2009 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools

Mapping Algorithms for NoC-based Heterogeneous MPSoC Platforms

Amit Kumar Singh, Wu Jigang, Alok Prakash, Thambipillai Srikanthan (Senior Member, IEEE)
Centre for High Performance Embedded Systems, Nanyang Technological University, Singapore
{amit0011, asjgwu, alok0001, astsrikan}@ ntu.edu.sg

Abstract— **Mapping of applications onto Multiprocessor System-on-Chip (MPSoC) can be realized either at design-time or run-time. At any time the number of tasks executing in MPSoC platform can exceed the available resources, requiring efficient run-time mapping techniques to meet the real-time constraints of the applications. This paper presents two run-time mapping heuristics for mapping the tasks of an application in close proximity so as to minimize the communication overhead. In particular, the communication overhead between two adjacent hardware tasks is eliminated by mapping them onto the same reconfigurable processing node. We show that the proposed approach is capable of alleviating Network-on-Chip (NoC) congestion bottlenecks to optimize the overall performance. Based on our investigations to map the tasks of applications' at run-time onto an 8×8 NoC-based Heterogeneous MPSoC, our mapping heuristics are capable of reducing total execution time and average channel load of applications when compared to state-of-the-art run-time mapping heuristics.**

Keywords- Multiprocessor System-on-Chip (MPSoC) Design, Network-on-Chip (NoC), Run-time mapping, Mapping Algorithms.

I. INTRODUCTION

The complexity of embedded software applications is now at a point where these applications cannot be supported by a single general purpose processor, inevitably requiring high performance computing platforms. The advancement in Nanotechnology has made it feasible to integrate several embedded processing elements in a single chip to develop Multiprocessors System-on-Chip. MPSoCs are solution to these complex applications in order to meet the performance requirements [1].

The communication infrastructure required to have proper communication amongst multiple PEs can be bus-based, point-to-point or Networks-on-Chip (NoCs)-based [2]. The use of NoC based communication infrastructure is compulsory as NoCs have several advantages over others, such as scalability and shorter wires, which minimizes power consumption. In order to meet the ever-rising performance constraints, NoCs can integrate instruction set processors (ISPs), specialized processing elements like Digital Signal Processors (DSPs), FPGA fabric tiles, dedicated intellectual property cores (IPs) and specialized memories on a single chip towards the development of an MPSoC [3][4].

The homogeneous MPSoCs [5][6][7], consisting of identical processing elements can support some applications, whereas heterogeneous MPSoCs consisting of different types of processing elements can support wider variety of applications. Heterogeneous MPSoCs exploit the distinct features of different processing elements to improve the performance.

Most of the work in literature present static mapping techniques [8][9] that cover only certain scenarios. These techniques find the best placement of tasks at design-time and hence these are not suitable for dynamic workloads. There are a few works which focus on dynamic approaches [10][11]. In dynamic approach tasks are loaded into the system at run-time. Task migration [3][12] can also be used to insert a new task into the system at run-time. In Heterogeneous MPSoCs, task migration is used at run-time to improve the performance. In task migration the tasks are relocated from one processing element to another processing element when a performance bottleneck is detected or when the workload needs to be distributed more homogeneously. Issues related to the task migration such as the cost to interrupt a given task, saving its context, transmitting all of the data to a new processing element and restarting the task in the new processing element are discussed in [3], [12] and [13].

This work describes two new run-time mapping heuristics based on our packing strategy and their performance evaluation for a NoC-based heterogeneous MPSoC. State-of-the-art run-time mapping heuristics do not consider multitasking resources in the platform and also do not perform well when applied to different scenarios. The presented heuristics developed with the packing strategy consider multitasking resources and give better performance compared to state-of-the-art mapping heuristics. The MPSoC platform consists of software and hardware (*Reconfigurable Logic*) processing elements. The software processing elements can support only one task whereas the hardware processing elements considered here are large enough that can support more than one task in parallel. At run-time, the adjacent hardware communicating tasks of an application may get mapped on same reconfigurable processing node, resulting in almost no communication overhead between the tasks. The heuristics try to map the tasks of an application in close proximity within a particular region in order to further reduce the communication overhead between the communicating tasks, thus resulting in a significant performance improvement. The performance metric includes overall execution time and average channel load.

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

II. RELATED WORK

Several chip multiprocessors are being developed by industry [14]. Some domain specific multiprocessors, network processors and general purpose multiprocessors have already been developed by industry for different type of computation requirements.

