapping of streaming applications onto these architectures. In [4], authors demonstrate a run-time task assignment algorithm to map a task-graph onto an MPSoC platform. The algorithm maps a task before all other task that needs a scarce resource, by taking availability of resources into account. Holzenspies et al. [18] describe a run-time spatial mapping technique consisting of four steps to map the streaming applications onto a heterogeneous MPSoC. The mapping solutions found are verifiable for feasibility. The algorithm is implemented onto an ARM926 processor running at 100 MHz and it takes less than 4 ms to run the HIPERLAN/2 example. Ngouangal et al. [19] describe a mapping technique based on the attraction forces between communicating tasks that tries to place them close to each other. While mapping tasks at run-time, the performance requirements, the number of available NPUs (Network Processing Units) and their respective positions are taken into account. Chou et al. [20] propose a run-time mapping strategy for allocating the application tasks to platform resources in homogeneous MPSoCs. The user behavior information is incorporated in the resource allocation process; that allows system to better respond to real-time changes and adapt dynamically to user needs. By considering user behavior, 60% communication energy is saved compared to an arbitrary task allocation strategy. Carvalho et al. [21] present heuristics for run-time mapping of tasks in NoC-based heterogeneous MPSoCs where tasks are mapped on the fly, according to the communication requests and the load in the NoC links.

III. Target MPSoC Architecture

Our MPSoC platform contains a set of different PEs which interacts via a communication network [22] as shown in Fig. 1. PEs can support either a hardware or software tasks. Software tasks execute in instruction set processor (*ISP)* and hardware tasks execute in reconfigurable logic blocks (*RA*) or in dedicated *IP*-cores. The *RAs* are considered large enough to support more than one hardware tasks in parallel. The *ISPs* are considered to support a single software task to avoid the memory constraint issue. For inter-task communication, message passing protocol is used by the communication network as described in [14]. XY routing algorithm is used to transmit and receive the messages. In XY routing, first, the packet is transferred in X-direction then in Y-direction for transferring packets from one processing node to another processing node.

In the MPSoC platform, one of the *ISP* is used as the Manager Processor (M) that is responsible for *task mapping*, *task scheduling*, *resource control* and *configuration control*. The configuration overhead results are used to simulate the *configuration control* process [23].

The initial tasks (starting tasks) for each application are started by M and new unmapped communicating tasks are loaded into the MPSoC at run-time from the *task memory* when a communication to them is required.

Task mapping involves *task binding* and *task placement*. In *task binding*, a decision is made whether the task will be mapped on an *ISP* (software task) or on a *RA/IP* (hardware task). As there are many *ISPs* and *RAs* in the platform, so in *task placement* real position for the task to be mapped is found. For *resource control*, the resources status is updated at

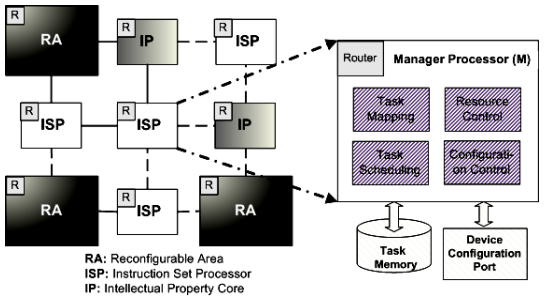

Fig. 1. Target MPSoC Architecture

run-time to provide the M with an accurate information about the resource occupancy as the mapping decision is taken based on the PEs and NoC use. *Task scheduling* is implemented by the queue strategy and there are three queuesfor hardware, software and initial tasks. A task enters into its corresponding queue if there is no free resource in the platform and waits until a resource of same type is not free.

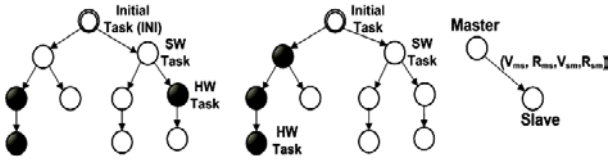

Fig. 2. Applications modeling (a), (b) and Master-Slave pair (c)

IV. Proposed Packing Strategy

The proposed packing strategy for mapping the applications onto the MPSoC architectures is described in this section. The *application*, *MPSoC architecture* and *mapping* are defined subsequently in order to explain the strategy.

