A Joint Scheduling and Mapping Method for Dynamically Reconfigurable SoCs Interconnected by an on-chip Mesh Network

Ling Wang[a], Xiangnan Sui[a], Yingtao Jiang[b]

[a] School of Computer science and technology, Harbin Institute of Technology, Harbin China
[b] Department of Computer and Electrical Engineering, University of Nevada, Las Vegas, USA

Abstract—Reconfigurable SoC interconnected by an NoC architecture mandates a low power and small footprint reconfiguration scheme that enables multiple applications to effectively share precious hardware resources. The trade-offs between the static compilation and the dynamic reconfiguration has yet to be evaluated at the highest possible abstraction level of the NoC designs. In this paper, reconfiguration is thus considered during the application scheduling and mapping stage. That is, for a given set of applications which target to run on a dynamically reconfigurable NoC architecture, a schedule and map of these applications needs to be found to minimize the communication cost, while satisfying the timing, area and other applicable design constraints. The proposed solution follows a three-step design flow. In the first task scheduling step, multiple applications are scheduled to a minimal number of processor nodes while meeting the timing constraints. Next, applications that shall be mapped onto the same hardware resource but run at the different time instances through hardware reconfiguration are merged. In this step, effort is dedicated to minimize the reconfiguration cost. In the last step, all the applications are finally mapped onto the targeted NoC architecture. The experiment results have shown that the proposed method has achieved 50% area reduction than a conventional scheme that does not consider reconfiguration cost.

Keywords—NoC, Mapping, scheduling, reconfiguretion

I. Introduction

SoC (Systems on Chip) designs have shown rapid growth in complexity with an increasing number of integrated processors, memory, accelerators, and various other types of IP cores [5][20]. Networks on chip (NoC), due to their structural advantages and modularity [2], have emerged as the design paradigm for connecting the many on-chip cores in SoCs. The choice of network topology as well as mapping and routing strategies adopted has direct implications on the network merits, such as the average inter-IP distance, the total wire length, and the communication load distributions, which in turn, determine the power consumption and the average latency of the network. The topologies proposed for on-chip networks vary significantly from regular, tiled-based architectures [5][6] to fully customized ones [7][10][11]. Since a fully customized NoC is designed and optimized for a specific application, it gives the best performance and power results just for that application. On the other hand, reconfigurable NoCs, where network topology, routing protocols and even some of the IP cores can be altered, can deliver the optimal performance and power result across all the target applications. Through reconfiguration, hardware resources can be optimized for each application that often has different functionalities and communication characteristics from the other applications. Yet in this case, the trade-offs between the static compilation and the dynamic reconfiguration have to be carefully evaluated at the highest possible abstraction level of the NoC designs, particularly at the application scheduling and mapping level.

There have been several studies that attempt to deal with mapping of the multiple applications onto reconfigurable NoC architectures [8, 9]. In [8], a worst-case-based mapping method was proposed, where the cores and the NoC are mapped to satisfy the most serious constraints imposed by all the target applications. In [9], a method that maps multiple applications based on the traffic characteristics of a single application was proposed. After application mapping, a reconfigurable NoC is created by embedding programmable switches between any two routers of a mesh-based NoC, but these programmable switches unfortunately have very high area penalty which can impose a serious problem to NoC designs. Even more noticeable, neither of the two schemes [8, 9] has considered reconfiguration cost during the mapping and route determination process.

One big drawback of these approaches is that

Corresponding Author: Ling Wang, Email Address: wangling@hit.edu.cn

they all target to find a suitable mapping solution without considering the cost associated with dynamic reconfiguration. In [18], a mapping flow is proposed for dynamic reconfigurable platform where reconfiguration cost is indeed minimized along with communication cost. However, scheduling, which preferably shall be considered along with the mapping, is actually absent from the proposed flow.

To address the above problems, a scheduling and mapping scheme is proposed, and this new scheme attempts to balance both the reconfiguration cost and the communication link cost. That is, in the core mapping phase, rather than mapping tasks directly onto the physical NoC, tasks are first mapped to virtual cores, which are then mapped onto the physical NoC. The proposed scheme thus is divided three major steps which include scheduling, core mapping, and eventually NoC architecture generation.

In this paper, to overcome the aforementioned problems, an integrated scheduling and mapping scheme is proposed. In specific, when scheduling is performed, each task of the target applications will be first scheduled to virtual cores, after which these virtual cores will be mapped to the physical cores of a NoC. Note that reconfiguration cost is considered along this scheduling and mapping process, and thus the communication cost will be eventually minimized.

The rest of the paper is organized as follows. In Section 2, a general dynamically reconfigurable NoC architecture is briefly introduced. The design flow and a motivation example are presented in Section 3. The three phases of the proposed design scheme are detailed in Section 4. Experiment results are reported in Section 5, and conclusions are finally drawn in Section 6.