Static mapping techniques for NoC-based and bus-based MPSoCs are presented in [8][9][15] and [16] to solve the problem of mapping. These techniques find the fix placement of tasks at design time with a well known computation and communication behavior. Therefore, these mapping techniques are not suitable for an adaptive system that changes its configuration over time and requires re-mapping/run-time mapping of applications.

Smit et al. [4] present a run-time task assignment algorithm to map a task-graph on an MPSoC platform. The algorithm maps a task before all other task that needs a scarce resource by taking availability of resources into account. Efficient heterogeneous multi-core architectures for streaming applications and run-time mapping of these applications onto these multi-core architectures are presented in [17].

Nollet et al. [18] describe a run-time task assignment heuristic for efficiently mapping the tasks in a multiprocessor systems-on-chip containing FPGA fabric tiles. With the presence of FPGA fabric tiles, algorithm is capable of managing a configuration hierarchy and it improves the task assignment success rate and quality.

Holzenspies et al. [19] present a run-time spatial mapping technique consisting of four steps to map the streaming applications onto a heterogeneous MPSoC. The algorithm is implemented on an ARM926 running at 100 MHz and it takes less than 4 ms to run the HIPERLAN/2 example.

Faruque et al. [20] present a run-time agent based distributed application mapping technique for NoC-based heterogeneous MPSoCs. The technique presented tries to map the applications in a distributed manner using an agent-based approach. The approach reduces the monitoring traffic and computational effort for the run-time mapping algorithms.

Ngouanga et al. [10] describe a mapping technique based on the attraction forces between communicating tasks that tries to place them near to each other. In [21] and [22] run-time mapping techniques to map the tasks onto MPSoC platforms are presented. The MPSoC platform in [21] is homogeneous while in [22], it is heterogeneous.

Task Migration mechanisms are presented in [3][18]. For migrating a task from one IP to another, the method in [3] uses *task migration points* as a point of reference. Authors in [18] use *checkpoints*, to define when a given task can be migrated.

Carvalho et al. [23] present heuristics for run-time mapping of tasks in NoC-based heterogeneous MPSoCs. Tasks are mapped on the fly, according to the communication requests and the load in the NoC links. The target MPSoC architecture contains software and hardware processing elements. Each processing element can support only one task. Differently from this, in our target MPSoC architecture the hardware processing (*Reconfigurable Logic*) elements can support more than one task in parallel. At run-time if two adjacent hardware communicating tasks of an application get mapped on same processing element, then the communication overhead between the tasks is greatly reduced. Additionally, the mapping heuristics proposed here tries to map the communicating tasks of an application close to each other so as to minimize the communication overhead in order to further improve the performance. Mapping heuristics Nearest Neighbor (NN) and Best Neighbor (BN) presented in [23] are taken for evaluation and performance comparison with our proposed mapping heuristics.

III. TARGET MPSOC ARCHITECTURE

MPSoC architecture used in this work contains a set of different processing elements which interact via a communication network [24]. Software tasks execute in instruction set processors (*ISPs*) and hardware tasks execute in reconfigurable logics (*reconfigurable area-RA*) or in dedicated *IPs*. The reconfigurable areas or blocks considered in this work are large enough that can support more than one hardware tasks in parallel. The communication network uses message passing protocol for inter-task communication similar to that described in [23].

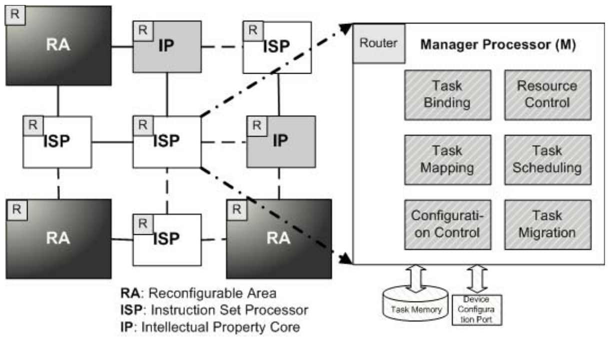


Figure 1. Conceptual MPSoC Architecture

Among the available processing nodes, one of the processing node is used as the Manager Processor (M) that is responsible for task scheduling, task binding, task placement (mapping), task migration, resource control and reconfiguration control. The M knows only the initial tasks of the applications. The initial task of each application is started by the M and new communicating tasks are loaded into the MPSoC platform at run-time from the *task memory* when a communication to them is required and they are not already mapped.