Application is modeled by acyclic directed graph *TG (Task Graph) = (T, E),* where *T* is set of tasks in the application and *E* is the set of edges in the application as in Fig. 2 (a) and (b) similar to [21]. Each application has one initial (starting) task and various hardware/software (HW/SW) tasks. A connection (edge) between two tasks defines master-slave pair as in Fig 2 (c), i.e., a connection contains master and salve tasks. The initial task has no master. Each edge in *E* contains four parameters $(V_{ms}, R_{ms}, V_{sm}, R_{sm})$ as in Fig 2 (c), where V_{ms} and R_{ms} represent the *volume (V)* and *rate(R)* of data to send from master to salve (*ms*) and V_{sm} and R_{sm} represent the *volume (V)* and *rate(R)* of data to send from slave to master (*sm*).

MPSoC architecture is a directed graph *AG (Architecture Graph) = (P, L),* where *P* is the set of tiles p_i and $l_{i,j} \in L$ presents the physical link between two tiles p_i and p_j.

Mapping of *applications'* tasks onto the *MPSoC architecture* is represented as *mpg*: $t_i (\in T) \mid\rightarrow p_i (\in P)$

State-of-the-art run-time mapping heuristics Nearest Neighbor (NN) and Best Neighbor (BN) in [21] tries to map a slave task near the node at which master task is mapped, i.e., at a neighboring node. These heuristics do not consider the node's position while finding placement of the slave task. This results in higher communication overhead as the slave tasks from different applications gets mapped to any of the neighbor node of their masters. In our packing strategy, we systematically chose a neighboring node for the task mapping that reduces the communication overhead. Also, by incorporating multi-tasking hardware resources in the MPSoC platform, the communication overhead is further reduced as adjacent communicating hardware tasks gets mapped onto the same processing node.

In our packing strategy, the communicating tasks of an application are mapped close to each other in a systematic manner so that they need not to communicate from distance apart in order to avoid the communication overhead. Here, the demonstration is done with nine applications at a time to be mapped onto the MPSoC platform. To map applications, firstly, clusters are found for each application and initial tasks are placed at the centre of the clusters (as shown in Fig. 3). All the tasks belonging to an application are tried to map within a particular cluster in which its initial task is mapped. The cluster boundaries are virtual so more than one application can share some parts of a given region. In our approach, these clusters are chosen in a distributed manner so that communicating tasks for each application (here 9 applications) will be mapped in close proximity within a particular cluster. Thus, there will be minimum interference between two applications resulting in reduced channel congestion. If initial tasks are not mapped in a distributed manner, i.e., are mapped randomly then they might get placed very close to each other. Now, when rest of the tasks of each application are mapped, the applications need to share the same NoC region that will result in longer waiting time for a resource to become free for task mapping, and increased channel congestion.

For each application (here 9 applications), after the initial tasks get mapped and start their execution, new communicating tasks are mapped according to the communication request. To map a requested task, firstly, the task is tried to be mapped at the same processing node making the request as hardware resources support more than one task. A task is supported by a processing element (PE) if the type of PE is same as the type of the task and there is a free resource at the PE location. If task is not supported by the node making the request, then it is tried to be mapped at hop distance of one. If task is not supported at hop distance one then it is tried to be mapped at hop distance of two and so on. The attempt goes up to the NoC limit. At each hop distance, first, left or down side PE around the node making the request (master task position) is selected to map the task, so that the PEs on the top-right edge of the cluster can be used by tasks of other applications that are also getting mapped in the similar manner. Thus, *resource utilization* is increased. This strategy is applied to each application to be mapped and most of the applications get the free resources from other application's top-right edge of the cluster. If neither left nor down side processing element is able to execute the task only, then the task is tried to be mapped onto the top or right side PE.

