Designing Best Effort Networks-on-Chip to Meet Hard Latency Constraints

CIPRIAN SEICULESCU, EPFL
DARA RAHMATI, Sharif University of Technology
SRINIVASAN MURALI, iNoCs
HAMID SARBAZI-AZAD, Sharif University of Technology
LUCA BENINI, University of Bologna
GIOVANNI DE MICHELI, EPFL

Many classes of applications require *Quality of Service* (QoS) guarantees from the system interconnect. In *Networks-on-Chip* (NoC) QoS guarantees usually translate into bandwidth and latency constraints for the traffic flows and require hardware support in the NoC fabric and its interfaces. In this article we present a novel NoC synthesis framework to automatically build networks that meet hard latency constraints of end-to-end traffic streams without requiring specialized hardware for the network components. The hard latency constraints are met by carefully designing the NoC topology and selecting the appropriate routes for flow using lean best-effort network components. We perform experiments on several *System on Chip* (SoC) benchmarks. We compared against a topology synthesis method with no support for real-time constraints and we show that the proposed method can produce topologies that can meet significantly tighter worst case latency constraints (on average 44%). We also show that the tightest worst case latency can be provided with little overhead on power consumption (on average 8.5%).

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*

General Terms: Design, Algorithms

Additional Key Words and Phrases: NoC, topology, worst case latency, topology synthesis

ACM Reference Format:
Seiculescu, C., Rahmati, D., Murali, S., Benini, L., De Micheli, G., and Sarbazi-Azad, H. 2013. Designing best effort networks-on-chip to meet hard latency constraints. ACM Trans. Embedd. Comput. Syst. 12, 4, Article 108 (June 2013), 23 pages.
DOI: http://dx.doi.org/10.1145/2485984.2485996

1. INTRODUCTION

Networks on Chips (NoCs) use scalable networking principles on a chip scale. They provide better performance, modularity and faster design closure when compared to bus based interconnects. Today, building NoCs that meet hard latency constraints imposed by real-time streams is a major challenge for designers. Several applications have strict requirements on latency for one or more traffic streams. For example, in

The authors would like to acknowledge the financial contribution of European Research Council under project AdG-246810-NANOSYS and project AdG-291125-MULTITHERMAN.
Authors' addresses: C. Seiculescu and G. De Micheli, LSI EPFL Switzerland; email: {ciprian.seiculescu, giovanni.demichli}@epfl.ch; D. Rahmati and H. Sarbazi-Azad, HPCAN, Sharif University of Technology, Tehran, Iran; email: drahmati@ce.sharif.edu, azad@sharif.edu; S. Murali, iNoCs Sarl Lausanne Switzerland; email: murali@inocs.com; L. Benini, DEIS, University of Bologna, Bologna, Italy; email: luca.benini@unibo.it.

DOI: http://dx.doi.org/10.1145/2485984.2485996

radar and avionics applications, packets should be delivered from source to destination within a maximum time interval.

Many classes of applications require some amount of determinism in the behavior of the communication flows. Providing guarantees for communication flows in NoCs is usually referred to as providing *Quality of Service* (QoS) guarantees. Usually, the QoS guarantees translate into bandwidth and latency constraints for the traffic flows. The bandwidth constraints can be met by allocating enough resources, giving a bound on worst case latencies is a bigger challenge. While the guarantees that are required by applications in the multimedia domain are concerned with the average case, and infrequent cases that do not meet the requirements are allowed (also called *soft QoS*); there are applications in safety critical domains (such as automation, aerospace, military) that require hard bounds that must be always met (also called *hard QoS*).

To meet the hard real-time latency constraints, designers use NoC architectures that provide in hardware some form of guaranteed QoS support. Several different schemes are used, such as the use of TDMA *(Time Division Multiple Access)* slots [Goossens et al. 2005; Kopetz et al. 1989], packet priorities [Shi and Burns 2008] and time-triggered communication [Paukovits and Kopetz 2008]. Some of the schemes, such as the use of simple packet priorities, achieve soft QoS guarantees, where an absolute worst-case bound of latency cannot be provided. While some schemes, such as the TDMA based ones, can provide hard worst-case bounds. However, all these QoS based NoC architectures incur additional hardware overhead and/or penalize average performance to provide worst case guarantees. In fact, NoC architectures are usually classified as either "best effort" or "QoS" architectures.

In this article we present a novel NoC synthesis framework to automatically build networks that meet hard latency constraints of end-to-end traffic streams. One key novelty in our approach is that we do not use special hardware mechanisms to meet the QoS constraints, but we build our networks using simple, lean network components, identical to those used for best-effort NoC instantiation. Moreover, the worst-case guarantee is achieved with minimal impact on the average-case performance of the NoC. In other words, we can build a network that provides hard QoS guarantees on end-to-end packet delivery time using a best-effort hardware infrastructure.

While there are many works that have addressed the issue of synthesizing best-effort NoCs to meet the zero-load latency constraints [Pinto et al. 2003; Ho and Pinkston 2006; Ahonen et al. 2004; Srinivasan et al. 2005; Hansson et al. 2005; Zhu and Malik 2002; Xu et al. 2006; Murali et al. 2006], none of them synthesize topologies that can meet hard latency constraints. To the best of our knowledge, this is the first work on NoC topology synthesis that can build NoCs that meet hard worst-case bounds on latency constraints of traffic streams using only best-effort network components. In fact, we target the design of NoCs that use wormhole flow control, with round-robin arbitration at the switches, which are commonly used in most best-effort designs. We leverage mathematical models to compute safe worst case latency bounds on a wormhole based best effort NoC. We integrate these models with the topology synthesis process. Along with meeting the worst-case QoS constraints, the topologies synthesized also meet the average bandwidth and latency constraints, while minimizing power consumption.

Several works [Murali et al. 2006; Seiculescu et al. 2010] have shown the power and area advantage of application specific NoC topologies for *Systems-on-Chip* (SoCs) where the communication patterns are known. Apart from improving power and area through the customization of the interconnect according to application requirements, in this work we show that other performance metrics like hard QoS requirements can be fulfilled. Therefore, the contributions of this article are twofold.

—Through experiments we show that through careful topology design the required worse-case delay bounds can be met using even best-effort NoC hardware.

—We provide an algorithm to automatically design the NoC topology to meet the required worse-case delay bounds.

We perform experiments on several *System on Chip* (SoC) benchmarks. When compared to a topology synthesis methods with no support for real-time constraints, our proposed method can produce topologies that can meet significantly tighter worst-case latency constraints (on average 44%). The results also show that the power consumption and average zero-load latency values of the topologies designed using the proposed scheme are only marginally higher than a synthesis method that does not support real time constraints (on average 8.5%). Indeed, when only certain traffic streams require hard QoS guarantees, the topologies synthesized by our proposed method has negligible power consumption or area overhead.

2. RELATED WORK

An introduction to the NoC principles can be found in De Micheli and Benini [2006]. More aspects for the design of NoCs are summarized Flich and Bertozzi [2010] and Cristina Silvano and Palermo [2011]. Several works have addressed the synthesis of NoC topologies to meet application performance constraints [Pinto et al. 2003; Ho and Pinkston 2006; Ahonen et al. 2004; Srinivasan et al. 2005; Hansson et al. 2005; Zhu and Malik 2002; Xu et al. 2006; Murali et al. 2006]. However, all of these works produce topologies that meet the average latency constraints and not the worst-case ones. Mapping of core to regular topologies is addressed in Hu and Marculescu [2003], Murali and Micheli [2004a, 2004b], and Murali et al. [2005]. However, except for [Murali et al. 2005], QoS is not considered. In [Murali et al. 2005], the authors consider critical streams and map them more efficiently than normal streams. However, they do not provide any bounds on the latencies. Hansson et al. [2005], present a method to synthesize NoCs for a TDMA based architecture. They also consider the design of TDMA slot tables to meet hard QoS constraints. However, the method only applies to NoCs where the underlying architecture uses additional hardware to support QoS. Moreover, with a TDMA scheme, the average performance of the system could be poorer than with a scheme where packet injection times are not so tightly constrained.