II. Dynamically Reconfigurable NoCs

In a reconfigurable, mesh-based interconnection network that is of a concern to this paper, a router is connected directly to its adjacent routers and all the routers are connected to a reconfiguration controller, as shown in Fig. 1 [16]. Here the Reconfiguration Controller (RC) is tasked to control the reconfiguration process, and its basic components include the Repository, the Configuration Port (CP) and the Reconfigurable Interface (RI). The Repository contains a memory unit that stores the reconfigurable modules' configuration data, bearing a great similarity to the configuration files used for configuring an FPGA chip. The RI is necessary to implement a static routing between a reconfigurable module and the rest of the system. All the reconfiguration-related activities, including changes of network topology, routing protocols and/or IP cores, are controlled by this reconfiguration controller. In general, dynamic hardware reconfiguration can only be implemented on dynamically reconfigurable devices. Thus, FPGAs are used in this study.

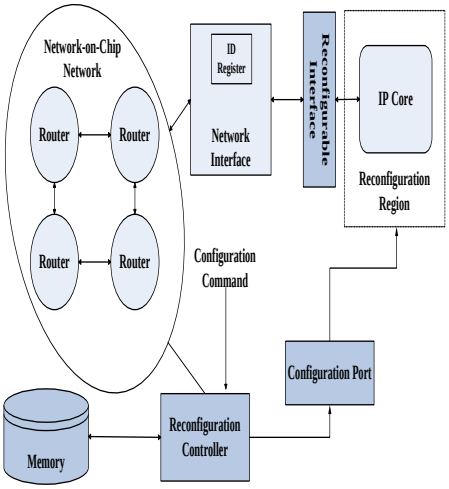


Fig. 1. The Reconfigurable NoC model

III. Problem Formulations

Definition 1: A task graph TG (V, E) is an acyclic directed graph that represents an application, with each vertex $v_i \in V$, representing a task, and an edge between two vertices v_i and v_j, denoted as $e(v_i, v_j) \in E$, representing a communication link between v_i and v_j. The weight of edge $e(v_i, v_j)$ represents the communication bandwidth required between v_i and v_j.

Definition 2**:** A core graph VP (U, L) is an undirected graph, with each vertex $u_i \in U$ representing a virtual core node, and the edge between two vertices u_i and u_j, denoted as $l(u_i, u_j) \in L$, representing a communication link between u_i and u_j. The weight of edge $l(u_i, u_j)$ represents the communication bandwidth between u_i and u_j.

Definition 3: An NoC topology architecture graph AG (Q, R) is an undirected graph, with each vertex $q_i \in Q$ representing a core node in the NoC topology and the edge between two vertices q_i and q_j, denoted as $r(q_i, q_j) \in R$, representing a communication link between q_i and q_j. The weight of edge $r(q_i, q_j)$ represents the communication bandwidth between q_i and q_j.

The problem that attempts to schedule and map multiple applications onto an Network-on-Chip architecture is defined as follows:

Given a set of applications, applications 1, 2, …*i*,…, *A*, represented by a set of task graphs, TG = { TG_i (T_i, R_i), *i*=1， …, | A |}, and a number of IP cores, find a mesh architecture AG that connects all the IPs, and schedule and map these applications onto these IP cores with minimum communication cost,

$$totalcom = \sum_{j=1}^{|E|} B(e_j) dist(source(e_j, dest(e_j))$$

while satisfying the area and timing constraints. Here
(i) area is determined as the number of mapped nodes on the AG;
(ii) timing is defined as the scheduling time of the TG;
(iii) e_j is the flow in the input task graphs TG and $|E|$ is the total number of flows of TG;
(iv) B(e_j) is the bandwidth of the flow e_j in the topology architecture described as AG;
(v) Source(e_j) and dest(e_j) represent the source and the destination nodes of flow e_j in the TG, respectively;
(vi) dist(source(e_j), dest(e_j)) represents the hop count between source(e_j) and dest(e_j) with pre-determined routes in AG, assuming XY routing is adopted.

The above problem is a special case of processor scheduling that is known as NP-complete. Thus, a heuristic algorithm is developed to solve the above problem. In the literature, various topologies [2], including Mesh, torus, GNLS [16] and etc., have been proposed for NoC designs. In this paper, although we concentrate on the mesh topology, the proposed method can be readily applied to other topologies.

IV. Design Flow of Scheduling and Mapping Applications onto Reconfigurable NoCs

An example is given to illustrate how system performance is impacted when reconfiguration is considered during the scheduling and mapping stage. Two parallel applications a1 and a2 are shown in Fig.2.a and 2.b, respectively, and their respective core graphs generated after scheduling are shown in Fig.2.c and 2.d.

If reconfiguration cost is not considered during scheduling/mapping, these two core graphs (Fig. 2.c and 2.d) are mapped to the topology one at a time. As a result, the total communication cost is 140 and the area cost is 5 (Fig. 2.g). But if reconfiguration cost is considered after scheduling, one can see that as V2 and V3 run at different time slots, they actually can be reconfigured and mapped to the same core as shown in Fig. 2.e. As such, the area cost now drops to 4, and its total communication cost is 132, as shown in Fig. 2.h. Further reduction of the reconfiguration cost is possible. Instead of having V2 and V3 run on the same core through reconfiguration, if V2 and V4 are actually mapped to the same core, as shown in Fig. 2.f, the area cost after scheduling and mapping remains 4, but the total communication cost decreases to just 127 (Fig. 2.i).

