Computation and Communication Aware Run-Time Mapping for NoC-based MPSoC Platforms

Samarth Kaushik, Amit Kumar Singh, Thambipillai Srikanthan
Centre for High Performance Embedded Systems, Nanyang Technological University, Singapore
{samarth2, amit0011, astsrikan}@ ntu.edu.sg

Abstract

Design-time strategies are suited only for mapping predefined set of applications and thus cannot predict dynamic behavior incurred due to the target applications and state of the platform at run-time. This dynamism demands run-time mapping of application tasks to maintain a critical balance between performance and resource optimization. Any disturbance may either lead to digression from expected performance or complete drain of valuable resources. So, it becomes mandatory to devise algorithms which can intelligently distribute the application tasks among processors taking communication overhead, computation load and resource utilization in consideration. This paper proposes a heuristic that illustrates how to incorporate these factors for mapping multiple tasks onto MPSoC platforms, where communication takes place through a Network-on-Chip. The heuristic attempts to balance the processing load on the platform processing elements (PEs) while reducing communication overhead by mapping highly communicating tasks on the same PE. For MPEG-4 application, the proposed technique reduces total execution time by 33%, resource usage by 37% and energy consumption by 40% when compared to state-of-the-art run-time mapping techniques.

I. INTRODUCTION

System-on-Chip (SoC) design is experiencing a radical shift from uni-processor architecture to multi-processor architecture in order to accomplish the ever increasing demand for high performance. The rising complexity of real-life applications cannot be addressed by simply trying to make single-core processors run faster, instead it requires multiple processors which can cohesively communicate and provide increased parallelism [1]. The underline concept is to consider an application as a conglomeration of many parallelized small tasks which can be uniformly distributed among multiple processors in order to accomplish high computation performance and energy efficiency.

To support multimedia applications, i.e., video decoders, 3D games etc, which can be represented as a cluster of many small tasks executing in parallel, a Multiprocessor System-On-Chip (MPSoC) architecture is needed. The MPSoC consists of multiple processing elements (PEs). These PEs are interconnected by mesh of configurable on-chip connections called Network-on-Chip (NoC) that enables flexibility, power efficiency and scalability over conventional bus based infrastructures [2, 3]. It employs packet switching approach for communicating data, i.e., packets among interconnected PEs are transferred through multiple point-to-point data links controlled by switches. The reported literature provides several MPSoC architecture models [17, 18, 19]. In [19], 64 homogeneous general purpose processors arranged in 8x8 array, interconnected by 2D mesh network and running SMP Linux forms a multi-core SoC. A homogeneous MPSoC consists of identical PEs, whereas heterogeneous MPSoC consists of different types of PEs, i.e., Digital Signal Processors (DSPs), FPGA fabric tiles, dedicated intellectual property cores (IPs) etc., to achieve higher performance.

The next challenge is to map parallelized tasks of an application onto MPSoC platform, which entails a judicious mechanism of mapping these tasks on various PEs, either at design-time or at run-time. Numerous design-time mapping techniques have been developed but they are limited to predefined set of applications and are unaware of run-time resource management [18], whereas run-time mapping techniques can be employed to large number of applications and incorporate run-time resource management. Most of the existing run-time mapping techniques attempt to reduce communication latency and energy consumption. In [20], authors propose a force directed algorithm that reduces communication overhead by mapping communicating tasks in close vicinity in the platform. Run-time mapping heuristics reported in literature by Singh et al. [5] reduce communication overhead by mapping communicating tasks on the same PE. The mapping techniques reported in literature do not consider computation load balancing while reducing communication overhead. We present a run-time task mapping technique that reduces computation load variance and delineates substantial performance improvements along with efficient resource utilization.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 presents the problem definition, target architecture and pitfalls of existing mapping techniques. Section 4 describes our proposed mapping strategy. Experimental set-up and results are presented in Section 5. In Section 6, we conclude the paper and provide future directions.

II. RELATED WORK

Various design-time techniques for mapping multiple tasks on MPSoC platform have been proposed, for example, algorithms for NoC-based and bus-based MPSoCs are presented in [7], [8], [9]. Wu et al. [7] present a genetic algorithm based mapping heuristic to improve the energy efficiency of the system. Murali et al. [8] present strategy to map multiple use-cases on SoC using Tabu Search Algorithm. Marcon et al. [9] present Simulated Annealing technique for mapping multiple tasks on SoC. These mapping algorithms are not suitable for an adaptive system that changes its configuration over time and requires re-mapping of applications at run-time.