There have been many works that provide QoS guarantees by adding specialized hardware to the NoC architecture. The Æthereal NoC, presented in Goossens et al. [2005], adds guaranteed services on top of best effort services to provide QoS support. Kopetz et al. [1989] describe the MARS architecture where TDMA is used to provide QoS guarantees. In Shi and Burns [2008], a priority based scheme is used to provide QoS, and Paukovits and Kopetz [2008] present the concept of a predictable *Time-Triggered NoC*, where QoS is ensured through communication services. Marescaux and Corporaal [2007] present a QoS NoC architecture that uses both best effort and guaranteed traffic, but requires specialized hardware and virtual channels. Many works present improvements and variations [Bolotin et al. 2004; Bjerregaard and Sparso 2005; Bouhraoua and Elrabaa 2006; Feliciian and Furber 2004; Kavaldjiev et al. 2004; Leroy et al. 2005; Mello et al. 2006; Millberg et al. 2004; Mondinelli et al. 2004; Mullins et al. 2006; Radulescu et al. 2005; Rijpkema et al. 2003], but all use specialized hardware to provide QoS. A recent survey of NoCs [Salminen 2008] shows that most NoC architectures do not have specialized hardware for QoS and therefore a method to design NoCs that provides guarantees on the delay of communication flows is necessary.

Lee [2003], presents a method to characterize the real time behaviour of communication flows in best effort wormhole general networks for parallel computing. However this method requires traffic regulation, which is not desirable in a majority of applications. In Rahmati et al. [2009], we presented models to analyze and compute worst

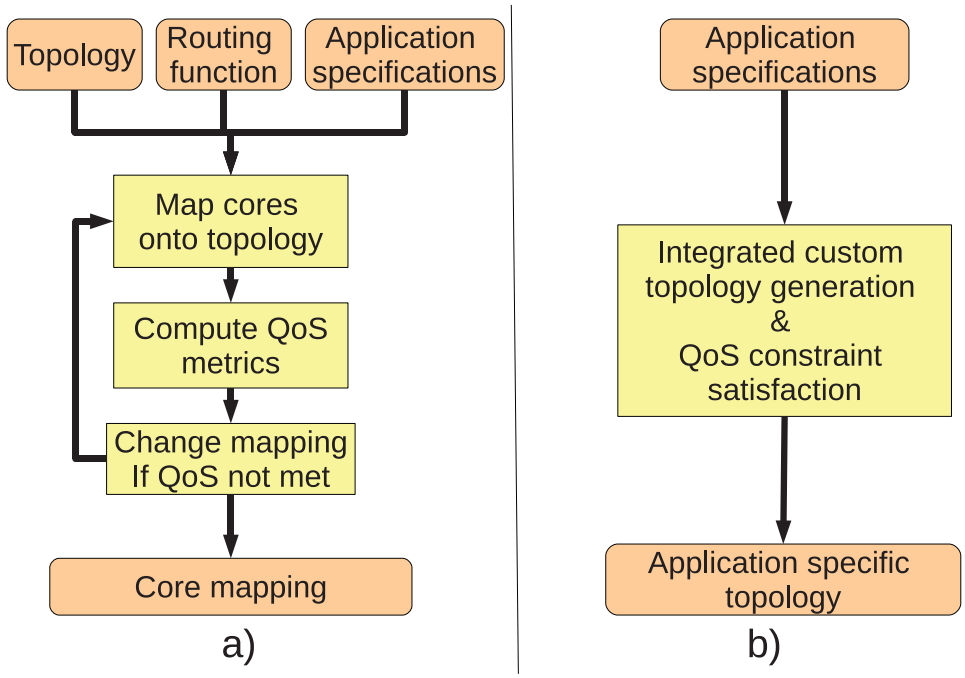

Fig. 1. Algorithm flow: a) Task mapping with QoS, b) Real-time topology synthesis.

case latency bounds, even the input traffic is not regulated. In this work, we use the methods for calculating the worst-case latencies during the topology synthesis process itself, and we use the computed values to tune the instantiation of network components to produce topologies that meet the real time-constraints.

3. REAL-TIME SYNTHESIS COMPARED TO MAPPING ONTO REGULAR TOPOLOGIES

In our previous work [Murali et al. 2005] we showed a method to automate tasks (cores) mapping onto custom topologies with QoS guarantees. There are three main differences between the mapping algorithm and the proposed application specific NoC synthesis algorithm with worst-case delay guarantees. First of all the mapping algorithm takes as input (apart from the application specification) the topology on which to map and the routing function that is to be used. As the mapping problem is constrained by the topology and the routing function the search space is significantly reduced as compared to topology synthesis, where there are no such constraints. Because of the larger search space that needs to be explored it is not trivial to go from the mapping of cores to application specific NoC topology synthesis. As such the synthesis algorithm has to solve new problems like connecting cores to switches, determine the switch to switch connectivity and find routes for flows that meet the worst-case latency requirements.

In Figure 1 we show the main parts of the task mapping algorithm (a) and for the real-time synthesis algorithm (b). As can be seen from the figure the second main difference comes from the way that QoS is provided. In case of the task mapping algorithm, QoS is ensured by checking the constraints for a given mapping and remapping if the constraints are not met. Also given a mapping of the cores, the routes are determined by the routing function. In the case of the topology synthesis algorithm, routing of each communication flow is integrated and with the topology building process and strongly connected to worst-case delay constraint checking. This integrated approach not only

ensures better performance of the designed NoC when compared to the mapping approach, but also give us the ability to build topologies that meet worst-case latency constraints without the need for specialized hardware. The synthesis algorithm also explores solutions with different number of switches to determine the topology that has the lowest power consumption and meets the worst-case latency constraints.

The final difference is related to the QoS model used in the two works. In the mapping algorithm an average case QoS metric is used, as the models are based on simulation. The previous work assumes a sort of soft QoS where traffic is regulated and therefore it is different from the hard QoS metrics we consider in this work. In the present work, the integrated topology design and routing with constraints checking enables the synthesis algorithm to use the extra degrees of freedom not only to design topologies that meet worst-case latency constraints, but minimal power and area as well. In the remainder of the article we will describe in detail the integrated synthesis approach.

4. WORST CASE LATENCY MODELS

If a fair arbitration scheme like round-robin is used in the switches, then the worst-case latency of packets is determined by the topology of the NoC and the routes chosen for the communication flows. A carefully designed NoC topology and well-selected routes can decrease the worst case latency bound and QoS can be provided without the use of extra hardware in the switches. To improve the design methods for designing application specific NoC that can provide real time guarantees, it is important to be able to calculate the worst-case latency of the flows for a given network configuration.

For this work we assume that the switches use round-robin arbitration. For illustrative purposes, we consider an architecture with input-queued switches with no virtual channels, characteristic of many existing NoC designs [Stergiou et al. 2005]. The methods presented here can be easily modified for other switch architectures and to support virtual channels. A credit based (or on-off) flow control is used to provide back-pressure and to prevent the switches from forwarding the *flow control units (flits)* when the downstream buffers are full. The target cores are assumed to be ideal and eject the flits as soon as they reach the target network interface. If a target core is not ideal an end to end flow control mechanism can be used to prevent flits from entering the network when it is backlogged. Buffer size is assumed to be uniform across all the switches. In this work we do not address the issue of buffer sizing, as it is beyond the scope of this article.

In this section, we present a brief description of the mathematical model proposed in Rahmati et al. [2009] to calculate worst-case latencies for a given topology, routing function and traffic flows. In the next section, we show how the model can be used to build the topology and find paths for the given set of traffic flows between the cores of the applications.

The buffer depth of a switch is calculated as the sum of all buffers between the arbitration points of two consecutive switches. Since no traffic regulation is assumed in the model, the worst-case latency is achieved when all buffers are full and when the packet of a flow loses arbitration to all other flows that it can contend with. Under these assumptions, the upper bound on delay for a flow is given by Equation (1) (UB_i represent the upper bound delay for flow i) where ts_1 and ts_2 represent the packet creation and ejection times which are constant. The sum adds the contribution of the worst-case interference at every hop (u_i^j interference of other flows on flow i at switch j) from the path of the flow for which the upper bound delay is calculated. The number of hops on the path of flow i is denoted by h_i.

$$UB_i = ts_1 + ts_2 + \sum_{\forall j} u_i^j \quad with\ j = 0 \ldots h_i. \tag{1}$$

If a core has flows going to different destinations and it is capable of having multiple outstanding transactions, then the packets generated by the same core to different destinations can also contend with one another. Source contention, as it is also called, is modeled by using a virtual switch that does not physically exists on the path of the flow. Equation (2) describes how to calculate the contribution of the source contention.

$$\begin{aligned} &u_i^0 = MAX(U_i^0, U_{I(x)}^0) + \textstyle\sum_{\forall x} U_{I(x)}^0. \\ &with\ x = 0 \ldots z_0(i, 0) \end{aligned} \tag{2}$$

The hop delay from output buffer to output buffer is denoted by U_i^j and $I(x)$ returns the index of a flow from the pool of flows that contend with flow i at switch j. The number of flows that contend with flow i at switch j and use the output port c is denoted by $z_c(i, j)$. Using the same notation, the delays at the rest of the switches on the path of flow i can be calculated with Equation (3).

$$\begin{aligned} &u_i^j = MAX(U_i^j, U_{I(x)}^j) + \textstyle\sum_{\forall x} U_{I(x)}^j \\ &with\ x = 1 \ldots z_c(i, j), \quad 1 \leq j \leq h_i. \end{aligned} \tag{3}$$

The delay of a flow at the current switch U_i^j is calculated in a similar manner, only this time it is based on the delays from the next switch.

$$\begin{aligned} &U_i^j = MAX(U_i^{j+1}, U_{I(x)}^{j+1}) + \textstyle\sum_{\forall x} U_{I(x)}^{j+1} \\ &with\ x = 1 \ldots z_c(i, j+1), \quad 0 \leq j \leq h_i - 1. \end{aligned} \tag{4}$$

To calculate the upper bound delay, the Equations (2), (3), and (4) have to be calculated in a recursive manner. The recursive formulation is guaranteed to complete because the delay of any flow at the last switch in the path is fixed. The termination conditions are given by Equation (5):

$$U_i^{h_i} = L_i, \quad U_{l(x)}^{h_{l(x)}} = L_{l(x)}. \tag{5}$$

The ejection time of a packet at the last switch in cycles for flow i is denoted as L_i.

