A Method to Remove Deadlocks in Networks-on-Chips with Wormhole Flow Control

Ciprian Seiculescu*, Srinivasan Murali§ *, Luca Benini‡, Giovanni De Micheli*
* LSI, EPFL, Lausanne, Switzerland,{ciprian.seiculescu, giovanni.demicheli}@epfl.ch
§ iNoCs, Lausanne, Switzerland, murali@inocs.com
‡ DEIS, University of Bologna, Bologna, Italy, luca.benini@unibo.it

ABSTRACT

Networks-on-Chip (NoCs) are a promising interconnect paradigm to address the communication bottleneck of *Systems-on-Chip (SoCs)*. Wormhole flow control is widely used as the transmission protocol in NoCs, as it offers high throughput and low latency. To match the application characteristics, customized irregular topologies and routing functions are used. With wormhole flow control and custom irregular NoC topologies, deadlocks can occur during system operation. Ensuring a deadlock free operation of custom NoCs is a major challenge. In this paper, we address this important issue and present a method to remove deadlocks in application-specific NoCs. Our method can be applied to any NoC topology and routing function, and the potential deadlocks are removed by adding minimal number of virtual or physical channels. Experiments on a variety of realistic benchmarks show that our method results in a large reduction in the number of resources needed (88% on average) and NoC power consumption, area reduction (66% area savings on average) when compared to the state-of-the-art deadlock removal methods.

Keywords

Network-on-Chip (NoC), deadlock, topology, application specific

1. INTRODUCTION

Network-on-Chip (NoC) paradigm has evolved as a promising solution to the interconnect problem of *Systems on Chips (SoCs)* [1]. To achieve high throughput and low latency, most NoCs use wormhole flow control. With wormhole flow control, it is possible for a set of packets to have a cyclic dependency of resources, where each of them waits on a port held by another and none of them can advance. Such a scenario is called as a routing level *deadlock* [10]. Deadlocks in a network can block communication between cores and can even lead to a complete network failure. Therefore it is essential to have methods to handle them.

In many SoCs, the cores are heterogeneous in nature and the communication patterns are well defined. The NoC topologies and routing functions are custom designed to match the application specifications, leading to more power-performance efficient designs. For regular network topologies, deadlocks can be avoided by restricting the routing function so that certain turns in the network are prohibited. In [17], [18], methods to prohibit turns on irregular custom topologies to avoid deadlocks are presented. However, the methods have to be used during the construction of the NoC topology, otherwise connectivity between cores cannot be guaranteed. While [18] can be applied on arbitrary topologies it does require bidirectional links, which is not always the case with application specific topologies. While new links could be opened to make connections bidirectional, this is not always possible when the technology imposes constraints on the number of links, as described in [21].

Most of the existing manual and automated topology synthesis methods [6] use additional virtual channels (VC) or physical links to remove deadlocks in the design. *Resource ordering* is a simple method to prevent deadlocks in custom topologies [10]. In this method, the communication channels (physical links or VCs) are assigned to different classes that are ordered. Achieving deadlock free operation of custom topologies with minimal area-power overhead however, is a major challenge. In this work, we present a new method to remove deadlocks in custom NoC topologies. Our method adds virtual channels (VCs) minimally to remove deadlocks (please not that is also possible to add physical channels if the NoC architecture does not support VCs), thereby incurring a very low area-power overhead.

We perform experiments on several SoC benchmarks and show that our method results in a large reduction in the number of additional links needed to remove deadlocks (an average of 88%) when compared to the resource ordering method. This translates to a large reduction in NoC area (an average of 66%) and power consumption (an average of 8.6%). Our experiments also show that the method is practical, as the total area, power overhead to remove deadlocks is less than 5% when compared to a design with no mechanism to remove deadlocks.

2. RELATED WORK

An introduction to the NoC paradigm and the issues related to the architectural synthesis of NoCs are presented in [1]. Methods to map applications to cores of regular topologies are presented in [2]-[3]. Methods to synthesize application specific NoC topologies are presented in [4]-[9]. Our method is applicable to arbitrary topologies, so both regular and irregular application specific topologies can be given as inputs.