This strategy maps the communicating tasks of each application in close proximity, thus reducing the communication overhead. Additionally, by incorporating multi-tasking hardware resources, adjacent communicating hardware tasks are mapped onto the same processing node, resulting in further reduction of communication overhead and more compact mapping. *Average channel load* is reduced as it directly depends on the congestion in the channel and communication overhead. *Total execution time* mainly depends on computation time and communication time. Communication time is reduced by reducing the communication overhead. The search space to find a supported node for requested task is reduced by searching in a systematic manner as explained above, resulting in reduced

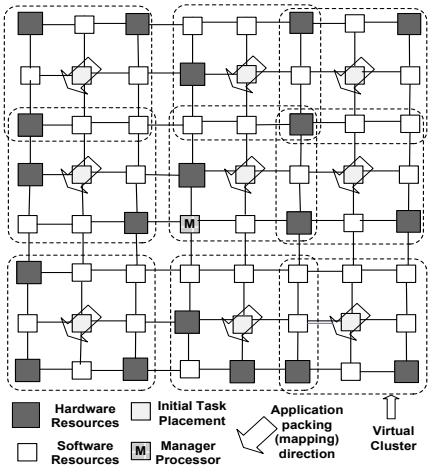

Fig. 3. Initial tasks placement for mapping applications

placement time. Thus, *total execution time* is also reduced. *Average packet latency* depends on the distance between source and destination PE and the congestion in the communication path. Distance between source and destination PE is reduced by mapping the communicating tasks close to each other and channel congestion is also reduced (explained above), thus *average packet latency* is reduced.

V. Runtime Mapping Algorithms

A. First Run-Time Mapping Algorithm (RTM1)

This algorithm is based on the packing strategy (discussed in section III) along with the search space (circular search space) of Nearest Neighbor (NN) heuristic in [21].

To map multiple applications onto the MPSoC platform, Algorithm RTM1 is followed by each application that is getting mapped. First, initial tasks for each application are mapped after finding a suitable cluster (step 1 & 2) in a distributed manner as shown in Fig. 3. Clusters are chosen in distributed manner to reduce the channel congestion and waiting time to map a task as discussed in section IV. The M knows only the initial tasks. When initial tasks start their execution, communication requests are sent to the M to map the slave tasks at run-time.

The run-time requested task (step 3) is mapped for sure if there is any free supported PE (step 4) in the NoC. The task is first tried to map at the same PE (step 5 & 6) on which its master is mapped. If the same PE can't support the requested task then it is tried to map at hop distance of one. If there is no supported resource at hop distance one then it is tried to map at hop distance of two and so on. The search space goes up to the NoC limit (step 9). At each hop distance first left and down side PE(s) (step 10) are selected to map the task and if none of them support the task then only the task is tried to be mapped on right or upside PE(s) (step 16).

After mapping the task on a PE, the resources are updated and the M waits for the next requested task (step 13 & 14). If there is no free supported resource in the platform (step 19) i. e. number of tasks exceeds the platform resources then the

Algorithm RTM1 /* First algorithm for Run-Time Mapping*/

Input: *TG(T,E)*, *AG(P,V)* /* task $t_i \in T$; processing element $p_i \in P$*/
Output: *mpg* (mapping *TG(T,E)* → *AG(P,V)*)
1) Find a suitable cluster for the application (from Fig. 3);
2) Map the initial task (INI$\in T$) at the centre of the cluster;
3) **for** each unmapped task t_i that is requested **do**
4) **if** (there is free supported resource in the platform)
5) **if** (resource free at master task position)
6) Map the task and update *mpg* & resources**;**
7) wait and go back to step 4 if new task is requested;
8) **else**
9) **for** hop_distance = 1 to NoC limit **do**
10) Select left and down side node(s) for mapping;
11) **if** (node(s) support the task)
12) Select first free supported node $p_i \in P$ to map t_i;
13) insert(p_i to *mpg*); update(resources by *mpg*);
14) **wait** and go back to step 4 if new task is requested;
15) **else**
16) Select right and up side node(s) for mapping;
17) Do similar steps as described in steps 11 to 14;
18) **end for**
19) **else**
20) Insert (t_i, to corresponding Queue) ;
21) **wait** for a free supported resource p_i in the platform;
22) **release**(t_i from corresponding Queue) & map onto freed p_i;
23) insert(p_i to *mpg*); update(resources by *mpg*);
24) **wait** and start from step 3 if new task t_i is requested;
25) **end for**

requested task is entered into its corresponding queue (step 20) and waits until a supported resource is not freed (step 21). A resource becomes free when the task mapped on it finishes its execution. After a resource becomes free, platform resources are updated and a supported queued task is released from the queue (step 22) and its mapping position is found in order to map it. The same process (step 3 to 23) is repeated for each unmapped task that is requested in order to map all the tasks of each application.