In Rahmati et al. [2009], we show the correctness of the models and the tightness of the bounds. We also show how the models can be extended to address multiple virtual channels, buffer sizes and packet lengths.

5. TOPOLOGY DESIGN TO MEET WORST-CASE CONSTRAINTS

In this section, we give an example of how topology design can reduce the worst-case delay for some flows when compared to a single switch (crossbar) topology, which is non intuitive. Consequently we will give the intuition of why careful synthesis can potentially reduce the worst case delay of some flows. Intuitively one would expect the crossbar to have the lowest worst-case delay. While that is true for the average worst case delay over all flows, it is not true for individual flows. In many design there are few flows that have hard real-time constraints (e.g., interrupts) and many flows that are best effort (e.g., cache refills). Therefore a multi-switch topology can be optimized to reduce the worst case delay of only those flows that have real-time constraint in the detriment of the other that are best effort.

In Figure 2, we present a simple example of the worst case delay for three flows for a single switch (crossbar) and a two switch topology. Let us assume that flow 1 is a real-time flow while flow 2 and 3 are best effort flows. Also let us assume that the packet size for all flows is 5 flits and the sink is ideal so the ejection latency is 5 cycles. In Figure 2(a) we show the case for the single switch. In this case the three flows that have the same destination will contend for the output port. In the worst case for flow one we have to assume that it would lose the arbitration to both flows 2 and 3. Since the

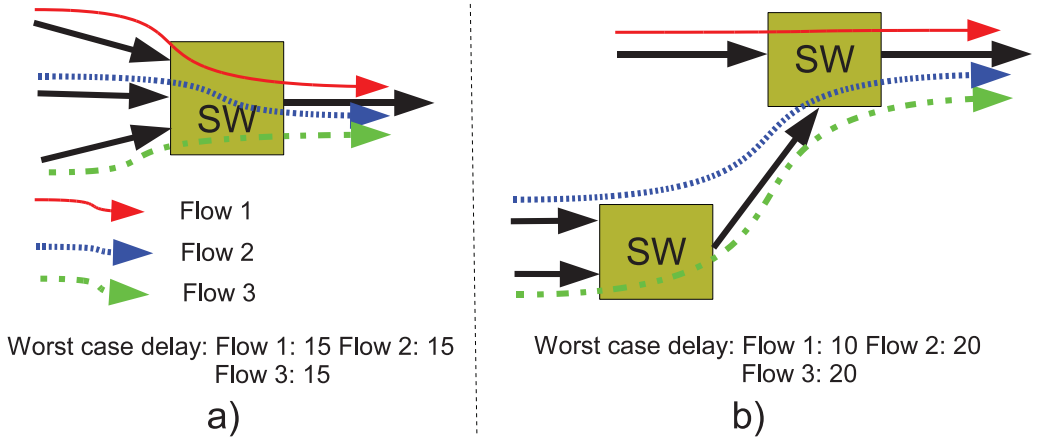

Fig. 2. Example of topologies with worst case delays: a) single switch (crossbar); b) multi-switch.

ejection latency of the flows is 5 cycles and flow 1 could in the worst case wait for the other two flows, it would see a delay of 10 cycles to win the arbitration for the output port. If we add to this the ejection latency of flow 1 we can compute a worst case delay of 15 cycle for flow 1. Since the topology is symmetric we can similarly calculate the same worst case delays for flows 2 and 3 (i.e., 15).

However since only flow 1 is a real-time flow we want to improve the worst case delay of only this flow. In Figure 2(b) we show how that can be done with a two switch topology. In this case flow 2 and flow 3 first contend for the output port of one switch and only the winner will contend with flow 1 for the output of the other switch. So in this case when we calculate the worst case delay for flow 1 we have to assume that in the worst case flow 1 will lose the arbitration to a packet of either flow two or 3 coming from the other switch. So the worst case delay is 10 cycles (5 cycle to win the arbitration and 5 cycles ejection latency). This will however increase the worst case delay of flow 2 and 3. When we calculate the delay of flow 2 we must assume that it will lose the arbitration with flow 3 on the first switch and with flow 1 on the next switch. Add to that the ejection delay and the total worst case delay for flow 2 is 20 cycles. Similarly flow 3 will have 20 cycles worst case delay.

If we look at the average worst case delay over all the flows we see that the single switch topology is better. However since only flow 1 has real time constraint in our example, in the two switch topology the worst case delay of flow one was reduced compared to the single switch case.

6. REAL-TIME NETWORK SYNTHESIS

The goal of the real-time network synthesis algorithm is to find a power-efficient topology that meets the application requirements and the real-time constraints. Therefore the real-time network synthesis algorithm requires the following inputs. The most important input is the communication description of the SoC. The description is given in the form of a directed graph that is defined as follows.

Definition 6.1. 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) with $i \in 1, 2 \ldots n$ and with $j \in 1, 2 \ldots n$ representing the communication between the cores v_i and v_j. The bandwidth of the traffic flow from cores v_i to v_j is represented by $bw_{i,j}$ and the latency constraint for the flow is represented by $rt_{i,j}$.

The number of switches and the values of the architectural parameters (the required operating frequency and the *flow control unit* size) are varied to explore different design points. For a given combination of the number of switches and architectural parameters the real-time network synthesis algorithm designs a topology.

We provide power, area and delay models for the NoC components to the algorithm in order to estimate the power consumption of the topology and to make sure that the architectural requirements are met. For a given technology library, we do synthesis and place&route of RTL code of the NoC components using commercial tools, to obtain the models. Floorplan information can also be provided as input in order to better estimate the wire length and the wire power consumption.

The output of the synthesis algorithm is a topology that minimizes the power consumption and that meets the worst case delay constraints given in the communication specification.

6.1. Real-Time Synthesis Algorithm

The major steps of the method that finds designs the topology for a given combination of the number of switches and architectural parameters, are presented in Algorithm 1. Please note that the step number corresponds to the line number in the Algorithm 1 listing. The algorithm takes as parameters the communication graph (G), the number of switches (num_sw) and the architectural parameters.

ALGORITHM 1: Real_Time_Topology_Synthesis(G, num_sw, architectural parameters)

```
1: set α = 0
2: if α ≥ max_α then
3:    Exit
4: end if
5: assign_cores_to_switch(num_sw, α)
6: for i = 1 to |E| do
7:    Choose next unmapped inter switch flow
8:    set β = α
9:    build_cost_graph(β)
10:   Find min cost path
11:   if path not found or β ≥ max_β then
12:      increment α
13:      Goto step 2
14:   end if
15:   Check RT constraints for all mapped flows
16:   if previously mapped flow violates RT constraints then
17:      if destination(current flow) = destination(violated flow) then
18:         destination_contention(violated flow)
19:      else
20:         path_contention(violated flow)
21:      end if
22:      increment β
23:      Goto step 9
24:   end if
25:   if current flow violates RT constraints then
26:      increment β
27:      Goto step 9
28:   end if
29: end for
30: Save topology
```

The assignment of switches to cores is done by partitioning the cores in blocks (one partition block per switch) using a min-cut partition method. Minimum cost routes for flows are found using Dijkstra's shortest path algorithm in conjunction with turn prohibition for routing level deadlock freedom. These steps are general and use well know algorithms and we will not describe them in detail as they are similar to similar to our previous work [Murali et al. 2006]. However, the customization of Dijkstra's general algorithm, for finding shortest path, to perform routing is done through the cost graph that is fed as input. One important difference from our previous work is the way that the cost are calculated that allows us to drive the path finding routine to create routes that meet the real time constraints. As flows are routed one by one, unlike in our previous work, routing a flow does not influence only its worst-case delay, but it also changes the worst-case delay of the previous flows. So another major difference from our previous work is the iterative approach to find a route that meets it own constraint but also does not cause the previously mapped flows to miss theirs. As part of the iterative approach, there are also some special cases that are treated differently during routing in order to find viable route. These new feature needed for finding routes that meet worst-case delay constraints are described in detail in the following paragraphs.