In [12], the authors give the necessary and sufficient condition for the adaptive routing algorithm to be deadlock free and analysis of routing algorithms for regular topologies are presented in [13]-[16]. However, most of these methods are not applicable for custom irregular topologies. A necessary and sufficient condition for deadlock freedom with static routing is given in [10]. A survey of the deadlock free routing algorithms is presented in [11]. Our method also uses the necessary and sufficient condition from [10] to detect the deadlocks.

In [19], a methodology to design adaptive deadlock free routing functions for application specific traffic patterns is presented. In [9], [5], the authors use the turn-prohibition method to remove deadlocks. However, all these methods are integrated with the topology synthesis process and cannot be applied to arbitrary NoC topologies and routing functions.

978-3-9810801-6-2/DATE10 © 2010 EDAA

3. PROBLEM FORMULATION

Algorithm 1 Deadlock removal

1: {Initialize CDG using Topology and Routes}
2: Build $CDG(C, D)$ from $TG(S, L)$ and R_k $\forall k \in [1 \ldots |\text{G(V,E)}|]$
3: C = $GetSmallestCycle()$
4: **while** $C \neq \emptyset$ **do**
5: ‹f_cost, f_pos›=$FindDepToBreakForward(C$, flows)
6: ‹b_cost, b_pos›=$FindDepToBreakBackward(C$, flows)
7: **if** $f\_cost \leq b\_cost$ **then**
8: $BreakCycleForward(C, f\_cost, f\_pos)$
9: **else**
10: $BreakCycleBackward(C, b\_cost, b\_pos)$
11: **end if**
12: Update $TG(S, L)$ and R_k $\forall k \in [1 \ldots |\text{G(V,E)}|]$ from the current $CDG(C, D)$
13: C = $GetSmallestCycle()$
14: **end while**

An input topology is represented as a graph, defined as follows:

DEFINITION 1. *A Topology Graph $TG(S, L)$ is a directed graph where the vertexes $s_i \in S$, $i = 1 \ldots N$ represent the switches and N is the number of switches in the topology and the edged $l(s_i, s_j) \in L$ represents a physical link between switch s_i and switch s_j.*

We also need as input the description of the communication flows. The communication graph is defined as follows:

DEFINITION 2. *The communication graph is a directed graph, $G(V, E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j) representing the communication flow between the cores v_i and v_j.*

For each communication flow, a route has to be defined which describes which of the links in the topology are used by a particular flow to reach from source to destination. We also take the description of the routes as input. A route is defined as follows:

DEFINITION 3. *A Route is a set of channels (a physical link and the corresponding VC) R = {$l_{1,0}, \ldots l_{n,k}$} where the channel $l_{i,j}$ uses the physical link $l_i \in L$ and the VC j. The route defines the channels that a flow will use to reach from source to destination. The order of the channels in the set is also the order in which the channels will be traversed by a packet in the network.*

From the *topology graph* $TG(S, L)$, the *communication graph* $G(V, E)$ and the set of all routes R_k corresponding to flows in $G(V, E)$ the algorithm will build the *Channel Dependency Graph (CDG)* which is used to find the possible deadlock conditions, and then to remove them. It is defined as follows:

DEFINITION 4. *The Channel Dependency Graph is a directed graph $CDG(C, D)$, with each vertex $c_i \in C$ represents a channel in the topology (a physical link and the corresponding VC) and an edge d(c_i, c_j) represents a dependency between the two channels. A dependency is given when there is at least one route which used channel c_i and then immediately channel c_j.*

In Figure 1 an example of a topology is shown. There are four switches in the topology {SW1, SW2, SW3, SW4} connected by four channels {L1, L2, L3, L4} to form a ring. We we consider four communication flows F1, F2, F3 and F4 that have the following routes R1 = {L1, L2, L3}, R2 = {L3, L4} , R3 = {L4, L1} and R4 = {L1, L2}. For the topology in Figure 1 and the four flows we obtain the CDG from Figure 2. In [10] the authors have shown that a necessary condition for a deadlock to occur in a network with wormhole routing is to have a cycle in the CDG. We define a cycle as follows:

DEFINITION 5. *A cycle in the $CDG(C, D)$ is defined as a set ϕ_k = { $c_1 \ldots c_j$} with $c_i \in C$ and there is an edge in $d(c_i, c_{i+1})$ $\forall i = 1 \ldots j$ plus an edge $d(c_j, c_1)$ in the CDG. We denote the set of all cycles in the CDG by Φ.*

Cycles in the CDG can be broken by adding new vertices and removing one or more of the edges between the vertices involved in the cycle. The corresponding operations in the network for adding vertices to the CDG is to add new VCs to create new channels between switches and to remove edges in the CDG corresponds to modifying the routes of some of the flows. The network with the topology from Figure 1 can have deadlocks because the corresponding CDG from Figure 2 has a cycle. To prevent deadlocks, we have to break the cycle in the CDG. We can break the cycle in the CDG by adding a new vertex L1' in the CDG and modifying the route of the flow F3 to use L1' instead of L1. The resulting acyclic CDG is presented in Figure 3 and the corresponding modified topology is shown in Figure 4.

The problem is formulated in following way: given a topology graph $TG(S, L)$, the communication graph $G(V, E)$, the set of all routes R_k and the corresponding $CDG(C, D)$ with $\Phi \neq \emptyset$, how to get $TG'(S, L')$ and R'_k such that the corresponding $CDG'(C', D')$ will have $\Phi' = \emptyset$ and $|L'| - |L|$ is minimal.

We present an algorithm that removes all deadlock conditions, by removing all cycles in the CDG generated for the network. A cycle is removed by adding one or more VCs in the topology (or vertices in the corresponding CDG) and re-routing some of the flows on the newly created channels. The cycles are removed one by one starting with the smallest one. The algorithm terminates when all cycles are removed. The goal of the algorithm is to remove all deadlock conditions by adding the minimum number of extra VCs.

4. DEADLOCK REMOVAL APPROACH

Algorithm 2 $FindDepToBreakForward(C$, flows)

1: {Find flows taking part in cycle C}
2: F = ∅
3: **for all** i ∈ flows **do**
4: **if** | path(i) ∩C| > 1 **then**
5: F = F ∪ {i}
6: **end if**
7: **end for**
8: {Calculate costs for each flow in cycle C}
9: **for** $k = 1 \ldots |F|$ **do**
10: cost(i)(k) = 0, $\forall i \in C$
11: val = 1;
12: **for** $i = 1 \ldots |path(F_k)|$ **do**
13: current_vertex = $path(F_k)(i)$
14: **if** current_vertex ∈ C **then**
15: cost(current_vertex)(k) = val;
16: val = val + 1
17: **end if**
18: **end for**
19: **end for**
20: cost(i) = max(cost(i)(k)) $\forall i \in C$ and $\forall k \in 1 \ldots |F|$
21: f_cost = min(cost(i)) $\forall i \in C$
22: f_pos = argmin(cost(i)) $\forall i \in C$
23: Return ‹f_cost, f_pos›

In Algorithm 1 the major steps of our method are presented. Given the topology graph, the communication graph and the description of the routes as inputs, in Step 2 of the algorithm the CDG is build for the current configuration of the network. In Step 3, we run an initial search to find the smallest cycle in the CDG. We use

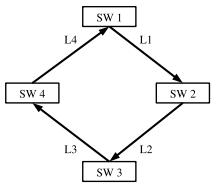

Figure 1: Topology example

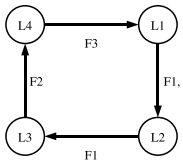

Figure 2: CDG example

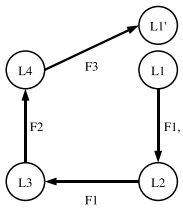

Figure 3: Modified CDG

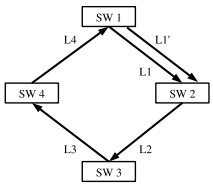

Figure 4: Modified Topology

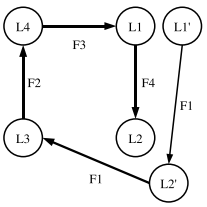

Figure 5: Break in forward direction

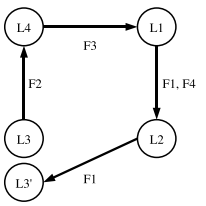

Figure 6: Break in backward direction

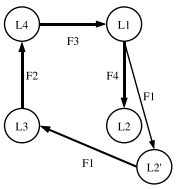

Figure 7: CDG with new vertex and cycle

Table 1: Cost table in forward direction

	D_1	D_2	D_3	D_4
F1	1	2	0	0
F2	0	0	1	0
F3	0	0	0	1
F4	1	0	0	0
MAX	1	2	1	1

the heuristic of breaking the smallest cycle first, as it can also lead to breaking a larger cycle sharing some of the edges with this one. If no cycles are found in the initial search, the algorithm ends as the initial topology is deadlock free and no modifications are needed.