Faruque et al. [10] present a run-time agent based distributed application mapping technique for large MPSoCs such as 32×64 systems. The technique lowers

the communication traffic and computational effort.

Smit et al. [11] incorporate Min-Weight algorithm for run-time mapping of multiple tasks on heterogeneous processors to provide near optimal solution in a short computation time. To reduce time-complexity, the task graphs consist of few vertices or a large number of vertices with a degree of no more than two.

Chou et al. [12] propose an incremental technique for mapping run-time application onto an MPSoC architecture consisting of homogenous PEs operating at multiple voltage levels. The technique involves a global manager (GM) which is responsible for run-time mapping of incoming applications to the available PEs, managing the communication among them and system resource management.

Nollet et al. [13] present a hierarchical task assignment heuristic for mapping various tasks on an MPSoC containing FPGA fabric tiles. It employs more efficient usage of platform resource as a result of spatial task assignment. The FPGA tiles provide support for hierarchical configuration that enhances the quality of task assignment.

Holzenspies et al. [14] propose a run-time strategy for mapping inherently parallel streaming applications on MPSoC. The approach involves different phases of optimization which include design-time modeling, run-time feedback analysis and spatial mapping.

Ter et al. [15] present an incremental run-time mapping technique which spans both the task graph and the MPSoC platform to find optimal mapping of tasks. Resource allocation is divided into different phases of binding, mapping, routing, and validation.

Carvalho et al. [21] present a congestion-aware run-time task mapping heuristics, where the considered platform is capable of supporting only a single task at each PE. Singh et al. [5] describe a communication aware run-time mapping heuristic for MPSoC platforms accommodating multiple tasks on a single PE. The heuristic tries to minimize the communication overhead between two highly communicating tasks by mapping them on the same PE. It also reduces the average channel load and energy consumption as the tasks are mapped on the same PE and they don't need to communicate through channels of the NoC. However, the heuristic does not attempt to balance the computation load on each PE utilized for mapping and also involves a restricted approach for minimizing communication overhead.

III. PROBLEM DEFINITION

An application is modeled as a set of communicating parallel processes represented as a task graph [14]. The task graph is denoted as a directed graph $ATG = (T, E)$, 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. A task $t_i \in T$ is represented as (t_{id}, t_{comp}), where tid is the task identifier and tcomp is the task computation load in cycles. An edge $e_i \in E$ connecting the two tasks contains communication information between the tasks and is expressed as V_{ms} like in Figure 1, where V_{ms} represents the data volume for a single token to be sent from task m to s in terms of number of cycles taken for transfer when full channel bandwidth is available. The connected tasks are represented as master-slave pair. A master task in the task set T will be executed till last token is sent to its slave task. A slave task starts its execution after it receives a complete token from its master task. In Figure 1, T1 is master (m) and T2 is slave (s).

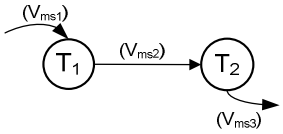


Figure 1: Tasks T_1 and T_2 of application and associated communication.

The MPSoC architecture is a graph $AG = (P, C)$, where P is the set of PEs identified by its identifier p_{id} and C represents the on chip communication channels for interconnecting the PEs. Each physical channel $c_{i,j} \in C$ 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 and is configured to have fixed amount of available memory. For evaluation, communicating tasks have been allocated full available channel bandwidth but it can be extended to varying channel bandwidth. Different NoC based 4×4, 5×5 and 6×6 PEs mesh arrangement have been employed. Among the available PEs, one is used as Manager Processor that 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 $mpg: t_i \in T \rightarrow p_i \in P$, which maps each 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 is shown in Figure 2, which is obtained by applying the mapping technique in [5]. It has been assumed that each PE has large memory capacity. First, the initial task 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 and mapping for the first requested task from a master is found and other requested tasks from the same master gets entered into an Application Queue (AQ). 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. After the mapped task starts executing, it's unmapped communicating (slave) tasks are requested. Similarly, the placement for the first requested task is found and others are placed in the same AQ. The same process continues until the communicating tasks from