Since the number of switches is most of the time different than the number of cores, the core to switch connectivity has to be decided (step 5). The core to switch assignment is done by partitioning the cores in as many blocks as there are switches. To perform the partitioning, we define a *Partitioning Graph* (PG) as in our earlier work [Murali et al. 2006].

Definition 6.2. The partitioning graph is a directed graph, $PG(U, H, \alpha)$, that has same set of vertices and edges as the communication graph. The weight of the edge (u_i, u_j), defined by $h_{i,j}$, is set to a combination dependent on α of the bandwidth and the latency constraints of the traffic flow from core u_i to u_j.

The PG is built using the parameter α (initialized to 0 in step 1) which is varied to generate different core to switch assignments when the required worst case delays cannot be met. Initially only the bandwidth influences the core to switch assignment and as α is varied, more importance is given to the real-time constraints of the flows. We use min-cut partition and an existing tool [Hendrickson 1994] to generate the core to switch assignment. The cores in one partition block are connected to the same switch. If the parameter α reaches a certain upper bound, the algorithm exits as it is unable to find a solution.

After the core to switch assignment is decided, the flows between cores that are on different switches (inter switch flows) have to be routed. We loop through the flows (step 6) and we first choose which is the next flow to be routed. The choice of the next flow can be done using several criteria. For example the highest bandwidth flows could be mapped first, or alternatively the flows with the tightest real-time constraint could be mapped first. We use a linear combination of the bandwidth requirements and the real-time constraint requirement to decide the order of the flows. The linear combination is calculated in a similar way to how the weights of the edges in the partitioning graph are calculated.

Since we do not use indirect switches, we have to route the inter switch flows through the switches to which the cores are connected (the number of switches is given as input). To find routes for the flows we do the following: i) we calculate the costs to go from each switch to every other switch; ii) using Dijkstra algorithm we find a minimum cost path, which becomes a temporary route for the flow (the route lists the hops and the ports used at each hop); iii) we test the real time characteristics the found route; iv) if the found route does not fulfill the real time requirements we repeat these steps changing

the cost until a path is found that meets the bound. The details for these steps are described in the following parameters.

A parameter β is used to influence the cost calculation in order to shift the optimization criteria from minimizing power to minimize the worst case delay. In step 8 we initialize the parameter β to the current value of α. This parameter is varied locally to change the optimization criteria only for the current flow that is routed, when a valid route is not found. In step 9 a cost graph is built. The cost graph is defined as follows.

Definition 6.3. The cost graph is a fully-connected directed graph, $C(S, L, \beta)$, where the vertices represent the switches in the topology. The weight of the edge (l_i, l_j), defined by $cost_{i,j}$, gives the cost of routing the flow from switch i to switch j.

The way the weights of the edges in the cost graph are calculated is presented in more detail in Section 6.2. Based on the cost graph the minimum cost path is found in step 10. This path is used as the route for the current flow. We use Dijkstra algorithm and turn prohibition to find deadlock-free minimum cost routes, as presented in Murali et al. [2006]. If a path is not found for any flow or the parameter β has reached the maximum value, then α is incremented and we return to step 2 to retry with a different core to switch assignment.

After a valid route is found for the current flow the algorithm tests weather the real-time constraints are met (step 15). This is done using the worst-case delay models presented in Section 4 recursively. Not only the current flow needs to be tested, but also the previously mapped flows, because the interference of the current flow can cause the already mapped flows to violate their bounds. If the constraints are met then the algorithm proceeds to mapping the next flow.

If there is a bound violation, there are two cases that need to be considered: i) a previously mapped flow has violated its constraint or ii) the current flow has violated the constraint. If a previously mapped flow is the one that violates its worst case delay constraint, there are a further two subcases that need to be considered. One subcase is if the flow that violates the bound and the current flow that we are trying to route have the same destination. This subcase is more complex as it is not possible to fully separate the routes of the two flows and it is handled by the function *destination_contention* (step 18). In the other subcase the contention is along the path of the current flow and the previously mapped flow and the function *path_contention* is called (step 20). The two functions annotate the states of some links so that in the next iteration (after incrementing β, steps 22, 23) the weights in the cost graph are modified such that the conditions that cause the previous flow to violate the constraint can be avoided. A more detailed description of the two functions is presented in Sections 6.3 and 6.4. If the current flow violated the worst case delay constraint then we simply increment the parameter β, and we do a local iteration (steps 26, 27).

If all the inter switch flows could be routed successfully then the topology is saved (step 30) and the algorithm finishes. The power and area of the generated topology is estimated after this point.

The time complexity of the algorithm is $O(|V|^4|E|^3 ln(|V|))$, where $|V||E|^2$ is the contribution given by checking the worst-case latency constraints as presented in Rahmati et al. [2009]. In practice the algorithm runs fast as valid path can be found much earlier when the constraints are not that tight.

6.2. Cost Calculation

Calculating the weights of the edges of the cost graph is an important step, because through the cost assigned to the different edges we can drive the algorithm to find paths that meet the worst case delay constraints and that minimize the power consumption of the topology. The cost calculation for one edge is presented in Algorithm 2.

ALGORITHM 2: cost($switch_i$, $switch_j$, β)

1: Find $link$ between $switch_i$ and $switch_j$ that support the required bandwidth
2: **if** $link$ found and flow_count_on($link$) $\geq$ $BOUND/\beta$ **then**
3: Goto step 1 and find next link
4: **end if**
5: **if** $link$ not found **then**
6: **if** can open new link **then**
7: $link$ = link_count($switch_i$, $switch_j$) + 1
8: cost[$switch_i$][$switch_j$] = marginal_power_new_link($switch_i$, $switch_j$, bandwidth)
9: **else**
10: cost[$switch_i$][$switch_j$] = INF
11: **end if**
12: **else**
13: cost[$switch_i$][$switch_j$] = marginal_power_existing_link($switch_i$, $switch_j$, bandwidth)
14: cost[$switch_i$][$switch_j$] += (flow_count_on($link$) / max_flow_count) $*$ β
15: **end if**
16: **if** link_status[$switch_i$][$switch_j$][$link$] = $PROHIBITED$ **then**
17: cost[$switch_i$][$switch_j$] = INF
18: **end if**
19: **if** link_status[$switch_i$][$switch_j$][$link$] = ADD_EXTRA_COST **then**
20: cost[$switch_i$][$switch_j$] += max_cost
21: **end if**

In the first step the function tries to find a link that exists between the two switches and that has sufficient capacity to accommodate the bandwidth requirement of the new flow. If such a link is found it tests to see if the number of flows already mapped to that link are smaller than a certain bound determined experimentally (step 2). As the bound depends on β, it forces the algorithm to reuses fewer existing links as β is incremented in successive iterations. If the number of flows on the link is larger than the bound then it goes back to step 1 to look for the next viable link.

If an existing link that can be reused was not found, then the algorithm would need to open a new link between those switches. If a new link cannot be opened (because the size of the switches would become too large and they would not support the required frequency) the cost for that edge in the cost graph is set to a large (INF in step 10). This prevents the flow to be routed between the two switches. If the link can be opened then the cost of that link is given by the marginal power increase due to the addition of the new link (step 8). The power consumption increases when a new link is open due to an increase in the size of the switches and the increase in switching activity as the low adds the extra traffic.

If an existing link is found and it can be reused, then the cost of reusing that link is given by the marginal increase power that the new flow creates (step 13). In this case the switch size remains the same, but the switching activity is increase due to the extra traffic. In case a link is reused there are other flows that use that link. We add extra cost that is proportional to the number of flows that are already mapped on that link weighted by the value of the parameter β. In this way we drive the path computation to reuse links that will be shared by fewer flows and even to use new links as the value of β is increased.

Finally based on the status of the link that is set from a previous iteration by the *destination_contention* and *path_contention* functions we can force to prevent communication between some switches (step 17) or to increase the cost in order to favor the use of other links (step 20). The way the link status is set and the function that the link status has, is presented in detail in Sections 6.3 and 6.4.