B. Second Run-Time Mapping Algorithm (RTM2)

The first algorithm (RTM1) looks for only a free supported resource and does not consider the load in channels while mapping a task. Thus, RTM1 is not a congestion aware mapping. We incorporate path load computation approach in RTM1, to get a congestion aware mapping algorithm (RTM2). For each mapping *m*, the path load is computed from Equation (1), where $r_{chl(i,j)}$ and $r_{chl(j,i)}$ are the rates in the individual channels, from the master task position to the salve task position (task tried to map) and the rates in the channels in opposite direction.

$$\cos t_m = \sum r_{chl(i,j)} + \sum r_{chl(j,i)} \qquad (1)$$

Algorithm RTM2 /*Second algorithm for Run-Time Mapping*/

In algorithm RTM1, path load computation is incorporated by replacing the line (12) by:

- Calculate path load for node(s) (from equation 1)
- Select node p_i with minimum path load

The rest of the lines remain same as in algorithm RTM1.

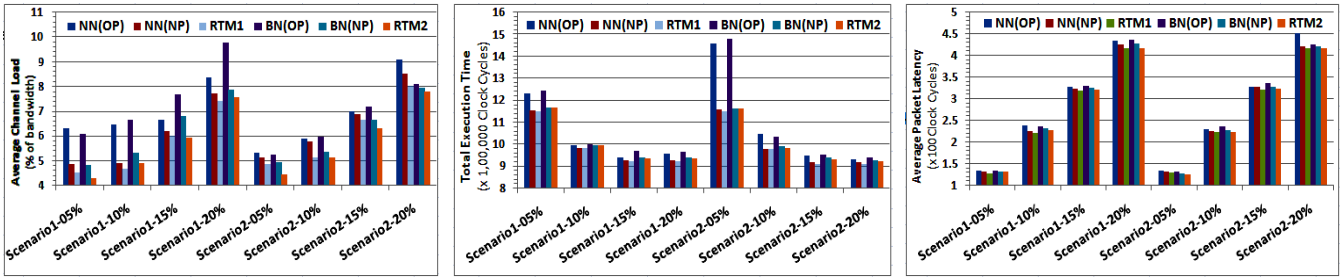

Fig. 4. Graphs showing Average Channel Load, Total Execution Time and Average Packet Latency for two simulated scenarios (Scenario1 and Scenario 2) for different communication rate (% of available bandwidth)

Table 1: Summary of the improvements (%) on new platform (NP) for older heuristics (NN & BN) and our proposed heuristics (RTM1 & RTM2) based on packing strategy. Improvements with respect to NN and BN on older platform (OP) for two simulated scenarios are shown.

Performance Metric	Scenario 1				Scenario 2			
	NN(NP) (% improvement w.r.t. NN(OP))	RTM1 (% improvement w.r.t. NN(OP))	BN(NP) (% improvement w.r.t. BN(OP))	RTM2 (% improvement w.r.t. BN(OP))	NN(NP) (% improvement w.r.t. NN(OP))	RTM1 (% improvement w.r.t. NN(OP))	BN(NP) (% improvement w.r.t. BN(OP))	RTM2 (% improvement w.r.t. BN(OP))
Average Channel Load	14.70	18.59	17.64	24.67	3.38	9.69	6.04	10.57
Total Execution Time	3.24	3.41	3.24	3.44	9.38	9.96	8.71	9.30
Average Packet latency	2.64	4.23	1.76	3.60	3.58	4.80	2.12	3.45

Here, path load for each supported node is calculated and the node with minimum path load is selected to map the requested task. By considering path load (congestion in channels) while mapping a task the channel load is distributed in the whole NoC and communication overhead gets reduced.

Our analysis in time complexity shows that RTM1, RTM2 and heuristics NN, BN have time complexity of same level. The mapping complexity of above four algorithms is O(C), where C is the number of processing elements in the NoC.

VI. Experimental Setup and Results

Experiments are performed by *ModelSim co-simulation* (*System-C* for applications and *RTL-VHDL* for the NoC).

A. Experimental Setup

The simulation platform used is an extended version of the simulation platform used in [21]. Each application is modeled as in Fig. 2, with an initial task and HW/SW tasks. The task processing time is fixed. Each task transmits from 200 to 400 packets with size varying from 100 to 400 16-bit flits.