In this work the focus is on *resource control*, *task binding* and *task placement (mapping)*. The resources status is updated at run-time to provide the Manager Processor with an accurate information about the resource occupancy as the mapping decision is taken based on the PEs and NoC use. *For task scheduling* queue strategy is used and there are three queues, one for each type (i.e. hardware, software and initial) of task. If there are no free resources in the system the task enters into their corresponding queue and waits until a resource of same type does not get free.

IV. OUR PACKING STRATEGY

This section introduces our packing strategy for efficient mapping of applications onto a NoC-based heterogeneous MPSoC.

A. *Definitions*

Definitions necessary to explain our packing strategy are as follows:

Definition 1: An *application task graph* is represented as an acyclic directed graph *TG = (T, E),* where *T* is set of all tasks of an application and *E* is the set of all edges in the application. Figure 2 (a) describes an application having initial, software and hardware tasks along with the edges (*E*) connecting these tasks and (b) shows the master-slave pair (communicating tasks). The starting task of an application is the initial task that has no master. *E* contains all the pair of communicating tasks and is represented as $(mt_{id}, st_{id}, (V_{ms}, R_{ms}, V_{sm}, R_{sm}))$, where mt_{id} represents the master task identifier, st_{id} represents the slave task identifier; V_{ms} and R_{ms} are the data volumes and data rate sent respectively from master to slave; V_{sm} and R_{sm} are the data volumes and data rate sent respectively from slave to master respectively. The message rate is described as percentage of available link bandwidth. XY routing algorithm is used to transmit and receive the messages and both rates are relevant in the model as the path taken by messages is different. In XY routing first the packet is transferred in X-direction then in Y-direction for transferring packets from one node to another node.

Definition 2: A NoC-based heterogeneous MPSoC architecture is a directed graph *AG = (P, V),* where *P* is the set of tiles p_i and $v_{i,j}$ є *V* presents the physical channel between two tiles p_i and p_j. A tile p_i є *P* consists of a router, a network interface, a heterogeneous processing element, local memory and a cache.

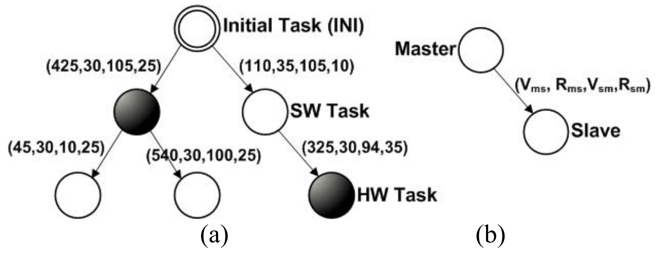


Figure 2. Application Modeling and Master-Slave pair

Definition 3: The *application mapping* is represented by *mpng* : $t_i\,(\in T) \mid\rightarrow p_i\,(\in P)$ to map the tasks of the application onto the NoC-based heterogeneous MPSoC.

B. *The Packing Strategy*

State-of-the-art run-time mapping heuristics to map the applications onto an MPSoC platform consider single task supported processing elements and do not perform well when applied to different scenarios. Here, we have incorporated large enough hardware resources in the platform that can support more than one task in order to map two tasks on one processing element. The mapping heuristics developed with the packing strategy try to map the communicating tasks in close proximity reducing the communication overhead, resulting in improved overall performance compared to the state-of-the-art mapping heuristics.