6.3. Destination Contention

The *destination_contention* function is called, if the route that was found for the current flow, would lead to a previously mapped flow to violate its constraint. Also the contention between the current flow and flow that violates the constraint happens on the output port of the last switch, just before the destination NI. Thus we want to force the new flow to contend with a previously mapped flow that does not have real-time constraints, but that goes to the same output as the flow that violates the constraint. By doing this we increase the worst-case delay of the flow that is not RT and of the new flow, but it would not modify the worst case delay of the flow that now violates the constraint. The steps to achieve this are presented in Algorithm 3

ALGORITHM 3: destination_contention(violated flow)

```
1: ls = last switch for the violated flow
2: for i = 1 to num_sw do
3:   if i ≠ ls then
4:     for j = 1 to link_count(i, ls) do
5:       if link[i][ls][j] is not used by non RT flow that contends with the violated flow at the
         destination then
6:         link_status[i][ls][j] = PROHIBITED
7:       end if
8:     end for
9:     {Prohibit the use of new link}
10:    link_status[i][ls][link_count(i, ls) + 1] = PROHIBITED
11:  end if
12: end for
```

In step 1 we find the last switch (ls) of the flow that was mapped and which violates the constraint. We loop through all the existing switches (step 2) and see if there are existing links to the switch ls. If such links exist, it checks whether it is used by a non real-time flow that goes to the same output as the flow that violates the constraint. If no such flow, exists the link is prohibited from being used in the next iteration by setting the link status to *PROHIBITED* (step 6). If a new link is opened then there is no flow mapped on that link. So if the current flow uses a new link it will still contend with the previously mapped flow at the destination. Therefore the use of new links toward the switch ls is prohibited (step 10).

Example 6.4. In Figure 3 we show an example of how the destination contention is removed. Assume we have the topology from Figure 3(a) and there are three flows. Flow 1 is a real time flow that is routed through switch 3 and 4. Flow 2 is a non-real-time flow and is routed through switch 2 and 4. Flow 3 is the current flow that has to be routed and after the first iteration the path using switch 1 and 4 was found. Suppose that by routing flow 3 through switch 1 and 4 causes flow 1 to violate the worst case delay bound. Therefore the *destination_contention* function will set the state of the links as shown in Figure 3(b). The link between switch 3 and 4 is prohibited, and also opening new links between switch 1 and 4, 2 and 4 and 3 and 4 is also prohibited. The existing link between switch 2 and 4 can be used as it is currently used by flow 2 which does not have real-time constraint but already contends with flow 1. A potential route for flow 3 is shown in Figure 3(c), where flow 1 uses the existing link between switch 2 and 4 that is also used by flow 2.

6.4. Path Contention

If *path_contention* function is called, it means that the route of the current flow intersects the route of previously mapped flow before the destination switch and causes the

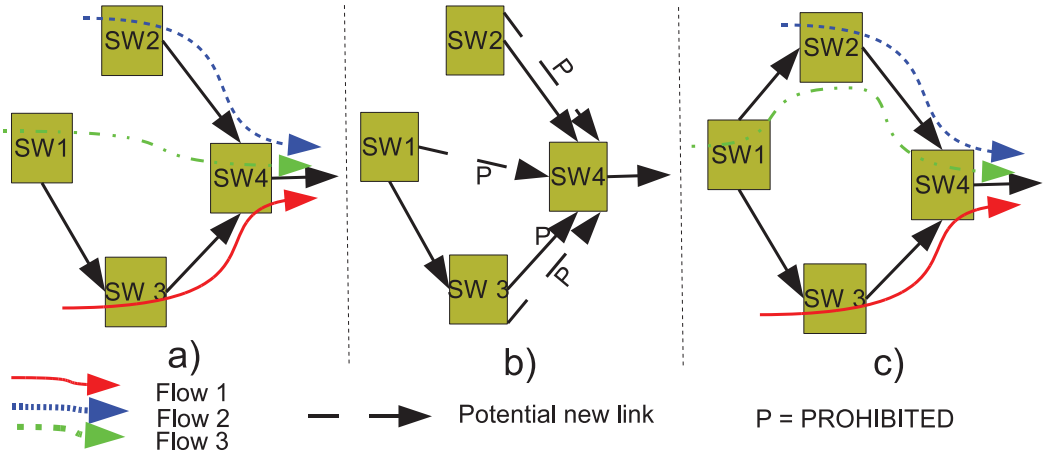

Fig. 3. Destination contention example: a) initial topology; b) link status annotation; c) final topology.

previously mapped flow to violate its bound. In this case we want to drive the path finding algorithm to avoid to use the links that the previously mapped flow uses. If the route of the new flow does not intersect with the previously mapped flow than it will not increase the worst case delay of the previously mapped flow. To that end the links used by the flow that violate its worst-case delay constraint are annotated so that in the next iteration the cost for using such a link is increased.

The steps to annotate the links are presented in Algorithm 4. The function loops through the existing links in the topology and if it finds a link that is used by the flow that violates the bound it set its status to *ADD_EXTRA_COST* (step 5). The value of maximum cost of all edges in the cost graph is added in the cost calculation function to links that have the status set to *ADD_EXTRA_COST*. These links will have high cost and will be avoided by the path finding routine. These links can be used (unlike in the case where the cost is *INF*), if due to other constraints, no other links can be used.

ALGORITHM 4: path_contention(violated flow)

```
1: for i = 1 to num_sw do
2:   for j = 1 to num_sw do
3:     for k = 1 to link_count(i, j) do
4:       if violated flow uses link[i][j][k] then
5:         link_status[i][j][k] = ADD_EXTRA_COST
6:       end if
7:     end for
8:   end for
9: end for
```

Example 6.5. An example of how the path contention is avoided is shown in Figure 4. An example topology is presented in Figure 4(a), and there are two flows. Flow 1 is a real-time flow that is already routed. Flow two is the current flow for which we have to find a route. Assume that in the first iteration the route that is found is from switch 1 to 2, 3 and 4. So flow 2 intersects with flow 1 at the output of switch 1. Assume that by using this route for flow 2 causes flow 1 to violate the required worst case delay bound. In this case the *path_contention* function is called and will set the status of the link between switch 1 and 2 and of the link between switch 2 and 3 to *ADD_EXTRA_COST*

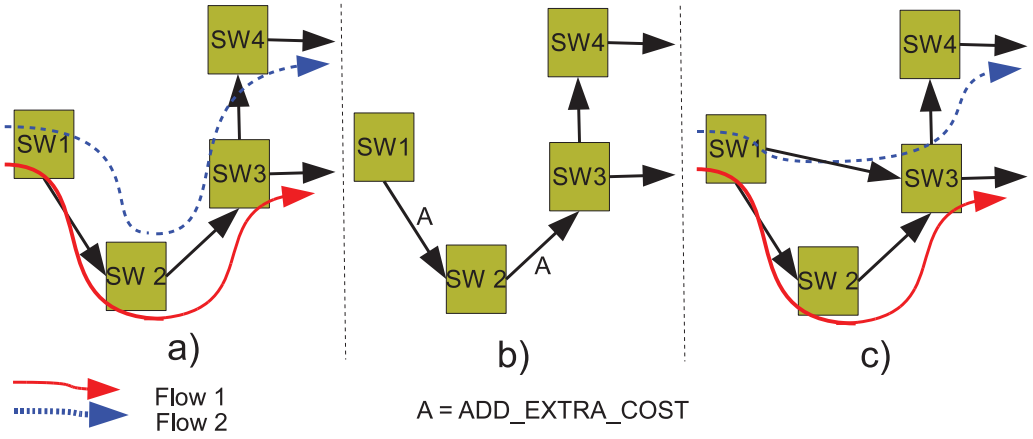

Fig. 4. Path contention example: a) initial topology; b) link status annotation; c) final topology.

as shown in Figure 4(b). Therefore in the next iteration those links will have higher cost and will be avoided if possible. A possible solution is shown in Figure 4(c) where a new link between switch 1 and 3 is opened and the route of flow two uses switches 1, 3 and 4, and therefore the contention with flow 1 is removed.

7. EXPERIMENTAL RESULTS

To evaluate the real-time topology synthesis algorithm, we chose two complex benchmarks extracted from real-life SoC platforms. The first benchmark is a state-of-the-art multimedia and wireless communication SoC [Philips 2004; ST 2004; TI 2004]. The communication patterns for this benchmark are very irregular [Seiculescu et al. 2010], as the described SoC is composed of two subsystems. One subsystem handles the multimedia functions. It contains an ARM processor, hardware video accelerator and a complex memory system that can access off-chip SDRAM and FLASH. The second subsystem is used for the wireless communication and is built around a DSP processor and several peripherals. Communication between the two subsystems is done with the help of a DMA core and several on-chip memories. There are a total of 26 cores that communicate in the system. The second benchmark is a SoC for high-end signal processing applications (such as those used in embedded image and radar processing). This SoC features multiple local memories, having a spread communication pattern. In this benchmark there are 36 cores and each core communicates to four other cores.