The NoC is modeled in VHDL [22], in an 8×8 2D-mesh topology (Fig. 3) and is responsible for data transfer between the tasks. One node is used as manager processor (M), 44 nodes as SW resources and remaining 19 nodes contain large enough reconfigurable area that can support three HW tasks in parallel. For realistic scenarios, hardware tasks are limited to three at one node position by considering the area constraint of the *reconfigurable* tiles and software tasks are limited to one by considering the memory space constraint that is required to store the software. Processing nodes are modeled by two *SystemC-Cthreads*, one for the M and the other for the remaining Processing nodes.

The evaluated scenarios are: (*i*) applications having two adjacent hardware tasks (Fig. 2(a)); and (*ii*) applications having three adjacent hardware tasks (Fig. 2(b)). In both the scenarios, 20 identical applications, each one with 10 tasks and injection rate varying from 5 to 20% of available bandwidth, are considered.

The number of applications executed at a time is limited to 9 (each 10 tasks) by dividing the NoC into 9 clusters. If less than 9 clusters are taken then resources are underutilized and if more than 9 are taken then tasks need to wait in queue for more time to get a free resource. The experiment is performed by varying the number of clusters and best performance is obtained by considering 9 clusters (Fig. 3).

B. Experimental Results

Algorithms are evaluated for *average channel load*, *total execution time* and *average packet latency*. The proposed RTM1 and RTM2 run on new platform, while the latest approaches Nearest Neighbor (NN) and Best Neighbor (BN) presented in [21] run on the older platform (OP) and on new platform (NP) as well, denoted as NN(OP), NN(NP), BN(OP) and BN(NP) respectively. As explained before, in older platform each PE supports only one task and in new platform hardware PEs are modeled to support more than one task. The evaluated results for two simulated scenarios are shown in Fig. 4. In scenario 1, two adjacent HW tasks get mapped on one node and in scenario 2; three adjacent HW tasks get mapped on one node as the applications in scenario1 and scenario 2 contain two and three adjacent hardware tasks respectively.

Average channel load reflects the NoC use. Our packing strategy reduces the communication overhead by mapping the tasks of an application in close proximity in a very systematic manner. Additionally, in the new platform, the adjacent hardware tasks get mapped on same node, resulting in further reduction in the communication overhead. Thus channel load is reduced as it depends directly on communication overhead. The improvements of heuristics NN, BN, RTM1 and RTM2 for two simulated scenarios are shown in table 1. We get maximum improvement of 24.67% in scenario 1 for RTM2 with respect to BN on older platform (OP).

Total execution time is comprised of placement time, configuration time, communication time, waiting time (when no free resource in the platform) and computation time

amongst which communication time dominates. Our packing strategy along with multi-tasking hardware resources in the platform, successfully reduce communication overhead and thus the communication time. The placement time is also reduced (explained in section IV). Thus, total execution time is reduced when compared to runtime mapping heuristics in [21]. The improvements (%) are shown in table 1. An improvement of 9.96% in scenario 2 for RTM1 with respect to NN on older platform (OP) was found, while all the heuristics have similar execution time, as shown in Fig. 4, due to similar time complexity.

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. It is also successfully reduced as our packing strategy maps the tasks of an application close to each other reducing the distance between source and destination PE. Also, multi-tasking resources minimize the congestion and the distance between source and destination PE by mapping the adjacent HWs on same node. The improvements are shown in table 1.

VII. Conclusion

We have proposed novel packing strategies for the runtime mapping of applications onto an 8 × 8 multi-tasking NoC-based heterogeneous MPSoC platform. It allows for the hardware resources (Reconfigurable Logic) to support multiple tasks on the same hardware element in order to eliminate communication overhead while upholding concurrent processing. First algorithm tries to map the tasks of an application in close proximity, reducing the communication overhead between the communicating tasks. The second algorithm considers traffic in addition to the proximity of tasks while mapping, resulting in more uniformly distributed channel load. Our mapping algorithms reduce the channel load, execution time and latency. The improvements are clearly enunciated in the experiments and results section. In addition, we have shown that the state-of-the-art run-time mapping heuristics achieve performance improvement when evaluated on the proposed platform. Current MPSoC platform is limited to running one task on each software element while the hardware element can support multiple tasks. Work is underway to ensure that each CPU (software resource) and hardware element of MPSoC can operate as multitasking nodes so as to facilitate further improvements to overall compute performance.