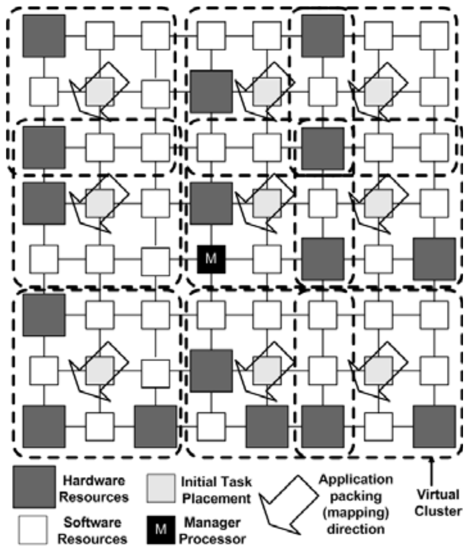


Figure 3. Initial tasks placement for mapping applications with packing strategy

In our packing strategy, all tasks of an application are tried to be mapped close to each other. For each application to be mapped on the MPSoC platform, firstly, clusters are found for each application and initial tasks are placed at the centre of the clusters as shown in figure 3. The cluster boundaries are not fixed for any application and hence a common region can be shared by tasks of different applications. These virtual clusters are used to find the initial task (starting task) placement in a distributed manner so that the new communicating tasks can be mapped close to each other, this greatly reduces the communication overhead between the communicating tasks. After the initial task gets mapped, new communicating tasks of an application are mapped according to the communication request. To map a requested task, firstly, the task is tried to map at the same node making the request as hardware resources can support more than one task. A task in supported by a processing element (PE) if type of PE is same as task type and there is free resource at the PE location. If task is not supported by the node making the request then it is tried to be mapped to the left or down side PE around the node making the request. Now, if neither left nor down side processing element is able to execute the task only then task is tried to be mapped on the top or right side processing element. The same strategy is followed by each application to map their tasks on the MPSoC platform.

With packing strategy the communicating tasks of an application are tried to map close to each other in a compact manner to reduce the communication overhead between the communicating tasks. As with this strategy, first each application tries to map its task towards bottom-left (either on bottom or left PE) side within the cluster hence the processing elements present on top-right edge of the cluster may be used by another application that is also trying to map its tasks towards bottom-left (either on bottom or left PE). In this manner if one application is getting mapped then the applications that are tried to be mapped on top-side, right-side or top-right side may get the free resources on the top and right edges of the first application's cluster and tasks of the other applications can be mapped on these resources. This strategy is applied to all the applications to be mapped, thus most of the applications get the free resources from other application's top-right edge of the cluster. Additionally, as hardware resources can support more than one task in parallel so two communicating hardware tasks can be mapped on one resource location, reducing the communication overhead and making the mapping more compact.

This strategy tries to keep the communicating tasks as close as possible, thus reducing the communication overhead between two communicating tasks. *Execution time* of an application mainly depends on computation time and communication time. With the help of the packing strategy the communication overhead is reduced, resulting in reduced communication time and hence the execution time. *Channel load* also gets reduced as it depends on the communication overhead.

V. RUN-TIME MAPPING ALGORITHMS

Our run-time mapping algorithms are motivated by the packing strategy as discussed in section IV. The given algorithms are light-weight in terms of execution cycles and channel load. First initial tasks are mapped and then new tasks are mapped into the MPSoC at run-time when communication to them is required.

A. Initial Task Mapping

The initial tasks are considered as software tasks so these are mapped onto software processing elements. Initial task mapping has significant impact on the performance of run-time mapping. There are two different ways to map the initial tasks. In the first method, the initial tasks can be mapped on the first free position found in the network that can support the tasks. This may result the initial tasks to be placed very close to each other. Now, when new tasks of different applications are requested to be mapped, the applications have to share the same NoC region, resulting in longer waiting time for a resource to become free for a task to be mapped. This also increases the channel congestion as all the applications are tried to be mapped within a small region. In the second method, to map the initial tasks, clusters are found by partitioning the NoC into regions and the initial tasks are placed at the centre of these clusters. The placement of initial tasks in the clusters is shown in figure 3. The cluster boundaries are virtual so more than one application can share some parts of a given region. This work considers the clustering approach.

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

B. Mapping Algorithm 1

This algorithm is based on the packing strategy discussed in section IV along with the search space (circular search space) of Nearest Neighbor (*NN)* heuristic. If resource at the same node is not able to support the task then a free node able to execute the requested task around the node making the request is searched according to the packing strategy. The search space to map a task goes into circular fashion i.e. first at zero hop distance, if no resource is supported then at hop distance of one and so on. The search space goes on up to the NoC limit (step 12 in Algorithm 1). In our algorithm same search strategy along with the packing strategy is applied as explained in Algorithm 1 on the next page.

To map multiple applications onto the MPSoC platform, algorithm 1 is applied for each application. First, suitable clusters for each application are found and initial tasks are mapped at the centre of the clusters as in figure 3. Next, new requested tasks are mapped dynamically, by applying packing strategy along with the *NN* search strategy as described by algorithm 1.

Our run-time mapping algorithm inside a cluster is light weight in terms of total execution time and average channel load. The algorithm tries to map the communicating tasks in close proximity in a compact manner in order to reduce the communication overhead and so the communication time. Thus total execution time also gets reduced. Average channel load also depends on communication overhead, therefore it is also reduced.