We applied our proposed synthesis algorithms on the two SoC benchmarks. For comparisons, we used an existing state-of-the-art synthesis algorithm that does not support hard QoS constraints [Murali et al. 2006] to design the topologies for the benchmarks. We use the methods described in Section 4 to calculate the worst case latencies for the flows for the generated topologies for both cases.

7.1. Effect on Latency

We designed four topologies using the original synthesis algorithm from Murali et al. [2006], where real-time constraints are not considered. The different design points use different numbers of switches: 5, 8, 14, and 20 for the D26_media benchmark. We show results for 5 and 8 switches because, as we will show in the experiments, the worst case latency tends to be worse when there are fewer switches. On the other hand, these design points are important as they provide better power consumption when compared with topologies with many switches. We also give results for topologies with

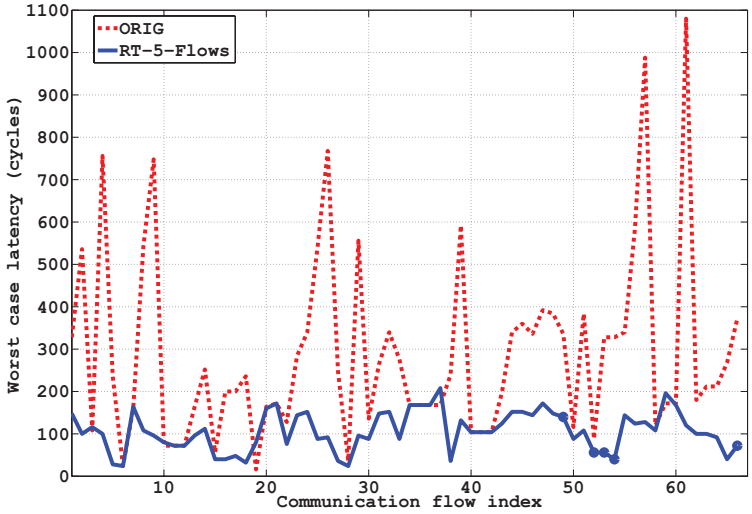

Fig. 5. Worst case latency on each flow.

more switches (i.e., 14, 20) to give a clear picture of how the worst-case latency depends on the number of switches.

We then ran our RT algorithm and found the smallest latency bound for which it was possible to find solutions. While in many real applications only a subset of flows have real time constraints, we wanted to study the maximum overhead incurred by our proposed procedure. Thus, we put real time constraints on all flows. We found that for the D26_media benchmark, the tightest constraint for the upper bound delay for which feasible topologies could be build is 180 cycles. To find out how tight the constraint was, we calculated the worst case latencies of the flows only due to source and destination contentions. To perform this, we connected all the cores through a single crossbar switch and calculated the worst case latencies. On the crossbar, the average worst case latency for this benchmark was 92 cycle and the maximum value across all flows was 148 cycles. This shows that the constraint we imposed is quite tight, as it is only 1.25× the maximum value of the flows from the ideal case.

In Figure 5 we show the worst case latencies for the 66 flows in the D26_media benchmark. The worst case latencies are reported for the case when the crossbar is used and for the cases when a 14 switch topology is designed with the original algorithm and with the RT algorithm. As can be seen, for the topology designed with the RT algorithm, the worst case latency of the flows is in the same range as the worst case latency for the flows mapped on the crossbar. On the topology designed with the original algorithm, most flows have worst-case latency values much higher than those of the crossbar.

Another less intuitive effect that is visible in the plot is that the RT algorithm provides lower worst-case latency than the crossbar. This is because, in a crossbar, each flow will have to contend with all the other flows to the same destination. Whereas, in a multiswitch case, this may not happen. For example, if there are 3 flows to the same destination. In the multiswitch case, two of them may share a path until a point where they contend with the third flow. The third flow only has to wait for one of them (with the maximum delay) to go through. Whereas, in a full crossbar, the third flow will have to wait for both the flows, in the worst case. Thus, we can see that, when only few flows require real time guarantees a multi-switch topology can give

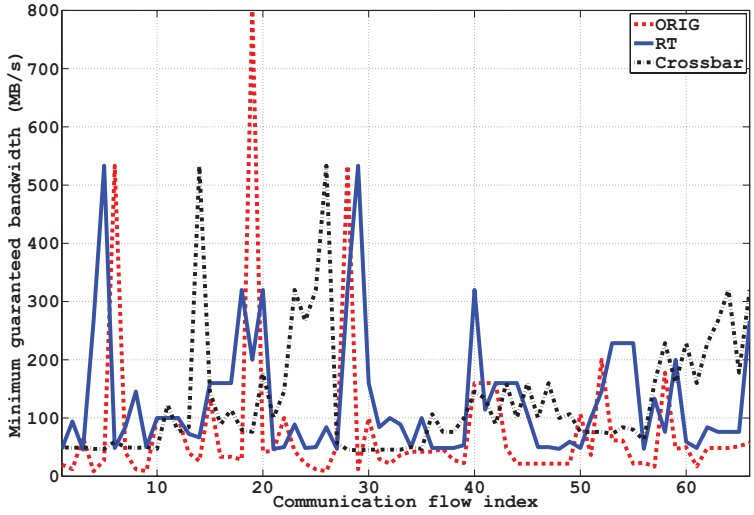

Fig. 6. Minimum guaranteed bandwidth for each flow.

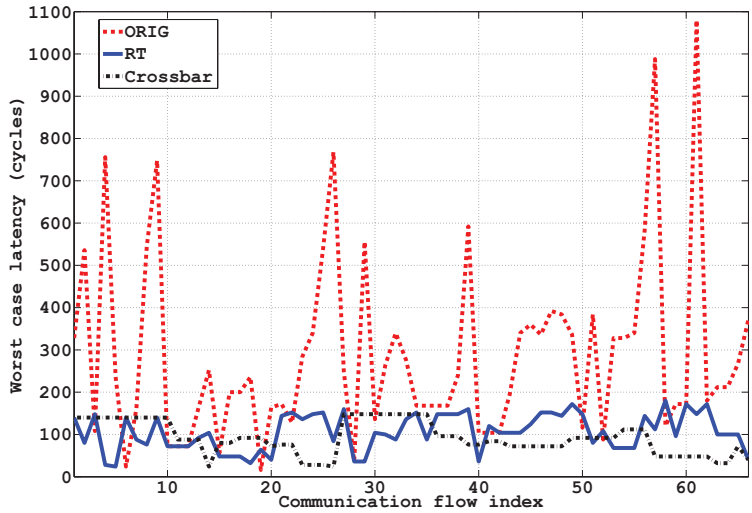

Fig. 7. Worst case latency when only 5 flows are constrained.

better bounds and it is really difficult to come with the best topology directly using designer's intuition. The worst case model from Rahmati et al. [2009] also gives a method to calculate the minimum guaranteed bandwidth under worst case contention in the network. In Figure 6, we show the calculated minimum guaranteed bandwidth for the 66 communication flows for the 14 switch topology.

So far we showed what happened to the worst-case latency when a constraint is set to all the flows. In Figure 7, we show the behavior of the RT synthesis algorithm when only 5 flows have worst-case latency constraints. The flows that had constraints are marked with bubbles on the figure. The latency constraints were added to flows going to and from peripherals. This is a realistic case, as many peripherals have small buffers and data has to be read at a constant rate, so that it would not be overwritten. In this

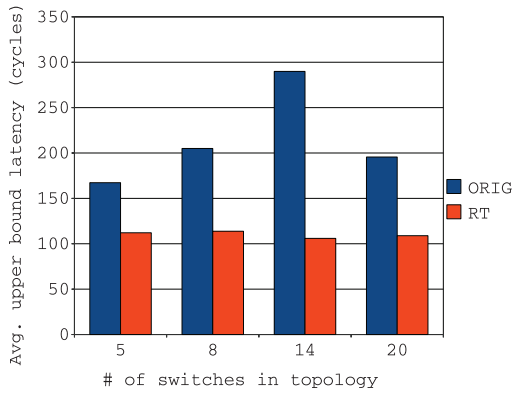

Fig. 8. Average worst case latency for D26_media.

case, the bounds on those 5 flows could be tightened further (two flows at 160 cycles and three flows at 60 cycles). Putting these constraints also leads to a reduction in the worst case latency of other flows as well. Due to the tight constraints, the RT algorithm maps the RT flows first. Then, the unconstrained flows also have to be mapped with more care so that they do not interfere with the previously mapped ones.