Acknowledgements

We thank Mr. Ewerson Carvalho (first author of the paper [21]) for providing us the basic simulation environment and helping us to expand it for the simulation of our ideas.

References

[1] http://view.eecs.berkeley.edu/wiki/Chip_Multi_Processor_Watch, 2008.
[2] Tilera Corporation. TILE64™ Processor. Product Brief, 2007.
[3] Borkar, S. Thousand Core Chips-A Technology Perspective. In *Proc. of DAC*, 2007
[4] Smit, L.; Smit, G.; Hurink, J.; Broersma, H.; Paulusma, D.; Wolkotte, P. Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In *Proc. of FPL*, 2004.
[5] Benini, L. and Micheli, G. Networks on Chips: A new SoC paradigm. IEEE Computer, v.35(1), 2002.
[6] Nollet, V.; Marescaux, T.; Avasare, P.; Mignolet, J-Y. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In *Proc. of DATE*, 2005.
[7] Vangal, S. ; Howard, J. ; Ruhl, G. ; Dighe, S. ; Wilson, H. ; Tschanz, J. ; Finan, D. ; Iyer, P. ; Singh, A. ; Jacob, T. ; Jain, S. ; Venkataraman, S. ; Hoskote, Y. ; Borkar, N. ; An 80-Tile 1.28TFLOPS Network-on-Chip in 65 nm CMOS. In *Proc. of ISSCC*, 2007.
[8] Lin, L. ; Wang, C. ; Huang, P. ; Chou, C. ; Jou, J. Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. *ASPDAC*, 2005.
[9] Hu, J.; Marculescu, R. Energy- and Performance-Aware Mapping for Regular NoC Architectures. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, v.24(4), 2005.
[10] Murali, S.; Coenen, M.; Radulescu, A.; Goossens, K.; De Micheli, G. A methodology for mapping multiple use-cases onto networks-on-chip. In *Proc. of DATE*, 2006
[11] Ruggiero, M.; Guerri, A.; Bertozzi, D.; Poletti, F.; Milano, M. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-onchip. In *Proc. of DATE*, 2006.
[12] Wronski, F.; Brião, F.; Wagner, R. Evaluating Energy-aware Task Allocation Strategies for MPSoCs. *DIPES*, 2006.
[13] Bertozzi, S.; Acquaviva, A.; Bertozzi, D.; Poggiali, A. Supporting task migration in multiprocessor systems-on-chip; a feasibility study. In *Proc. of DATE*, 2006.
[14] Nollet, V.; Avasare, P.; Eeckhaut, H.; Verkest, D.; Corporaal, H. Run-time Management of a MPSoC Containing FPGA Fabric Tiles. IEEE Transactions on Very Large Scale Integration Systems, Vol. 16, No. 1, 2008.
[15] Theocharides, T.; Michael, M. K.; Polycarpou, M.; Dingankar, A. Towards Embedded Runtime System Level Optimization for MPSoCs: On-Chip Task Allocation. In *Proc. of Great Lakes Symposium on VLSI*, 2009.
[16] Briao, E. W.; Barcelos, D.; Wagner, F. R. Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications. In *Proc. of DATE*, 2008.
[17] Smit, G.; Kokkeler, A.; Wolkotte, P.; Burgwal, M. Multi-core Architectures and Streaming Applications. *SLIP*, 2008.
[18] Holzenspies, P.; Hurink, J..; Kuper, J.; Smit G. Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSoC). *DATE*, 2008.
[19] Ngouangal, A.; Sassatelli, G.; Torres, L.; Gil, T.; Soares, A.; Susin, A. A contextual resources use: a proof of concept through the APACHES platform. In *Proc. of DDECS*, 2006.
[20] Chou, C-L.; Marculescu, R. User-Aware Dynamic Task Allocation in Networks-on-Chip. In *Proc. of DATE*, 2008
[21] Carvalho, E.; Moraes, F. Congestion-aware task mapping in heterogeneous MPSoCs. In *Proc. of SoC*, 2008
[22] Moraes, F.; Calazans, N.; Mello, A.; Moller, L.; Ost, L. Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. Integration, the VLSI Journal, Vol 38-1, 2004
[23] Moller, L.; et al. A NoC-based Infrastructure to Enable Dynamic Self Reconfigurable Systems. In *Proc. of ReCoSoC*, 2007.