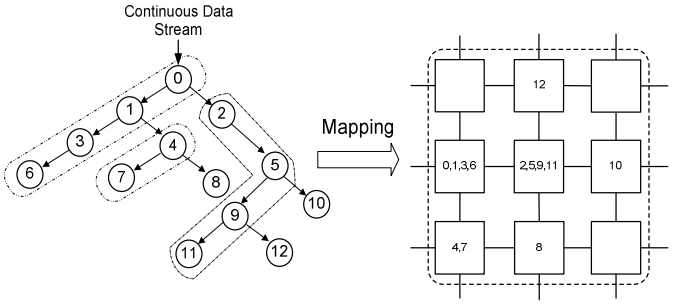


Figure 2: Mapping by existing heuristic proposed in [5]

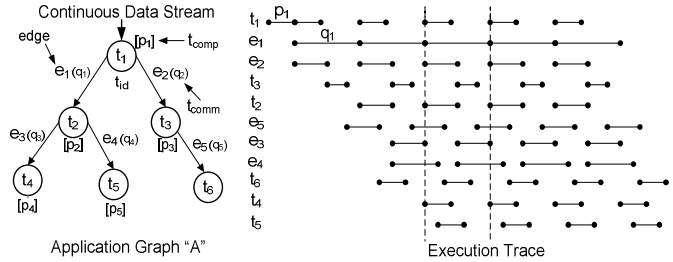


Figure 3: Execution Trace of Application Graph A

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 tasks is taken out to find its mapping on the nearest PE. The same process continues for the all 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 and the process is continued till the AQ is emptied. This technique has various drawbacks. Firstly, it does not try to balance the computation load on each PE which may lead to unbalanced execution on PEs. Secondly, it does not consider the amount of memory available on a PE before mapping a task on the PE. Lastly, it has a restricted approach rather than a global approach towards minimizing communication overhead as communication can be avoided only between the master and the requested slave task during mapping. The proposed heuristic takes these factors into consideration while mapping the application graph onto the MPSoC platform.

IV. PROPOSED ALGORITHM

Our technique performs pre-processing of the application graph before actual mapping is done in order to reduce the communication overhead and improve the load balancing on various PEs taking available memory on PEs into consideration.

A. Pre-Processing

In the proposed pre-processing technique, the application task graph is taken as an input and the technique tries to minimize communication latencies among various tasks while simultaneously trying to balance the processing load on various PEs. The scheme starts by targeting the communication intensive edges in the application and attempts to merge these highly communicating tasks on the same PE. The merging operation takes place only if memory constraint of the involved PE is satisfied, i.e., the PE must have sufficient memory to accommodate both the tasks and shared memory for their local communication. The shared memory is required by communication data on the edge of the connecting tasks. The proposed strategy forms a global approach as complete application graph is seen in entirety for removing communication bottlenecks, in contrast to mapping technique in [5] where merging of tasks on the same PE is possible only when communicating tasks are requested during execution. The main purpose of pre-processing is to remove any bottleneck that may arise due to overhead of transferring data among communicating tasks which can be understood by examining the execution trace of an application graph. In Figure 3, execution trace of application graph A is shown, where pi represents the trace for computation time of each task $t_i \in T_i$ and q_i represents the trace for communication time of each edge $e_i \in E_i$. This representation expresses the available parallelism that can be exploited to achieve high performance by removing the bottlenecks, for example, edge e_1 appears to be the main bottleneck in Figure 3. If the communicating tasks of edge e_1 are merged together on a single PE, then e_1 no longer remains the active bottleneck of the system and the whole execution trace will shrink leading to faster execution. This method of improvement relies on the fact that eliminating the slowest step in a pipeline will certainly lead to improved performance.

The proposed pre-processing scheme is decomposed into two phases and is presented through the Pre-Processing Algorithm (Algo 1):

In first phase (line 1 to 12), the strategy examines the complete application graph and attempts to remove the communication bottlenecks. The maximum computation load of a single task (max_p_{load}, line 2) and maximum communication load (max_c_{load}, line 3) between tasks is determined using the application graph. If the maximum communication load is greater than computation load (line 4) and the accumulation of computation loads of communicating tasks is less than the maximum communication load (line 6), then these two tasks are merged on a single PE after satisfying memory constraints of that PE (line 7). This process is continued till the computation load on any PE becomes the bottleneck and hence further reduction in communication overhead will not yield any performance improvement.