In Figure 8, we show the worst-case latencies, averaged over all flows for both the original algorithm and the proposed RT algorithm. From the figure, it can be seen that a synthesis algorithm not considering the maximum latencies can incur a significantly higher worst-case latencies than the required bound especially for low switch counts and we can observe large variations depending on the number of switches in the topology. For the RT case, the variations are much smaller as the algorithm builds the topology in such a way so that all flows meet the constraints. One other important remark is that even in the unconstrained case, increasing the number of switches can help reduce the worst-case latency of flows. This is because, with increasing switch counts, fewer flows share each link, there by reducing the chances of contention. This is contrary to the zero load latency (the latency to go from source to destination without having any interference on the way), which grows with the number of switches in the topology, as can be observed from Figure 9.

For the D36_4 benchmark, we designed topologies with 6, 8, 14 and 20 switches and we found that the tightest constraint for which topologies could be synthesized was 195 cycles. The average worst case latency for the different topologies designed for the D36_4 benchmark are shown in Figure 10. The effects on the average zero load latencies are shown in Figure 11. Using the original algorithm, for 6 switches, the zero load latency is very high as most flows have to reuse existing links and cross more hops. For all other switch counts the zero load latency is smaller, but it constantly increases with the switch count. With the proposed RT algorithm the zero load latency dose not suffer large variations, as the algorithm is reusing fewer links in order to meet the constraints.

7.2. Effect on NoC Components

To design topologies that meet the required worst case latency constraints, the RT algorithm may use more links and therefore additional switch ports when compared to the original synthesis method. By adding more links, contention between flows can be reduced, there by reducing worst-case latencies. But adding more links will increase the switch sizes and the power consumption of the NoC may also increase. It is important to analyze what is the overhead of the RT algorithm over the original method.

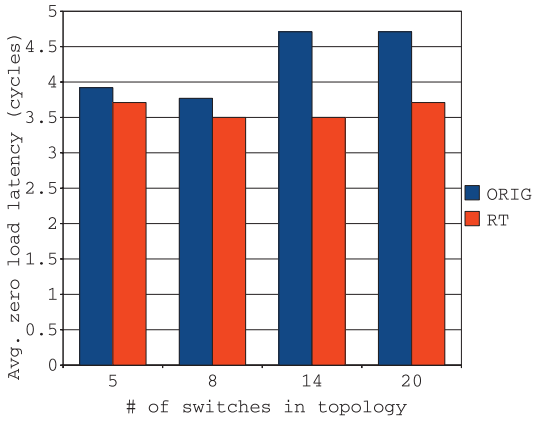

Fig. 9. Average zero load latency for D26_media.

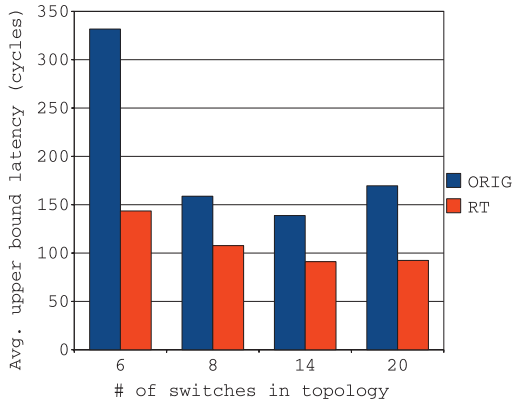

Fig. 10. Average worst case latency for D36_4.

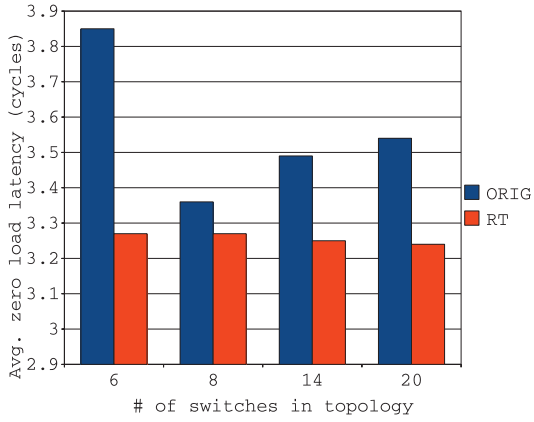

Fig. 11. Average zero load latency for D36_4.

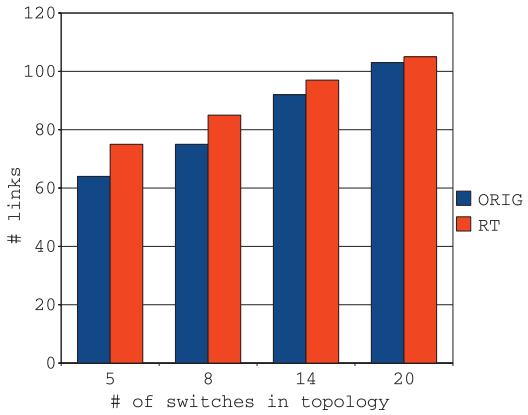

Fig. 12. Number of links for D26_media.

In Figure 12, we plot the total number of links used in the different topologies for the D26_media benchmark. As the number of switches is increased, the number of links used increases as well. With more links, the contention on each link is lower, reducing the worst case latencies. It can be seen that even though the number of links is more in the topologies synthesized with the RT algorithm, the overhead with respect to the original case is on average only 9.5%. In Figure 13 we show the corresponding plot on the number of links in the topology for the D36_4 benchmark. Since the communication pattern of this benchmark involves more flows, we have more links in the original design as well so the bounds can be met with even smaller overhead (average 5.5%).

7.3. Effect on Power Consumption

In this section we analyze the overhead of the RT algorithm with respect to the original algorithm in terms of the power consumption of the NoC topologies. To calculate the power consumption of the designed NoC we used the NoC component models from Stergiou et al. [2005]. The power consumption of the different switches are obtained from placement&routing of the RTL designs in 65nm technology process. Switches of different sizes are synthesized for different operating frequencies and switching activities. Based on the power estimates obtained after place and route, the switch power models are built. After the topology is built, based on the requirements from the communication specifications, the activity at the switches can be determined. Based on the switching activities and the power models, the actual NoC power consumption for the application is calculated.

Our RT algorithm has the biggest impact on the switch power, because by adding more links the size of the switches is increased, leading to an increase in the power consumption. The link power is affected by the link lengths and the switching activity (the amount of bandwidth that flows on the link). The length of the links depends on the floorplan and the parallel links introduced by the RT procedure will have the same length. Since the bandwidth of the flows remains the same as it depends on the application, when you add more parallel links the bandwidth of the flows is distributed among all the links. Therefore in the RT design you have more parallel links at lower switching activity and with the same length and as such the total power for the links is similar. Also the switching activity of the switches remains constant (as the bandwidth is application dependent) even though the RT procedure adds more ports (to add the parallel links). However since the power of the crossbar does not grow linear with the number of inputs and outputs we see an increase in the total switch power consumption.

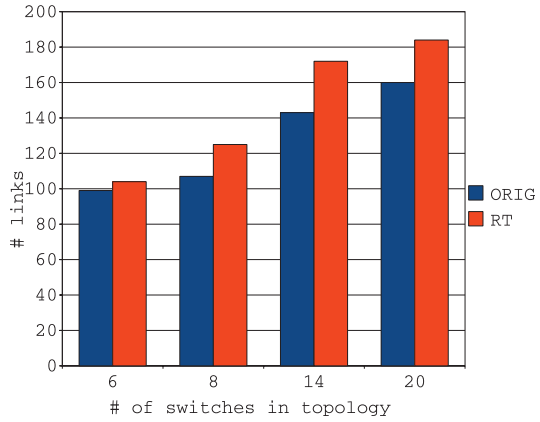

Fig. 13. Number of links for D36_4.

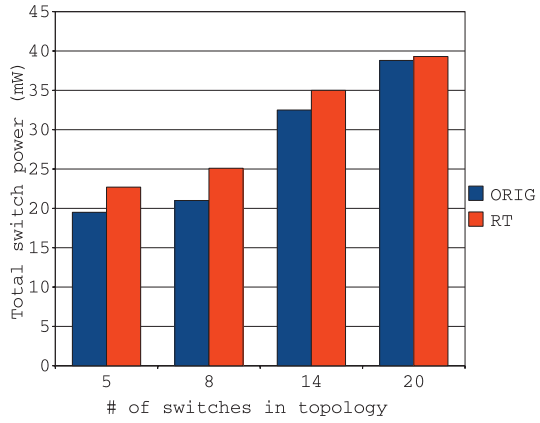

Fig. 14. Switch power consumption for D26_media.

Thus, we only plot the switch power for topologies. The power consumption values for the D26_media benchmark are shown in Figure 14. As can be seen, the switch power increase due to the RT process is only marginal (11% on average). In Figure 15 we show the power consumption for the D36_4 benchmark. The power overhead is slightly lower (6% on average) in this case, as fewer links need to be added to meet the constraint.