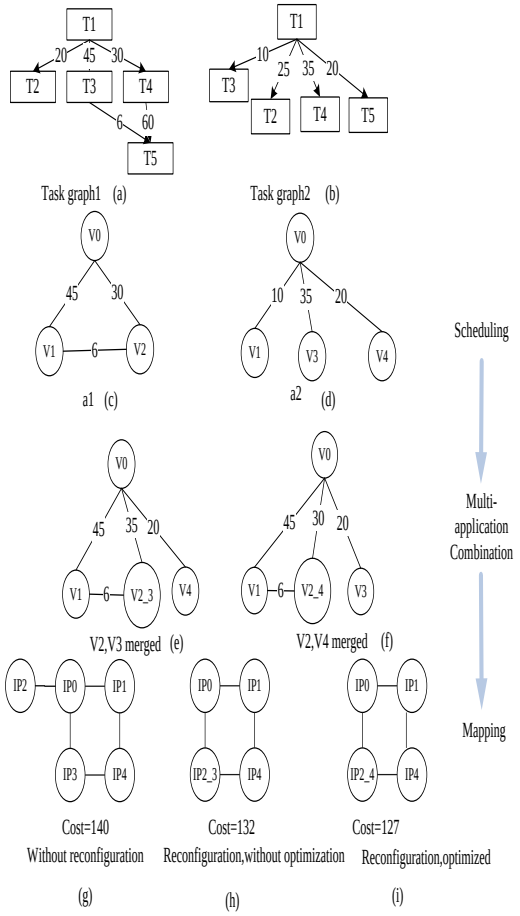


Fig. 2 An example showing Scheduling and Mapping with and without task merging through reconfiguration

The example illustrated in Fig. 2 has clearly indicated that communication cost can be effectively reduced without incurring additional hardware cost, if optimization on reconfiguration is performed along with scheduling and mapping.

This study thus attempts to develop a scheme that can effectively schedule and maps applications onto multiple IP cores, meanwhile taking into account of the reconfiguration cost. Due to its NP-completeness in nature, we attempt to solve the problem following a design flow that includes three phases:

(1) During the scheduling phase (Phase 1), applications are scheduled onto the minimal number of virtual cores (as opposed to the physical IP cores) while meeting all the governing timing constraints.

(2) During the application merging phase (Phase 2), applications that can share the same virtual core but run at different time slots are identified, and these applications are thus considered to be merged to minimize the reconfiguration cost.

(3) During the final mapping phase (Phase 3), virtual cores are mapped to the physical cores in the way that the communication cost can be minimized. Note that the final NoC architecture is generated at this step.

4.1. Phase 1: Task scheduling

In this phase, the proposed task scheduling algorithm takes the number of virtual cores and the application task graphs as its input. Note that, instead of directly mapping tasks to physical cores connected by a NoC architecture, we actually schedule them onto virtual cores. The number of virtual cores can be equal to or greater than the number of physical cores. In our current implementation, the number of the virtual cores is set to be the same as that of the physical cores. The main idea of this scheduling step is to put the tasks into the same group if their run times do not overlap, and then assign one virtual core to each task group.

4.1.1. Scheduling Algorithm

This algorithm tries to find a schedule for each application represented by a task graph. The input to the algorithm is the task graph of an application, and the output is a core graph represented by virtual cores and their connections. For simplicity, run time of each task on a virtual core is set to be exactly one cycle. The objective of this scheduling algorithm is to minimize the number of virtual cores in the output graph under the timing constraint. There are three major steps in the algorithm.

Step 1, tasks are sorted by an ascending order of their start times.

Step 2, schedule the task in the task list that has the earliest start time (i.e., the first entry of the task list) onto an available virtual core. Once a task is scheduled, it will be deleted from the task list. Repeat this process until all the tasks have been scheduled.

Step 3, after all the tasks in the input task graphs have been scheduled onto the virtual cores, the connections among virtual cores have to be determined, following a policy given below:

--If two tasks in the task graph are scheduled onto the same virtual core, there is no need to add an edge between them.

--If two task in the task graph are determined to be scheduled onto two different cores, we need to (i) either find two virtual cores between which an edge shall be added, or (ii) specify the communication bandwidth between the two tasks, when an edge already exists between the two virtual cores.

Detailed scheduling algorithm is given below.

TaskScheduling {

Input:
T: Task Graph
P: list of available Virtual Core (VP) cores
Nv: the number of VP cores

Output:
The Core Graph of the application // see Def. 2 in Section II

Variable Declarations:
Here Q[v] records the time when virtual core, node v, becomes available;
D[v] records the set of tasks running at each IP core;
T_s^R records the start time of task R;
T_r^R records the run time needed to complete task R;
$T_s^R + T_r^R$ gives the scheduled time to complete task R.