In second phase (line 13 to 19), once the processor becomes the bottleneck, resource optimization is carried out by merging the tasks with minimum computation load such that after merging, the communication overhead or computation load does not overshoot the computation bottleneck determined in the first phase. This step not only enhances resource utilization but also tries to balance the computation load among several PEs by bringing the computation

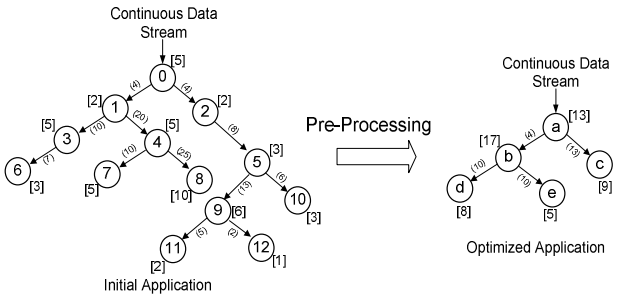


Figure 4: Optimized Application obtained after Pre-processing (Alg 1)

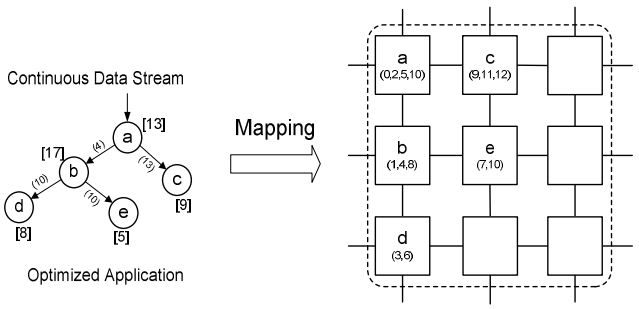


Figure 5: Mapping of Optimized Application by Algorithm 2

load of each PE as close as possible to computation bottleneck.

The worst case complexity (C) of the pre-processing phase has been calculated in terms of number of tasks *(n)* in the application, which is as follows:

$$C = n\text{-choose-}2 + (n-1)\text{-choose-}2 + (n-2)\text{-choose-}2 + \ldots\ldots + 2\text{-choose-}2$$

$$= {}^{n}C_2 + {}^{(n-1)}C_2 + {}^{(n-2)}C_2 + \ldots\ldots + {}^{2}C_2$$

$$= (n^3 - n) / 6 \qquad (1)$$

So, the worst case complexity is $O(n^3)$ which allows us to preprocess the applications with even large number of tasks in a limited time. However, the preprocessing may converge earlier depending upon the type of application.

For example, consider the pre-processing of application graph given in Figure 4, where values in [] and () denotes computation load and communication load in cycles, respectively. Initially, the edge between task 4 and 8 is the main bottleneck and computation load of task 4 & task 8 is less than the communication load, so task 4 and task 8 are merged together. Next, the edge between task 1 and 4 becomes the bottleneck and the sum of computation load of task 1 and task 4 is less than the communication load, so task 1 and task 4 are merged together. This process is repeated till computation load becomes the bottleneck at task 1. Now, resource optimization is carried out by merging task 2 and task 5 as they have minimum sum of computation load. This process is also repeated till an optimized graph is obtained where computation load is the main bottleneck and no further resource optimization can be done without affecting the gain in performance.

The output of this process results in an optimized application graph as shown in Figure 4. This graph contains set of tasks at each node, like, tasks (0, 2, 5, 10) on a, (1, 4, 8) on b, (9, 11, 12) on c, (3, 6) on d and (7, 10) on e.