Algorithm 1: Run-time mapping

```
Input: $TG(T,E)$, $AG(P,V)$
Output: mpng (mapping $TG(T,E) \rightarrow AG(P,V)$ )
type ($t_i$): type of task (HW, SW or INI)
type($p_i$): type of tile (HW, SW or INI)
NFR[type]: number of free resource(s) of type type in NoC
(1) Find a suitable cluster for the application (from figure 3)
(2) Map the initial task ($INI \in T$) at the centre of the cluster
(3) for all $t_i \in T$ (except INI, already mapped)
(4)   for all unmapped $t_i$ that is requested
(5)     if (NFR[type($t_i$)] != 0)
(6)       if (at requesting node numberFreeResources > 0 )
(7)         if (type($t_i$)== type($p_i$) at zero hop_distance)
(8)           Select requesting node $p_i \in P$ to map $t_i$
(9)           insert($p_i$ to mpng); update(resources by mpng)
(10)          wait and go back to (4) if new task is requested
(11)      else
(12)        for hop_distance = 1 to NoC limit
(13)          Select left and down side node(s) (near
              requesting node)
(14)          if (node(s) supported)
(15)            Select first free supported node $p_i \in P$ to map $t_i$
(16)            insert($p_i$ to mpng); update(resources by mpng)
(17)            wait and go back to (4) if new task is requested
(18)          else
(19)            Select right and up side node(s) (near
                requesting node)
(20)          if (node(s) supported)
(21)            Select first free supported node $p_i \in P$ to map $t_i$
(22)            insert($p_i$ to mpng); update(resources by mpng)
(23)            wait and go back to (4) if new task is requested
(24)        end for
(25)    else
(26)      insert($t_i$ to Queue(type($t_i$)))
(27)      wait until NFR[type($t_i$)] != 0 (updated at run-time)
(28)      if (NFR[type($t_i$)] != 0)
(29)        release($t_i$ from Queue(type($t_i$))) and go back to (6)
(30)  end for
(31) end for
```

The algorithm first tries to map a requested task at same node (hop distance zero). If same node cannot support the task then only left and down side node(s) at hop distance one is (are) searched and task is mapped onto one of the capable node in order to map the tasks of an application in bottom-left fashion. If neither left nor down side node(s) is(are) able to execute the task, then the task is mapped to the top or right side node(s) whichever is able to execute the task and found as first free. If no node is able to execute the task at hop distance one then the search space goes to hop distance of two and so on. The task is mapped in the same manner at each hop distances and platform resources are updated when a task gets mapped. If none of the PEs in the NoC is able to execute a task, then the task is placed in its corresponding queue and waits until a resource become free that can execute the task. The same process is repeated for each unmapped task that is requested in order to map all the tasks of an application. The same procedure is adopted for all the applications to be mapped.

C. *Mapping Algorithm 2*

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

$$\cos t_z = \sum r_{ch(i,j)} + \sum r_{ch(j,i)} \quad (1)$$

Algorithm 2: Run-time mapping

In algorithm 1, path load computation is incorporated by replacing the lines (15) and (21) 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 1.

As this heuristic includes the path load computation, hence it is a congestion aware mapping heuristic that tries to distribute the channel load in the NoC. Thus, in addition to mapping the tasks in close proximity to avoid the communication overhead between the tasks, this heuristic also tries to distribute the channel load more uniformly, resulting in reduced average channel load.

VI. EXPERIMENTS AND RESULTS

Experiments are performed by *ModelSim co-simulation* (*System-C* for applications and *RTL-VHDL* for the NoC). The results evaluated are *total execution time* and *average channel load* of applications. This section describes the experimental setup followed by the results.

A. *Experimental Setup*

Experiments are performed on the simulation platform similar to that in [23]. This section describes the experimental set up used to perform the experiments.

Each application is modeled as in figure 2, with an initial task, hardware tasks and software tasks. The values present on the edges represent the volume and rates of data to be sent and received by the master as explained in definition 1 of section IV. The task processing time is fixed at 25 microseconds. Rates are fixed from 5 to 20% of the available channel bandwidth, using a Pareto On-Off distribution. Each task transmits from 200 to 500 packets with size varying from 100 to 400 16-bit flits.