Procedure body:

// Step 1
(1) Initialize array Q[v] of each VP core by setting array Q[v] to null, here v=1, 2,…Nv.
(2) Initialize the task set of each VP core by setting array D[v] to null, here v=1, 2,…Nv.;
(3) Sort the tasks in ascending order of start time and save the sorted tasks as task list T';

// Step 2
(4) If T' is not empty, assign the first element in T' to R, and then delete this element from T';
(5) Select node v from P such that Q[v] is minimized;
(6) Schedule R to v by assigning the start time and run time to R;
(7) Update Q[v] by having Q[v] <= $T_s^R + T_r^R$
(8) Add the first task in the list T' to D[v]
(9) Repeat lines between 4 and 8 until all the tasks in T' have been mapped onto VP cores.

// Step 3
(10) Add edges among Q[v];
(11) Return the generated Core Graph;

}

The complexity of this scheduling algorithm is bounded by the number of virtual cores. As each application is scheduled onto a virtual core through a two-loop iteration, the complexity of this algorithm is $O(|Nv|^2)$, where Nv is the number of virtual cores to be scheduled.

4.1.2. An Illustrative Example

We take the benchmark from TGFF [17] as an example to illustrate the proposed task scheduling algorithm. As can be seen from Fig. 3, there are 13 task nodes, of which maximum 5 tasks can be allowed to start at the same time. In this regard, at least 5 VP cores are needed to run all the tasks. The scheduling result after applying the proposed task scheduling algorithm is thus given as Fig. 4.

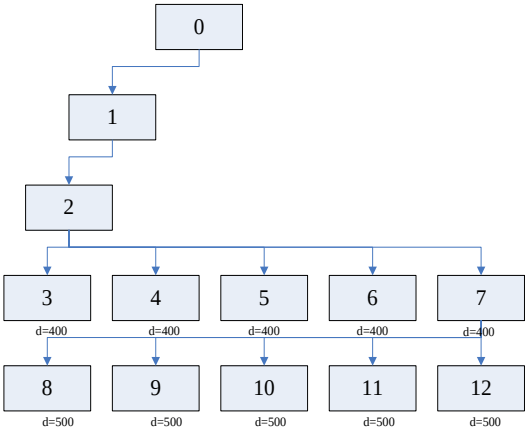


Fig. 3 A benchmark from TGFF[17]

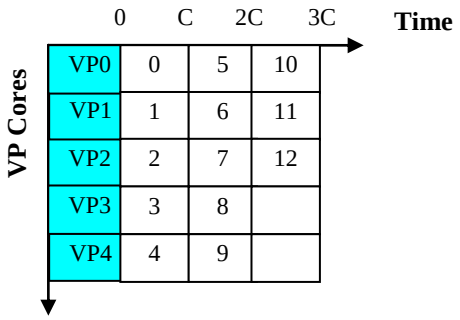


Fig. 4 Scheduling Result for the benchmark shown in Fig. 3

From Fig. 4, one can see that tasks 0, 5 and 10 have been scheduled to execute on virtual core 0 at the 0^{th}, the 1^{st}, and the 2^{nd} clock cycles, respectively. The other tasks have also been scheduled to execute on other virtual processors. The output core graph after task scheduling is shown in Fig. 5, where the weight of each edge corresponds to the communication cost between the two connecting VCs. For instance, communication cost between VP0 and VP1 is 491.

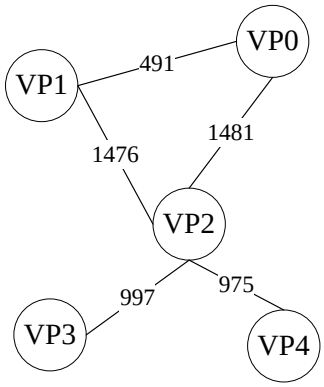


Fig. 5 Core Graph generated after task scheduling as shown in Fig.. 4

4.2. Phase 2: Merge of Multiple Applications and Minimization of Reconfiguration Cost

In Phase 2, the proposed algorithm takes the core graphs (each consists of virtual cores and weighted edges) obtained from phase 1 as its input, while the output is one merged core graph. The objective here is to minimize the number of virtual cores in the output core graph while satisfying the timing constraints.

4.2.1. Algorithm Description

This algorithm attempts to find two graphs, from the output core graphs obtained in Phase 1, that can be merged into one graph. This core graph merging process is repeated until no more graphs can be merged.

When an edge is identified to be merged with an existing edge, bandwidth between the two connecting virtual cores has to be updated to take the higher weight of the two edges. Then, two nodes can be merged assuming the longest communication bandwidth is maintained.

GraphMerging {

Input:

Two applications, Application1 and Application 2, represented by their respective core graphs cg1, and cg2

MT is set as time constraint condition

Output:

A core graph

Variable Declarations:

BW is the total communication bandwidth of a data flow graph.