Algorithm 1: Pre-Processing

Input: *ATG(T,E)*
Output: *Optimized ATG(T,E)*
(1) **do{**
(2) Find task $t_i \in T$ having maximum computation load (max_p_{load}) from *ATG (T,E)*.
(3) Find task $e_i \in E$ having maximum communication load (max_c_{load}) from *ATG (T,E)*.
(4) **if (**$max\_p_{load} < max\_c_{load}$**) then**
(5) Find computation and communication load of connecting tasks t_p & t_q of the edge $e_i \in E$, i.e., $p_{load}(t_p)$ *and* $p_{load}(t_q)$
(6) **if** ($p_{load}(t_p) + p_{load}(t_q) <= max\_c_{load}$) **then**
(7) Merge t_p and t_q to a single node if their memory requirements are satisfied on a single PE and update *ATG(T,E)*.
(8) **else**
(9) break; //goto second phase of optimization.
(10) **end if**
(11) **end if**
(12) **while(**$max\_c_{load} > max\_p_{load}$**)**
(13) Find t_i and t_j with Min ($p_{load}(t_i) + p_{load}(t_j)$) < max_p_{load}) from updated *ATG(T,E)*.
(14) **if** (t_i & t_j are communicating tasks) **then**
(15) Merge t_i and t_j to a single node if their memory requirements are satisfied on a single PE and update *ATG(T,E).*
(16) **else if** ($c_{load}(e_i) + c_{load}(e_j) < max\_p_{load}$) then
(17) Merge t_i and t_j to a single node if their memory requirements are satisfied on a single PE and update *ATG(T,E)*
(18) **else** goto (13) to find next minimum combined load.
(19) **repeat** steps (13) to (18) till condition at (13) is satisfied.

B. Mapping

The optimized application graph thus obtained by the pre-processing is mapped using the Mapping Algorithm (Algo 2). For the optimized application graph shown in Figure 5, first, the initial node (a) is mapped and slave nodes (b & c) are requested. Their mapping is found at the minimum hop distance with respect to (w.r.t.) the master (a). After mapping nodes b and c, their slave nodes (nodes d & e for node b) are requested and their mapping is found. A possible mapping for all the nodes of the optimized application is shown in Figure 5.

Algorithm 2: Mapping

Input: *Optimized ATG(T,E) , AG(P,C)*
Output: *mpg*
(1) Map initial node
(2) Request communicating slave nodes
(3) Map requested nodes at minimum hop distance w.r.t. their master node
(4) Repeat steps 2 to 3 till all tasks are mapped.

V. EXPERIMENTS AND RESULTS

Model-Sim simulator has been adopted for performing experiments in co-simulation (SystemC for applications and RTL-VHDL for NoC [16]). 2D-mesh Topology is

used for arrangement of the NoC. 2D-mesh Topology is used for arrangement of the NoC. Multi-Threading feature of SystemC represents the PEs with two SystemC threads named MPthread and TPthread. The MPthread controls the Global Manager (GM) which manages placement of tasks on different PEs and resource management. The TPthread delineates behavior implementation in terms of execution time and communication overhead for each task as conditioned in a configuration file.

We have evaluated two scenarios (*i*) multiple random, pipeline & tree like streaming applications having 5, 10, 15 and 20 tasks and (*ii*) 20 similar MPEG-4 application, where 5 instances are run concurrently. A 5×5 NoC-based MPSoC is considered, where all the PEs are homogeneous processors. One PE is used for the GM and rest 24 PEs as software resources. All the processors except those supporting the GM are considered to support multiple tasks. Initial task acts as the manager of application and the PE reserved for initial task is pre-defined. The number of times an application runs has been varied.

A. Total Execution Time

The total execution time is the overall duration to execute the application for a defined number of times which includes pre-processing, mapping, configuration, computation and communication time. The communication overhead stands out as the predominant bottleneck, which is greatly reduced by our proposed strategy. Thus, overall execution time gets reduced.

Figure 6 (a) depicts comparison between overall execution time when Smart Nearest Neighbor (SNN) heuristic proposed in [5] and our proposed technique is employed. It is evident from the figure that rise in the complexity of the application corresponds to rise in the improvement of execution time with our technique.

B. Resource Utilization

Resource Utilization is measured as the percentage reduction in the number of PEs used for mapping the application. The new proposed mapping strategy map the maximum number of tasks on the same PE while reducing communication overhead and balancing the processing load among various PEs. The mapping results from two strategies are compared; our approach shows an improvement of 23.33% over the SNN, for applications with 20 Tasks, as shown in Figure 6 (b).

C. Energy Consumption

Energy consumed while transmitting tokens from source PE to destination PE and then processing the token at the destination PE once it is received, amounts to the overall energy consumption of the system. The energy required in transmission and processing is referred as communication and computation energy, respectively. The same energy model as proposed in [6] has been adapted, where 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. For evaluation, full channel bandwidth has been allocated to each link but channel bandwidth may be varied. 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. Total energy consumption is measured as the sum of communication and computation energy from Eq. (2).

$$E_{total} = E_{comm} + E_{comp} \quad (2)$$

Our proposed heuristic reduces communication energy by mapping communication intensive tasks on the same PE, where the different tasks may share common memory, thereby reducing the data exchange among communicating tasks. Thus, in turn total energy consumption is greatly reduced. Figure 6 (c) shows average energy consumption by SNN and our proposed heuristic.