The evaluated scenarios are:

- Applications having hardware communicating tasks at leaf as in figure 4(a)
- Applications having hardware communicating tasks in between initial and leaf tasks as in figure 4(b)

In each scenario 20 identical applications, each one with 10 tasks are taken with varying injection rate (% of available channel bandwidth).

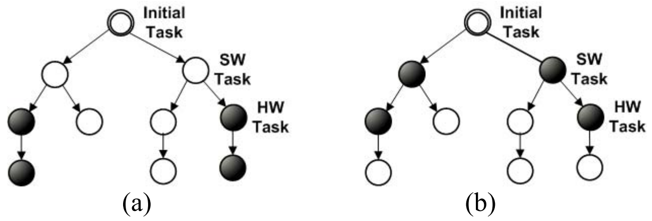


Figure 4. Applications for two Scenarios. (a): Application for scenario 1 and (b): Application for scenario 2

The NoC is modeled in VHDL [24], in an 8×8 2D-mesh topology. NoC is responsible for data transfer between the tasks. As handshake protocol is used to transfer the data therefore each flit is transmitted in two clock cycles, limiting the available channel bandwidth to 50% of its capacity. In the NoC (figure 5) one node is used as Manager Processor (M), 47 nodes as software resources and remaining 16 nodes contain large enough hardware resources such that at each node two tasks can run in parallel in these hardware. Thus there are effectively 32 hardware resources at 16 hardware node locations.

Figure 5 shows the NoC model used. The routers contain set of software and hardware (*Reconfigurable Logic*) processing elements. The data is transferred between the processing elements through the routers using the XY routing algorithm. The M is placed almost in the middle of the NoC so that it can take of all the processing elements equally.

The point should be kept in mind that the M does not know the whole application graphs. It knows only the initial tasks. When initial tasks start their execution, the slave tasks are mapped dynamically, according to the communication request.

The processing elements (PEs) are modeled using *System-C*. Two different *System-C* threads are used to model the PEs, one for the M and another for rest of the PEs as *MPthread* and *TASKthread* respectively. The *MPthread* is responsible for the MPSoC resource management, task mapping, task scheduling and task configuration. Also, this thread contains channels and PEs matrix to manage system use and scheduling queues. The *TASKthread* is responsible for the task behavior implementation that is described by a

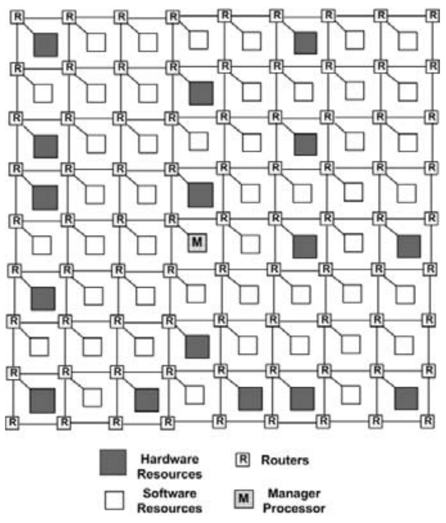


Figure 5. Our 8 x 8 NoC Model containing types of nodes used.

configuration file. This file contains execution time and communication rates and these values can be customized.

In the current work, software resources can execute only one task but the hardware resources are large enough to execute two tasks in parallel. Multi-tasking software resources along with multi-tasking hardware resources will be considered as future work.

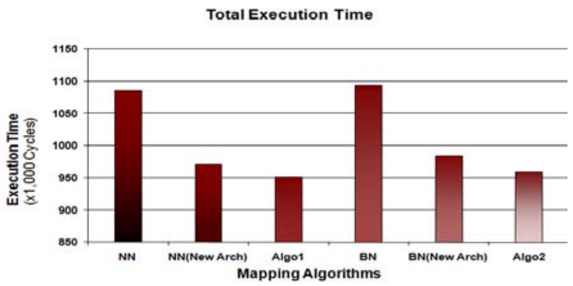
B. Experimental Results

Results are obtained for the state-of-the-art run-time mapping heuristics from the previous platform where each node was able to support only one task and from our architecture described in experimental setup. The performance improvements are described. Results obtained from our proposed heuristics show further performance improvement when compared with state-of-the-art run-time mapping heuristics. Results are simulated for rate of 5-20% of the available channel bandwidth. Without loss of generality, here we have shown results for 5% only.