To find the cycles in the CDG we run *breadth first search* starting search from every vertex of the graph. If the starting vertex is encountered during the search then a cycle is detected in the CDG and the starting vertex is part of a cycle. By running the search from every node, we find all cycles and the procedure $GetSmallestCycle$ returns the one that has the smallest length. In the simple case, in order to break a cycle, we need to remove an edge between any two vertices in the cycle by duplicating the vertex before or after the edge. Suppose, we duplicate the vertex after the edge, we connect the edge to the new vertex instead of the original one and the cycle can be broken.

However, in a general case, to break a cycle more than one vertex may need to be duplicated. For example, in Figure 7, we show how adding an extra vertex still maintains the cyclic dependency. The number of vertices that need to be duplicated to remove an edge depends on the configuration of the flows relative to the vertices in the cycle. There are two ways in which the vertices can be replicated, relative to the position of the edge that is removed from the cycle. We say that an edge is removed in *forward direction*, if vertices are duplicated from where a flow enters the cycle up to the edge. We say that an edge is removed in *backward direction* if the vertices are duplicated from the edge to where the flow causing the edge exits the cycle. It can be seen from Figures 5 and 6 that the direction can have an impact on the number of vertices that are duplicated. The algorithm checks the cost of breaking the cycle in both directions in steps 5 and 6. The procedures returns the edge that has the minimum cost for each of the directions.

In Step 7, the cost of breaking in the forward direction is compared with the cost of breaking in the backward direction and depending on which cost is smaller, one of the two procedures is called: $BreakCycleForward$ or $BreakCycleBackward$. Once the CDG is changed, in the next step the topology graph and the routes are updated accordingly. After the current cycle is broken in Step 13, the algorithm searches for the next smallest cycle in the updated CDG. The algorithm terminates when the CDG becomes acyclic.

4.1 Finding the Edge to Remove

Depending on which edge we want to break the cycle, number of vertices that need to duplicated in the CDG (and hence the number of new VCs to be added in the topology graph) can differ. Therefore it is important to find the edge that can be removed with the least number of replication of vertices. The steps to find the best edge to break are described in Algorithm 2.

The algorithm starts by finding all the flows that create dependencies at each edge that is involved in the cycle. This is necessary because in order to remove one edge, the flows that are creating that dependency have to be routed on new channels that will be added. In steps 3 to 7 of the procedure, all flows in the design are checked to see if they are part of a dependency on one or more edges in the cycle. In order to break a dependency created by a flow, all the channels used by the flow in the cycle prior to the dependency have to be duplicated. For example for the CDG in Figure 2 we show some examples on how to remove edges in Figures 5 and 3.

In the Steps from 9 to 19, we calculate the cost of breaking the cycle considering the effects of each of the flows individually. For each of the flows, we calculate the cost in the following way: we find where the flow enters the cycle, this gives the first vertex of the cycle that is used by the current flow. To break the dependency between this first vertex and the next vertex in the cycle, the cost is one as we only have to duplicate this first vertex. If we want to break the dependency at the next edge on the cycle, two vertices need to be duplicated and the cost for this is two. We continue in a similar manner until the flow leaves the cycle. In the end, we build a table that has as many rows as there are flows involved in the cycle and has as many columns as there are edges in the cycle. An example of a cost table for breaking the cycle in the forward direction for the CDG in Figure 2 is given in Table 1.

The cost table gives the effect of each flow independently, so now we have to take into account the combined effect of all the flows in the cycle. For that, we calculate the combined effect at each dependency in Step 20. We take the maximum cost between all flows because it tells how many vertices have to be replicated. The minimum cost and the position in the cycle of the dependency to break are returned by the function in Algorithm 2. In a similar manner the function that calculates the costs in the backward direction can be obtained. When calculating the costs for the backward direction, we analyze the paths in reverse order from destination to source.