OptimalCombine(cg1, cg2, MT){
1) Merge common Virtual Core (VP) cores of cg2 to cg1;
2) Sort VP cores in cg2 according to ascending order of their total communication bandwidths, and put them into a list, *L*;
3) If *L* is empty, go to Step 9;
4) Get the first element (VP core) of *L*, and is recorded as *n*;
5) Find core *m* from cg1 so that merging *m* with *n* will lead to the smallest BW;
6) If MT >0 then merge VP core *n* with *m* and then MT = MT-1;
7) Else Add *n* into dfg1 as a new VP core;
8) Delete *n* from *L* and go back to step3;
9) Update bandwidths of all the edges;
}

The complexity of the algorithm is determined by finding the virtual cores from the two input core graphs that shall be merged. For each virtual core of one graph (cg1), all the cores of the other graph (cg2) have to be traversed once (Step 5). The timing complexity is thus O ($N1 \times N2$), where *N*1and *N*2 are cg1 and cg2's numbers of the virtual cores, respectively.

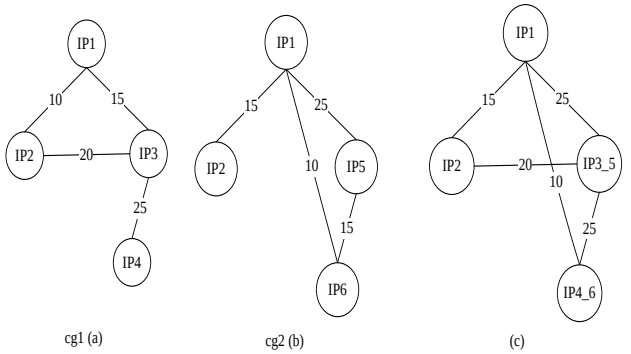


Fig.6 An Example showing merging of two core graphs (cg1 and cg2) to form a new core graph shown in (c) while minimizing communication bandwidth

4.2.2. An Illustrative Example

Suppose there are two core graphs, cg1 and cg2, derived from two applications, shown in Figs. 6.a and 6.b, respectively, and the timing constraint is set as 2. During the merging process, virtual cores IP1 and IP 2 in cg2 are first merged with the same virtual cores in cg1. Next, IP 5 and IP 6 in cg2 are put into the list L. After searching through cg1, IP3 in cg1 is merged with IP5 in cg2 since merging of these two will result in the lowest BW while still satisfying the timing constraints. Next, IP4 in cg1 and IP6 in cg2 are merged. Eventually, the merged core graph, with a communication bandwidth of 95, is shown in Fig. 6.c.

4.3. Phase 3: NoC mapping and optimization

The input of this phase is one core graph consisting of virtual cores and edges. The output of this phase is an mapped NoC architecture. The objective of this mapping step is to minimize the communcaiton cost under the timing and area constraints. There are three major steps in this algorithm. In the first step, an initial mapping is obtained, followed by a step where minimum path routing computation is performed. In the last step, the initial solution is iteratively improved by pair-wise swapping of nodes, and finally the NoC architecture is generated. That is, this algorithm attempts to map all the virtual cores to the physical cores of the NoC. As two or more virtual cores may be mapped to one core on NoC, communication cost of the NoC has to be considered in this phase.

4.3.1. Initialization

At the beginning, a mesh topology is first created with the minimum number of cores that shall be able to accommodate all the cores in the core graph. That is, for a core graph with N cores, a $\lceil N \rceil$ by $\lceil N \rceil$ mesh needs to be created. For instance, if there are 8 cores in the merged core graph, a mesh with 9 cores is created. The virtual core with the largest number of neighbors in the core graph is first mapped to an available core in the mesh with the largest number of neighbors. Next, of the core that have not to be mapped, the one which has the highest communication cost with the already mapped cores is selected for mapping. This procedure is repeated until all the cores in the core graph have been mapped onto the mesh architecture.

4.3.2. Routing Determination

The shortest routing is performed after the initial mapping. The total communication cost can be calculated using the shortest paths between any pairs of the source and the destination nodes. A quadrant graph is created between the source and the destination, as the shortest path between the source and the destination sits within the quadrant that they belong to. Then, Dijkstra's shortest path algorithm is

applied to the quadrant graph and the minimum path is obtained.

4.3.3. Mapping Optimization and NoC Architecture Generation

In this phase, a heuristic simulate-annealing-based algorithm is used to generate the final mapped NoC topology with the minimum total communication cost. In essence, a pair of nodes are selected from the initial mesh (Section 4.3.1) and they get swapped, after which routing is performed to calculate the corresponding communication cost. If the communication cost of this newly mapped NoC topology is lower, it will be considered as a better solution and optimization will continue from this updated topology. Above procedure is repeated until no further optimization is possible, and the mapped NoC topology is finally generated. The complexity of this routine is relative to size of the mesh. As the number of nodes increases, there will be greater opportunities for node swapping.