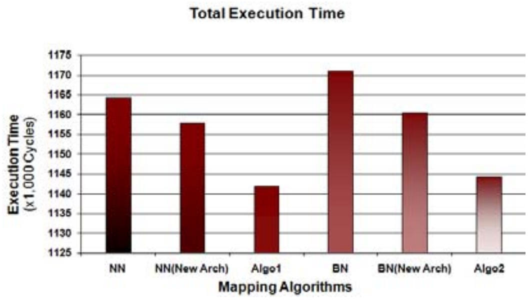
1) Total Execution Time

Total execution time for each task comprises of mapping time (the time to find the placement), configuration time, communication time, computation time and waiting time when no resource is free in the platform. Out of all these portions contributing to total execution time, the communication time dominates. The allocation time consists of mapping time and configuration time and is indirectly considered.

Here, as hardware nodes support more than one tasks in parallel so at run-time, two hardware communicating tasks are mapped on one node, reducing the communication overhead between the tasks that results in reduced communication time. Additionally, when the packing strategy is also applied, the tasks are mapped in close proximity, resulting in further reduction of communication

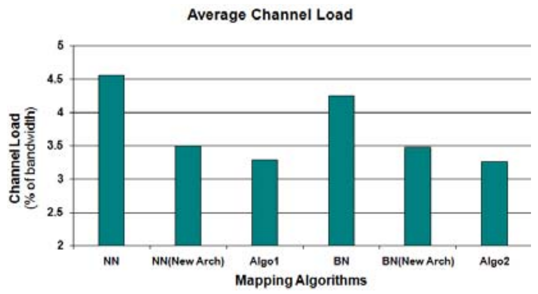


(a) Scenario 1

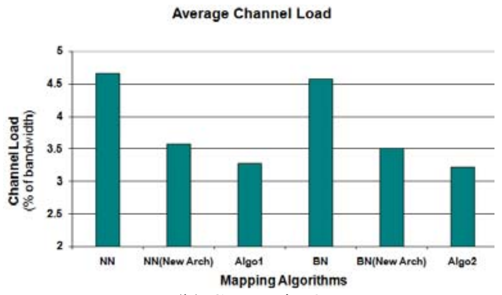


(b) Scenario 2

Figure 6. Graphs showing Total Execution time for the two simulated scenarios.



(a) Scenario 1



(b) Scenario 2

Figure 7. Graphs showing Average Channel Load for the two simulated scenarios.

overhead and so the communication time. Thus with our approach total execution time is greatly reduced. It has also been seen that the mapping time contributing to total execution time gets reduced because we have reduced the search space (explained in section V) to find the placement of a task.

Graphs in figure 6 show the total execution time taken by 20 identical applications for the two scenarios discussed in experimental setup sub section. The results show the total execution time taken by *NN* and *BN* heuristics on the older platform (each node support only one task) and on the new platform (hardware resource support more than one task) along with the execution time taken by Algorithm 1 (ALG1) and Algorithm 2 (ALG2) on the new platform. The percentage gain is shown with respect to the execution time on older platform.

The results show that in scenario 1, our mapping strategy reduces the total execution time by 12.40% for ALG1 when compared with *NN* on older architecture and by 12.27% for ALG2 compared to *BN* on older architecture. In scenario 2, it is reduced by 3.20% for ALG1 with respect to *NN* on older architecture and 2.30% for ALG2 when compared with *BN* on older architecture. *NN* and *BN* show significant reduction in total execution time when compared with the results from older architecture for both the evaluated scenarios. Our extra experimental results confirm that new algorithms get similar improvement over *NN* and *BN* for other rates too.

2) Average Channel Load

The average channel load represents the NoC use. In our mapping algorithms, ALG1 does not consider traffic during mapping, but explores the proximity of communicating tasks. In second algorithm (ALG2) we consider the traffic as well during mapping, thus trying to distribute the channel load more uniformly.

In the new platform, the communicating tasks get mapped on same node, reducing the communication overhead between the tasks. As the channel load directly depends on communication overhead, thus it gets reduced. The packing strategy further reduces the communication overhead.

Graph in figure 7 show the average channel load for the two scenarios. Average channel load is represented as % of bandwidth. The results are evaluated for NN, BN, ALG1 and ALG2 on new platform. For comparison, the results are evaluated for NN and BN on the older platform too.

In scenario 1, ALG1 and ALG2 reduce the average channel load by 27.91% and 23.53% respectively and in scenario 2 by 26.82% and 24.60% respectively when compared with the average channel load from *NN* and *BN* heuristics on the older platform. *NN* and *BN* also reduce the average channel load significantly when compared with the results obtained from the older platform for both the evaluated scenarios. We have got similar improvements for other rates too.