D. Computation Load Variance

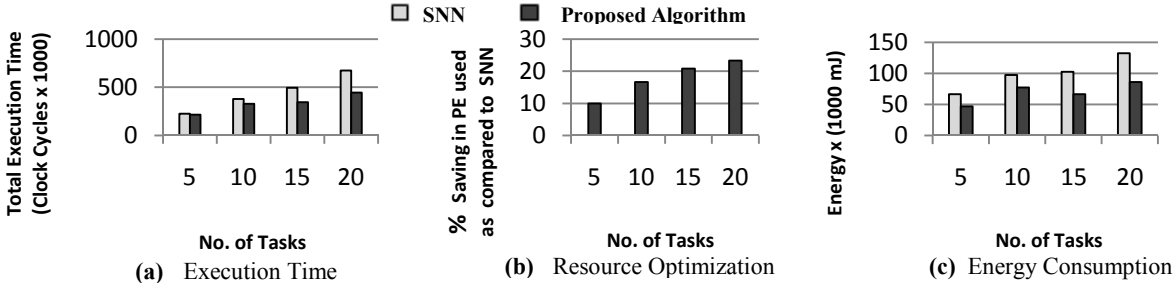
The fair distribution of computation load among the several PEs minimizes the probability of reaching the situation when a single PE remains active most of the time due to high computation load, whereas other PEs remain idle. This scenario leads to poor power efficiency and performance bottlenecks due to overloading of a single PE. However, the proposed heuristic attempts to fairly distribute the computation load among several PEs by combining the less computation intensive tasks onto a single PE till performance bottleneck is detected. Figure 7 shows the average computation load variance for different applications with 10, 15 and 20 tasks. It clearly indicates that the probability of PEs having smaller deviation from the mean computation load has been increased when mapping is carried out using the proposed algorithm.

E. Case Study - MPEG4 Decoder

The proposed heuristic has been applied on real-life application MPEG-4 decoder as mentioned in scenario (*ii*). The application is used in de-compression of encoded video digital data. It is composed of 13 tasks interconnected with each other in a cyclic tree like structure. In Figure 8, the results for execution time, resource Utilization and energy consumption are obtained by applying the proposed algorithm and comparing it to SNN heuristic. The proposed technique reduces total execution time by 33.48%, resource usage by 37.5% and energy consumption by 40.64% when compared to SNN.

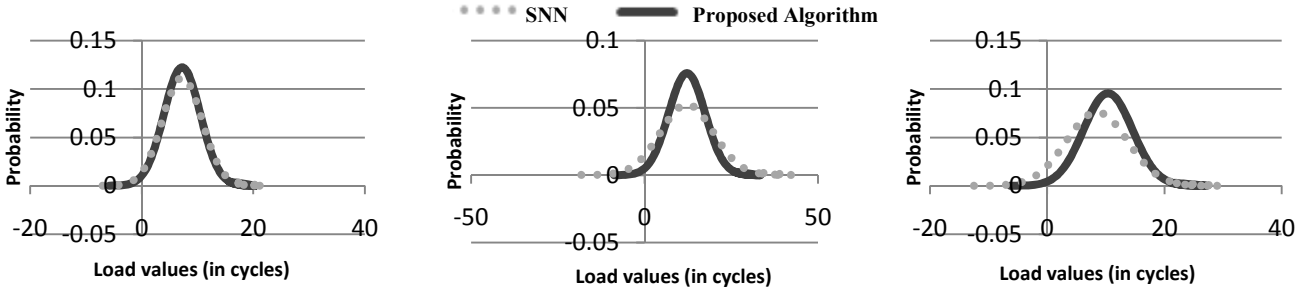
VI. CONCLUSION

This paper describes a new mapping strategy, where placement for a task is found to balance the computation load on different PEs and to reduce communication overhead in the MPSoC platform. Our mapping strategy provides significant improvement in total execution time,



(a) Execution Time **(b)** Resource Optimization **(c)** Energy Consumption

Figure 6: Overall Execution Time, Resource Utilization and Energy Consumption for SNN and Proposed Heuristic for the first evaluated scenario.