V. Experimental Results

To evaluate the performance of the proposed algorithms as described in Section 4, a couple of experiments using an in-house developed platform have been performed. In specific, a 2D mesh-based NoC is designed with 16-bit wide channels. Routers in this NoC support wormhole packet switching and deterministic XY routing. Benchmarks include TGFF suites [17] and real-world applications.

In this first set of experiments, we apply the proposed algorithm to TGFF Benchmarks. Table 1 reports the scheduling results of 4 TGFF benchmarks after phase 1 (Section 4.1) is completed. In the second and third columns of the same table, the numbers of the nodes and the edges of the input task graphs are given,
respectively. For comparison, the fourth and the fifth columns, respectively, report the numbers of nodes and edges in the output core graph.

Table 1. Scheduling Results (Phase 1 described in Section 4.1)

Bench mark	No. Nodes At TG	No. of Edges at Task graph	No. of Nodes at Core Graph	No.of Edges at Core graph	Schedul-ing Time (Cycles)
tgff1	23	31	6	11	35
tgff2	18	23	5	9	30
tgff3	12	14	4	5	25
tgff4	15	22	4	6	30

The scheduling time is given in the sixth column, where each node is assumed to run at 5 clock cycles. It can be seen that after scheduling, the number of cores in the output core graph is much less than that of the tasks in the input task graph. Some tasks are scheduled to the same core to reduce the number of the nodes in the output core graph. The number of communication edges in the output core graph is also much lower than that of the input task graph for the same reason.

Table 2. Task Graph Merging Results

Bench mark	Nodes	Edges	Scheduling Time (Cycles) Before	Scheduling Time (Cycles) After
tgff1* tgff3	6	11	80	60
tgff2* tgff4	5	9	75	60
tgff2* tgff3 *tgff4	5	9	100	85
tgff1*tgff2* tgff3*tgff4	6	13	135	125

In the second column of Table 2, various combinations of TGFF benchmarks are shown. For instance, benchmark tgff1*tgff3 indicates that the core graphs of tgff1 and tgff3 are merged. In tgff1*tgff3, there are 6 nodes (the third column) and 11 edges (the forth column). When tgff1 and tgff3 are executed in series, a total of 80 cycles is needed (the fifth column) as opposed to 60 cycles needed to schedule merged graph tgff1*tgff3 (the sixth column). In this case, the scheduling time of the merged graph is reduced by 15%. Here reconfiguration time between cores is set to be 2 cycles.

As certain cores can be reused through reconfiguration, the number of cores (nodes) in the merged graph is reduced dramatically, as shown in Fig. 7, which can be translated into significant area saving. Take the tgff1*tgff3 as an example: the number of cores is reduced.

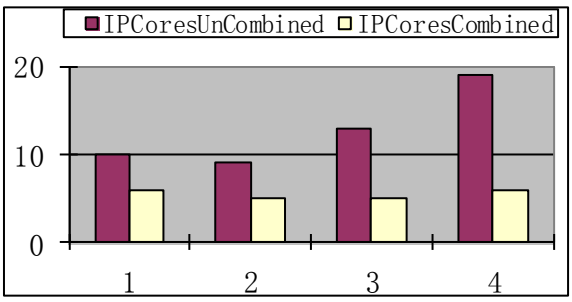


Fig. 7 the number of cores before the merging of the core graphs vs. the number of cores after the merging

from 10 (before merging) down to 6 (after merging). Finally, a merged graph is mapped to the actual mesh NoC, and the final mapping results are

tabulated in Table 3. In our experiment, we define a hotspot on an NoC is a core that experiences 30% more traffic than an average core. In the third column, the number of hotspots on the NoC for each application after final mapping is given. For instance, both tgff1*tgff2 and tgff2*tgff4 have 2 hotspot while for the other two applications, each has 3 hotspots. It is shown that our algorithm applying on NoC with more hotspots can achieve more reductions on communication cost.

Table 3 Mapping Results of Various Applications onto NoCs

Benchmark	No. Cores	Hot Spot	Initial Comm. Cost without reconfiguration	Optimal Comm. Cost with reconfiguration
tgff1* tgff3	6	2	1785	1664
tgff2* tgff4	5	2	1998	1733
tgff2*tgff3 *tgff4	5	3	2014	1751
tgff1*tgff2 * tgff3*tgff4	6	3	2788	2200

In Table 3, the initial communication cost without reconfiguration is given in the fourth column, while the optimal communication cost with reconfiguration is reported in the fifth column.

For instance, an NoC with 6 cores (the second column) is generated for benchmark tgff1*tgff3, where 2 hotspots exist (the third column). The communication cost without reconfiguration is 1785 (the fourth column), while the communication cost is reduced to 1664 with reconfiguration (the fifth column). The most significant reduction on communication cost is achieved for benchmark tgff1*tgff2*tgff3*tgff4 (Table 3), where communication cost is reduced from 2788 cycles to 2200 cycles, a merely 21.2% reduction

B Evaluation of the proposed scheme using real applications running on a set-top box SoC

We have also evaluated the performance of the proposed algorithm by applying it to real-world applications. In particular, five applications run in a SoC for set-top box are adopted, as given in Fig.8. We present the experimental results for the applications running on 4 different configurations: (i) applications A1 and A2 running on P1, (ii) applications A2, A3 and A4 on P2, (iii) applications A1 ,A2 , A4 and A5 on P3, and (iv) applications A1, A2, A3, A4 and A5 on P4. After the merging algorithm is completed for each design (Phase 2 in Section 4.2), the results are given in the Fig. 9.