VII. CONCLUSION

This paper details our packing strategy and two run-time mapping heuristics based on it, to map the applications efficiently onto an 8 × 8 NoC-based heterogeneous MPSoC. First heuristic tries to map the tasks of an application in close proximity, reducing the communication overhead (communication time) between the communicating tasks. The second heuristic considers traffic in addition to the proximity of tasks while mapping, resulting in more uniformly distributed channel load. The hardware resources (*Reconfigurable Logic*) present in the platform support two tasks in parallel, resulting in further reduction of communication overhead. Our mapping heuristics reduce the average channel load by a large amount for both the evaluated scenario. The total execution time for the first evaluated scenario is reduced significantly with a small improvement in the second evaluated scenario. State-of-the-art run-time mapping heuristics also gets performance improvement when evaluated on the new platform. The improvements are clearly enunciated in the experiments and results section.

The MPSoC platform considered in this work contains software processing elements that can support only one task and hardware processing elements (*Reconfigurable Logic*) that are able to support more than one task in parallel. In future, we plan to extend all the platform resources as multitasking processors along with the hardware resources and evaluation of different benchmarks on the platform.

ACKNOWLEDGMENTS

We thank Mr. Ewerson Carvalho (first author of the paper [23]) for providing us the basic simulation environment and helping us to extend it for our idea.

REFERENCES

[1] Jerraya, A.; Tenhunen, H.; Wolf, W. Guest Editors' Introduction: Multiprocessor Systems-on-Chips. IEEE Computer, v.38 (7), 2005.

[2] Benini, L. and Micheli, G. Networks on Chips: A new SoC paradigm. IEEE Computer, v.35(1), 2002.

[3] Nollet, V.; Marescaux, T.; Avasare, P.; Mignolet, J-Y. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. DATE, 2005.

[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. FPL, 2004.

[5] 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. ISSCC, 2007.

[6] Tilera Corporation. TILE64^{TM} Processor. Product Brief, 2007.

[7] Lin, L. ; Wang, C. ; Huang, P. ; Chou, C. ; Jou, J. Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. ASPDAC, 2005.

[8] 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.

[9] Marcon, C.; Borin, A.; Susin, A.; Carro, L.; Wagner, F. Time and Energy Efficient Mapping of Embedded Applications onto NoCs. ASP-DAC, 2005

[10] Ngouangal, A.; Sassatelli, G.; Torres, L.; Gil, T.; Soares, A.; Susin, A. A contextual resources use: a proof of concept through the APACHES platform. DDECS, 2006.

[11] Wronski, F.; Brião, F.; Wagner, R. Evaluating Energy-aware Task Allocation Strategies for MPSoCs. DIPES, 2006.

[12] Bertozzi, S.; Acquaviva, A.; Bertozzi, D.; Poggiali, A. Supporting task migration in multiprocessor systems-on-chip; a feasibility study, DATE, 2006.

[13] Kalte, H.; Lee, G.; Porrmann, M. Context Saving and Restoring for Multitasking in Reconfigurable Systems. FPL, 2005.

[14] http://view.eecs.berkeley.edu/wiki/Chip_Multi_Processor_Watch, Oct, 2008.

[15] Murali, S.; Coenen, M.; Radulescu, A.; Goossens, K.; De Micheli, G. A methodology for mapping multiple use-cases onto networks-on-chip. DATE, 2006

[16] Ruggiero, M.; Guerri, A.; Bertozzi, D.; Poletti, F.; Milano, M. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-onchip. DATE, 2006.

[17] Smit, G.; Kokkeler, A.; Wolkotte, P.; Burgwal, M. Multi-core Architectures and Streaming Applications. SLIP, 2008.

[18] 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 (VLSI) Systems, Vol. 16, No. 1, 2008.

[19] 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.

[20] Faruque, M. A. A.; Krist, R.; Henkel, J. ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication. DAC, 2008.

[21] Chou, C.-L. and Marculescu, R. Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels. Haraware/software Codesign and system synthesis (CODES+ISSS'07), 2007

[22] Lei, T. and Kumar, S. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. Digital Systems Design (DSD), 2003.

[23] Carvalho, E.; Moraes, F. Congestion-aware task mapping in heterogeneous MPSoCs. System-on-Chip (SoC), 2008

[24] 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