Given a cycle, the position of the dependency in the cycle that has to be removed and the cost of removing the dependency, $BreakCycleForward$ or $BreakCycleBackward$ functions brake the cycle by duplicating vertices. The cost already indicates the number of vertices that need to be duplicated. For breaking the cycle in the forward direction, the vertices are duplicated from the edge that is to be removed (output from the Algorithm 2) till the vertex where the flow with the largest cost enters the cycle. The path of the flows creating the edge that is removed are modified to use the new vertices.

5. EXPERIMENTAL RESULTS

In our experiments we generated application specific topologies for different switch counts on several realistic SoC benchmarks using an existing tool [9]. Please note that our deadlock removal

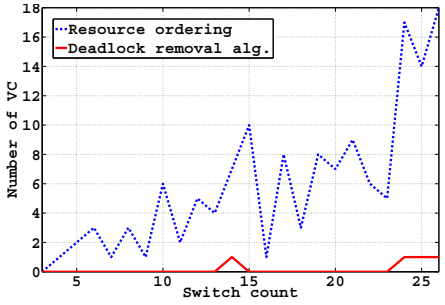

Figure 8: Comparison for $D26\_media$

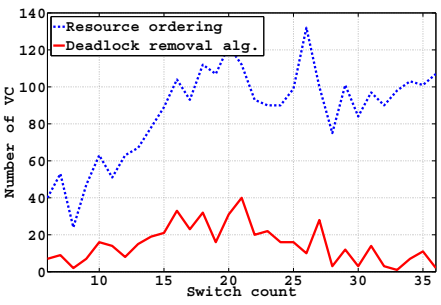

Figure 9: Comparison for $D36\_8$

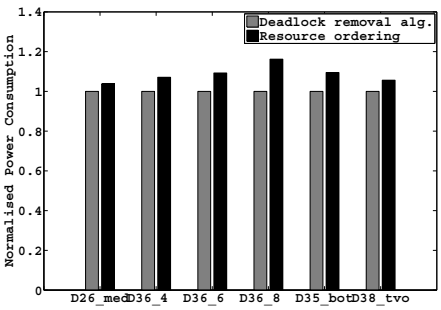

Figure 10: Power comparison

method is general and the input topologies could be either manually designed or obtained using any existing synthesis tools. For comparisons, we also apply the *Resource ordering* technique, a popular method to avoid deadlocks in wormhole routing networks [10]. In this method the communication channels are given a resource number. After a flow uses a channel, the next channel that it acquires needs to have a resource number higher than the current channel.

As a case study, we consider a SoC design for multimedia and wireless applications, referred to as $D26\_media$ that has 26 cores. We use this benchmark to generate topologies for different switch counts and then apply our algorithm and resource ordering to remove deadlocks from the topologies. The results are presented in Figure 8. The dotted line in the plot represents the number of extra VCs that were added to the topology in order to generate the necessary number of resource classes required to use resource ordering on the routes of all the flows. The number of classes needed for a flow depends on the length of the route and that leads to considerable overhead. The solid line represents the overhead of our method. As it can be seen, for most topologies the overhead is zero because the fixed routes on the initial topology do not lead to deadlocks. This result is significant because it also shows that an application specific topology can be deadlock free even without applying restrictions on the routing function. For a better comparison, we ran a similar test on a benchmark with more complex traffic patterns. The $D36\_8$ is a multi-media benchmark that has 36 processing cores. Each processing core sends data to eight other cores. The results of the experiments are presented in Figure 9.

For power and area estimations, we use the models for switches from [20]. The NoC power consumption for the different benchmarks is presented in Figure 10. A description of the benchmarks is given in [21]. The plot shows the relative power consumption overhead for the resource ordering method when compared to our deadlock removal algorithm. The values reported in the plot are for topologies with 14 switches. As can be seen from the figure, our method incurs a significant reduction in power consumption (an average of 8.6%) and in our experiments we observed a large reduction in NoC area (an average of 66%) when compared to the resource ordering method.