The numbers of cores before and after the merging process for all 4 designs are reported in Fig.10.a. For P1, 14 cores is actually needed if no core is merged (the second vertical bar in Fig. 10.a), and that number can be reduced to 7 (the first vertical bar in Fig. 10.a) if some cores can be shared. Across all four designs, on average, the number of cores needed is reduced by more than 50% with cores can be shared among applications through reconfiguration. Actually, one can see that the more cores can be shared, the more significant of area saving a design can achieve, as the case in P4.

The numbers of communication links before and after the merging process for all 4 designs are shown in Fig. 10.b.

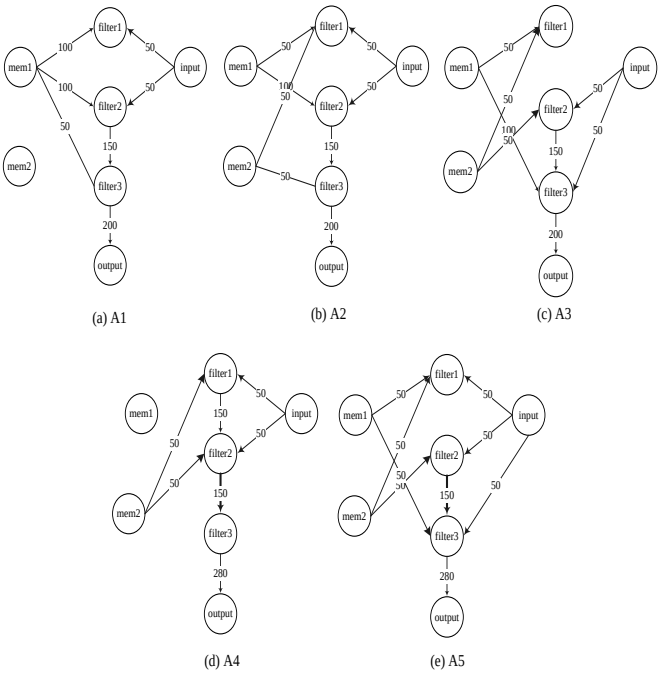


Fig. 8. Five applications running on a top-box SoC

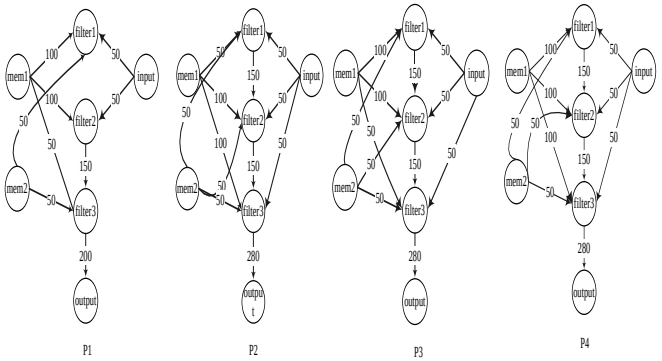
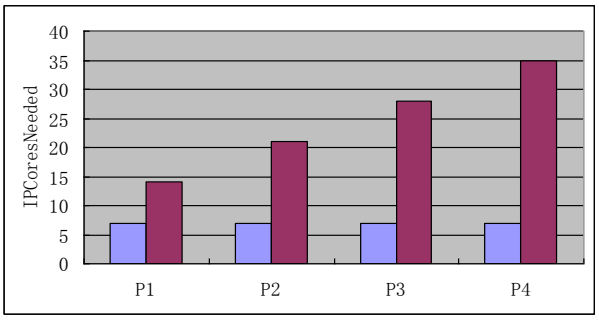


Fig. 9. The core graphs after merging for four designs.

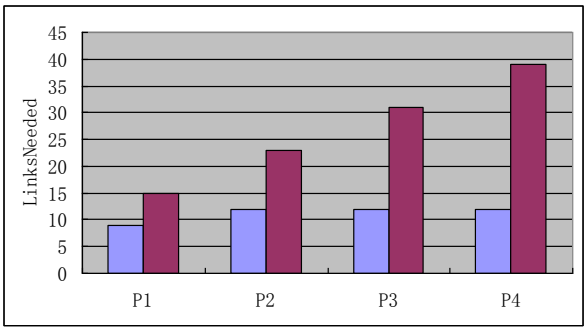
For P1, 15 links is actually needed if no core is merged (the second vertical bar in Fig. 10.b), and that number can be reduced to 9 (the first vertical bar in Fig. 10.b) if some cores are shared. Across all four

designs, on average, the number of communication links needed is reduced by more than 50% when core sharing among applications is possible through reconfiguration. Actually, one can see that the more cores can be shared, the less communication links needed to connect all the communicating cores, as the case in P4.