(a) Application with 10 Tasks **(b)** Application with 15 Tasks **(c)** Application with 20 Tasks

Figure 7: Computation Load Variance for Applications with 10, 15 and 20 Tasks for the first evaluated scenario.

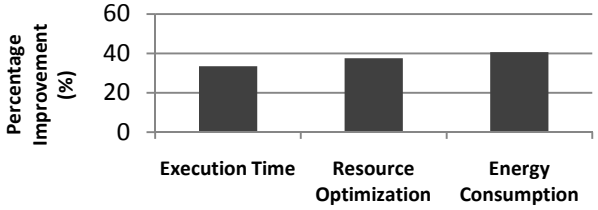


Figure 8: MPEG4 Decoder Application considered for second evaluated scenario.

resource utilization and energy consumption when compared to state-of-the-art run-time mapping heuristic reported in literature. The improvements are clearly enunciated in the experiments and results section. 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. A. Jerraya et al., Guest editors' introduction: multiprocessor systems-on-chips, Computer 38 (7) (2005) 36–40.
2. L. Benini and G. De Micheli, "Networks on chips: A new soc paradigm,"IEEE Computer, vol. 35, no. I, pp. 70–78, Jan. 2002.
3. J. Henkel et al., On-chip networks: a scalable, communication-centric embedded system design paradigm, in: Proceedings of VLSI Design, 2004, p.845.
4. G. Martin, "Overview of the MPSoC design challenge," in DAC '06: Proceedings of the 43rd annual Design Automation Conference. NewYork, NY, USA: ACM, 2006, pp. 274–279.
5. A. K. Singh et al., "Run-time mapping of multiple communicating tasks on MPSoC platforms", International Conference on Computational Science, ICCS 2010. pp. 1013-1020
6. A. K. Singh et al., "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms," Journal of Systems Architecture, vol. 56, no. 7, 2010. pp. 243-256
7. D. Wu et al., Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems, in: Proceedings of the conference DATE '03 in Europe pp. 90–95.
8. S. Murali et al., A methodology for mapping multiple use-cases onto networks on chips, in: Proceedings of the conference on DATE '06 in Europe, pp. 118–123.
9. C. Marcon et al., Time and energy efficient mapping of embedded applications onto noc, in: Proceedings of ASP-DAC, 2005, pp. 33–38.
10. Faruque et al., Adam: run-time agent-based distributed application mapping for on-chip communication, in: Proceedings of the DAC, 2008, pp.760–765
11. L. Smit et al., "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture". FPT'04: Proc. of the IEEE int. Conf. on Field-Programmable Technology, pages 421–424, 2004
12. C.-L. Chou et al., "Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels". CODES+ISSS'07: Proc. of Hardware/software Codesign and system synthesis, pages 161–166, 2007.
13. V. Nollet et al., Run-time management of a mpsoc containing fpga fabric tiles, IEEE Trans. Very Large Scale Integr. Syst. 16 (1) (2008) 24–33.
14. Holzenspies et al., Run-time spatial mapping of streaming applications to a heterogeneous multiprocessor system-on-chip (mpsoc), in: Proceedings of the conference on DATE '08 in Europe, ACM, New York, NY, USA, 2008, pp. 212–217.
15. Braak Ter et al., Run-time Spatial Resource Management for Real-Time Applications on Heterogeneous MPSoCs. In: Conference on Design, Automation and Test in Europe, DATE 2010, 8-12 March 2010, Dresden, Germany
16. F. Moraes et al., Hermes: an infrastructure for low area overhead packet-switching networks on chip, Integr. VLSI J. 38 (1) (2004) 69–93.
17. 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
18. 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.
19. S. Bell et al., Tile64tm processor: a 64-core soc with mesh interconnect in: IEEE International Solid-State Circuits Conference, 2008, pp. 88–598.
20. A. Ngouanga et al., A contextual resources use: a proof of concept through the apaches' platform, in: Proceedings of IEEE Des. and Diag. of Electronic Circuits and Sys., 2006, pp. 42–47
21. Carvalho et al., Congestion-aware task mapping in heterogeneous MPSoCs. System-on-Chip (SoC), 2008.