We also compared the power consumption of the topologies after removing the deadlocks with the original designs where deadlocks were not removed. From the experiments, we observed only a small overhead on power (of less then 5%) for the deadlock removal method. In practice our algorithm runs fast. We ran our experiments on a 2GHz Linux machine. The method runs with in minutes even for the largest benchmark and it is scalable.

6. CONCLUSIONS

Removing deadlocks in *Networks on Chips (NoCs)* with minimum area-power overhead is a major challenge in application-specific NoCs with custom topologies and routing patterns. In this work, we presented a method to remove the conditions that can lead to deadlocks with minimum area-power overhead. The application communication patterns are used to minimize the number VCs (or physical links) that need to be added to remove the deadlock conditions. The method can be applied any arbitrary NoC topology and routing function. Our experiments show that the method leads to large reduction in NoC area and power consumption overhead when compared to existing schemes. We also found that the method has less than 5% area, power overhead when compared to designs that do not support any deadlock removal method, thereby making it very practical.

7. ACKNOWLEDGMENT

We would like to acknowledge the financial contribution of CTI under project 10046.2 PFNM-NM and the ARTIST-DESIGN Network of Excellence.

8. REFERENCES

[1] G. De Micheli, L. Benini, “Networks on Chips: Technology and Tools”, Morgan Kaufmann, First Edition, July, 2006.
[2] J. Hu et al., ’Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures’, Proc. DATE, March 2003.
[3] S. Murali, G. De Micheli, “SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs”, Proc. DAC 2004.
[4] A.Pinto et al., “Efficient Synthesis of Networks on Chip”, ICCD 2003, pp. 146-150, Oct 2003.
[5] A. Hansson et al., “A Unified Approach to Mapping and Routing on a Combined Guaranteed Service and Best-Effort Network-on-Chip Architectures”, Technical Report No: 2005/00340, Philips Research, April 2005.
[6] K. Srinivasan et. al., “A low complexity heuristic for design of custom network-on-chip architectures”, Proc. DATE 06, pp. 130-135.
[7] X.Zhu, S.Malik, “A Hierarchical Modeling Framework for On-Chip Communication Architectures”, ICCD 2002, pp. 663-671, Nov 2002.
[8] J. Xu et al., “A design methodology for application-specific networks-on-chip”, ACM TECS, 2006.
[9] S. Murali et al., “Designing Application-Specific Networks on Chips with Floorplan Information”, pp. 355-362, ICCAD 2006.
[10] William J. Dally and Brian Towles, “Principles and Practices of Interconnection Networks”, Morgan Kaufmann, 2004
[11] Ni, L.M. McKinley, P.K., “A survey of wormhole routing techniques in direct networks”, Computer, 1993
[12] Duato, J., “A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks”, IEEE Transactions on Parallel and Distributed Systems, Volume 6, Issue 10, Oct. 1995 Page(s):1055 - 1067
[13] G.-M. Chiu., “The odd-even turn model for adaptive routing”, IEEE Transactions on Parallel Distributed Systems, 2000.
[14] R. Cypher and L. Gravano, “Requirements for deadlock-free adaptive packet routing”, Proc. 11th ACM Symp. Principles Distributed Computing, 1992.
[15] DH Linder, JC Harden, “An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes”, IEEE Transactions on Computers, 1991
[16] C.J. Glass and L.M. Ni, “Maximally fully adaptive routing in 2D meshes”, Proc. Conf. Parallel Processing, Aug. 1992.
[17] L Zakrevski et. al., “A new method for deadlock elimination in computer networks with irregular topologies”, Proc. IASTED Conf. PDCS, 1999
[18] D. Starobinksi et al., “Application of network calculus to general topologies using turn-prohibition”, IEEE/ACM Transactions on Networking, Vol. 11, Issue 3, pp. 411-421, June 2003.
[19] Palesi et al, "Design of bandwidth aware and congestion avoiding efficient routing algorithms for Networks-on-Chip Platforms", In Proc. Intl. Symp. on Networks-on-Chip, 2008.
[20] A.B. Kahng et. al., “ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration”, Proc. DATE 2009
[21] S. Murali et. al., “Synthesis of Networks on Chips for 3D Systems on Chips”. ASPDAC 2009, pages 242-247.