Finally, the generated mesh architectures for each of the four designs are shown in Fig. 11.



(a)



(b)

Fig10 Experimental results for four designs (a) Core reduction due to core sharing through reconfiguration (b) Communication link reduction due to core sharing through reconfiguration

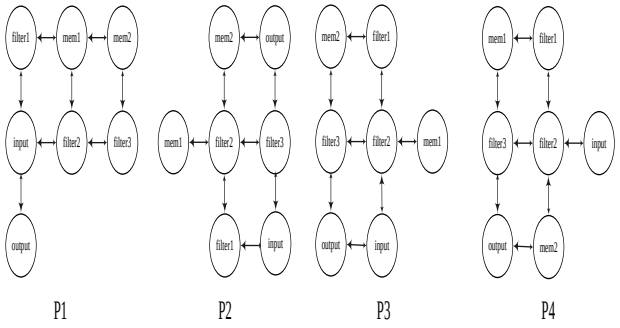


Fig.11. Optimal mesh topology generated

VI. Conclusions

In this paper, a scheduling and mapping multiple applications on dynamically reconfigurable NoC was proposed to help minimize the communication cost, while satisfying the timing, area and other applicable design constraints. To save area, applications that can be mapped onto the same hardware resource but run at the different time instances through reconfiguration are identified, and the cost incurred in reconfiguration is actually considered along with application scheduling and mapping. The experiment results have shown that the proposed method has achieved 50% area reduction than a conventional scheme that does not consider reconfiguration cost.

Acknowledgement

The authors acknowledge the support from NSF of China (60706004 and 60876018).

References

[1] L. Möller, R. Soares, E. Carvalho, I. Grehs, N. Calazans, and F. Moraes, "Infrastructure for Dynamic Reconfigurable Systems: Choices and Trade-offs," *Proc. 19th annual symposium on Integrated circuits and systems design*. 2006:41-49

[2] L. Benini and G. De Micheli. *Networks on Chip: A New SoC Paradigm*. IEEE Computers. 2002:70-78

[3] S. Murali and G. De Micheli, "Bandwidth-constrained Mapping of Cores onto NoC Architectures," Proc. Design, Automation and Test in Europe Conference and Exhibition. 2004, 2: 896-901

[4] J. Hu and R. Marculescu, "Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints," *Design, Automation and Test in Europe Conference and Exhibition*. 2004:234-239

[5] D.Bertozzi, A. Jalabert, S. Murali, R.Tamhankar, S. Stergiou, L. Benini and G. De Micheli, "NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-chip," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 2, pp. 113-159, Feb. 2005.

[6] K. Goossens, J. Dielissen, O.P. Gangwal, S. G. Pestana, A. Radulescu and E. Rijpkema, "A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification," *Proc. Design, Automation and Test in Europe Conference and Exhibition*. 2005:1182-1187

[7] T. Lei and S. Kumar, "A two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture," *Proc. Euromicro Symposium on Digital System Design*. 2003: 180-187

[8] K. Goossens, A. Radulescu and A. Hansson, "A Unified Approach to Constrained Mapping and Routing on Network-on-chip Architectures," *Proc. International Conference on Hardware-Software Codesign and System Synthesis*. 2005: 75-80

[9] J.Hu and R. Marculescu, "Exploiting the Routing Flexibility for Energy/ Performance Aware Mapping of Regular NoC Architectures," *Design, Automation and Test in Europe Conference and Exhibition*. 2003,1:1-6

[10] S. Murali, M. Coenen, A. Radulescu, K. Goossens and G. De Micheli, "A Methodology for Mapping Multiple Use-Cases onto Networks on Chips," *Proc. Design, Automation and Test in Europe Conference and Exhibition*. 2006, (1): 1-6.

[11] S. Murali, M. Coenen, A. Radulescu, K. Goossens and G. De Micheli, "Mapping and Configuration Methods for Multi-use-case Networks on Chips," *Proc. 11th Asia South Pacific Design Automation Conference*. 2006:1-6.
[12] A. Hansson, M. Coenen and K. Goossens, "Undisrupted Quality-of-Service during Reconfiguration of Multiple Applications in Networks on Chip," *Proc. Design, Automation and Test in Europe Conference and Exhibition*. 2007: 954-959.
[13] T. T. Ye, L. Benini and G.De Micheli, "Analysis of Power Consumption on Switch Fabrics in NetworkRouters," *Design Automation Conference*. 2002: 524-529
[14] R. Dick, D. Rhodes, and W. Wolf. "TGFF: Task Graphs For Free," *Hardware Software Codesign Conference*, pp. 97–101, 1998.
[15]I. Beretta, V. Rana, D. Atienza, and D. Sciuto, "A mapping flow for dynamically reconfigurable Multi-Core System-on-chip Design," *IEEE Trans. Computer Aided Design*, vol. 30, no.8, pp.1211-1224 , Aug. 2011.