Please note that an average the switch power account for about one third of the total power of the topology, the rest being used by the links and the network interfaces (NIs). Both the power of the NIs and of the links is unaffected by the RT procedure and therefore the impact of the RT design on the total power of the topology is minimal, while at the same time providing hard-latency guarantees for the flows.

A summary of all the results for the two benchmarks is presented in Table I. The table reports average upper bund values for flow in cycles, the average zero load latency (in cycles), the number of links in each topology and the corresponding power consumption (in mW).

8. CONCLUSIONS

We have shown that guaranteeing worst-case latencies for communication flows in a best-effort NoC fabric can be done without specific hardware support, by judiciously

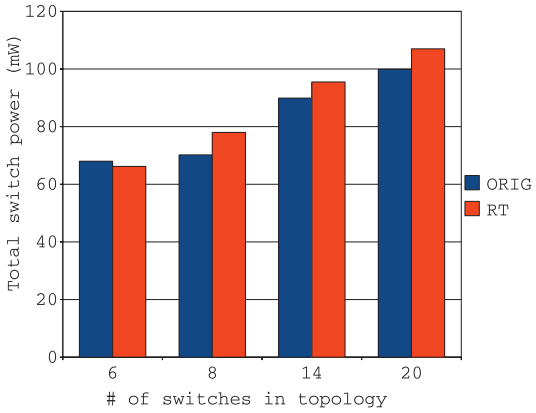

Fig. 15. Switch power consumption for D36_4.

Table I.

Bench-mark	Num of Switches	Upper bound (cycles)		Zero load latency		Num links		Power (mW)	
		RT	ORIG	RT	ORIG	RT	ORIG	RT	ORIG
D26_media	5	112.1	167.3	3.7	3.9	75	64	22.7	19.5
	8	113.8	205	3.5	3.77	85	75	25.1	21
	14	105.9	289.9	3.5	4.71	97	92	35	32.5
	20	108.9	195.6	3.7	4.71	105	103	39.3	38.8
D36_4	6	143.5	331.7	3.27	3.85	104	99	66.2	68
	8	107.7	158.8	3.27	3.36	125	107	78	70.2
	14	91.1	138.8	3.25	3.49	172	143	95.5	89.9
	20	92.4	169.6	3.24	3.54	184	160	107	99.9

synthesizing a QoS-aware topology. The proposed method gives worst case latency guarantees by carefully designing the application specific topologies for the NoC and can be used with switches that do not have any specialized hardware for QoS. The only assumption is that the switches use a fair arbitration policy, such as round robin. In our experiments we show that our proposed algorithm can guarantee meeting real time latency constraints with little resource and power consumption overhead (average 8.5%).

REFERENCES

AHONEN, T., SIGÜENZA-TORTOSA, D. A., BIN, H., AND NURMI, J. 2004. Topology optimization for application-specific networks-on-chip. In *Proceedings of the International Workshop on System Level Interconnect Prediction (SLIP'04)*. ACM, New York, 53–60.

BJERREGAARD, T. AND SPARSO, J. 2005. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. Vol. 2. IEEE, 1226–1231.

BOLOTIN, E., CIDON, I., GINOSAR, R., AND KOLODNY, A. 2004. Qnoc: Qos architecture and design process for network on chip. *J. Syst. Archit. 50*, 2–3, 105–128.

BOUHRAOUA, A. AND ELRABAA, E. 2006. A high-throughput network-on-chip architecture for systems-on-chip interconnect. In *Proceedings of the International Symposium on System-on-Chip*. 1–4.

CRISTINA SILVANO, M. L. AND PALERMO, G. 2011. *Low Power Networks-on-Chip*, 1st Ed. Springer.

DE MICHELI, G. AND BENINI, L. 2006. *Networks on Chips: Technology and Tools* (electronic version). Elsevier, Burlington, MA.

FELICIAN, F. AND FURBER, S. 2004. An asynchronous on-chip network router with quality-of-service (QoS) support. In *Proceedings of the IEEE International System-on-Chip Conference*. 274–277.

FLICH, J. AND BERTOZZI, D. 2010. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC.

GOOSSENS, K., DIELISSEN, J., AND RADULESCU, A. 2005. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Des. Test Comput. 22*, 5, 414–421.

HANSSON, A., GOOSSENS, K., AND RADULESCU, A. 2005. A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. ACM, New York, 75–80.

HENDRICKSON, B. AND LELAND, R. 1994. The Chaco user's guide: Version 2.0. *Sandia* Tech rep. SAND942692.

HO, W. AND PINKSTON, T. 2006. A design methodology for efficient application-specific on-chip interconnects. *IEEE Trans Parallel Distrib. Syst. 17*, 2, 174–190.

HU, J. AND MARCULESCU, R. 2003. Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'03)*. Vol. 1, IEEE, 10688.

KAVALDJIEV, N., SMIT, G., AND JANSEN, P. 2004. A virtual channel router for on-chip networks. In *Proceedings of the IEEE International System-on-Chip Conference*. 289–293.

KOPETZ, H., DAMM, A., KOZA, C., MULAZZANI, M., SCHWABL, W., SENFT, C., AND ZAINLINGER, R. 1989. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro 9*, 1, 25–40.

LEE, S. 2003. Real-time wormhole channels. *J. Parallel Distrib/Comput. 63*, 3, 299–311.

LEROY, A., MARCHAL, P., SHICKOVA, A., CATTHOOR, F., ROBERT, F., AND VERKEST, D. 2005. Spatial division multiplexing: a novel approach for guaranteed throughput on nocs. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. ACM, New York, NY, USA, 81–86.

MARESCAUX, T. AND CORPORAAL, H. 2007. Introducing the supergt network-on-chip: Supergt qos: more than just gt. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. ACM, New York, 116–121.

MELLO, A., TEDESCO, L., CALAZANS, N., AND MORAES, F. 2006. Evaluation of current qos mechanisms in networks on chip. In *Proceedings of the IEEE International System-on-Chip Conference*. 1–4.

MILLBERG, M., NILSSON, E., THID, R., AND JANTSCH, A. 2004. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*, Vol. 2. IEEE, 20890–.

MONDINELLI, F., BORGATTI, M., AND VAJNA, Z. 2004. A 0.13 mu;m 1gb/s/channel store-and-forward network on-chip. In *Proceedings of the IEEE International System-on-Chip Conference*. 141–142.

MULLINS, R., WEST, A., AND MOORE, S. 2006. The design and implementation of a low-latency on-chip network. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'06)*. IEEE, 164–169.

MURALI, S. AND MICHELI, G. D. 2004a. Bandwidth-constrained mapping of cores onto noc architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe.* 20896.

MURALI, S. AND MICHELI, G. D. 2004b. Sunmap: A tool for automatic topology selection and generation for NoCs. In *Proceedings of the Design Automation Conference*. 914–919.

MURALI, S., BENINI, L., AND DE MICHELI, G. 2005. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'05)*. ACM, New York, 27–32.

MURALI, S., MELONI, P., ANGIOLINI, F., ATIENZA, D., CARTA, S., BENINI, L., DE MICHELI, G., AND RAFFO, L. 2006. Designing application-specific networks on chips with floorplan information. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06)*. ACM, New York, 355–362.

PAUKOVITS, C. AND KOPETZ, H. 2008. Concepts of switching in the time-triggered network-on-chip. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*. 120–129.

PHILIPS. 2004. Philips nexperia highly integrated programmable system-on-chip (mpsoc). http://www.semiconductors.philips.com/products/nexperia.

PINTO, A., CARLONI, L. P., AND SANGIOVANNI-VINCENTELLI, A. L. 2003. Efficient synthesis of networks on chip. In *Proceedings of the International Conference on Computer Design*. 146.

RADULESCU, A., DIELISSEN, J., PESTANA, S., GANGWAL, O., RIJPKEMA, E., WIELAGE, P., AND GOOSSENS, K. 2005. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 24*, 1, 4–17.

RAHMATI, D., MURALI, S., BENINI, L., ANGIOLINI, F., DE MICHELI, G., AND SARBAZI-AZAD, H. 2009. A method for calculating hard qos guarantees for networks-on-chip. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD'09)*. ACM, New York, 579–586.

RIJPKEMA, E., GOOSSENS, K., RADULESCU, A., DIELISSEN, J., VAN MEERBERGEN, J., WIELAGE, P., AND WATERLANDER, E. 2003. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEE Proc.: Comput. Digital Techn. 150*, 5, 294–302.

SALMINEN, E., KULMALA, A., AND HÄMÄLÄINEN, T. 2008. Survey of network-on-chip proposals. http://www.ocpip.org.

SEICULESCU, C., MURALI, S., BENINI, L., AND DE MICHELI, G. 2010. Sunfloor 3d: A tool for networks on chip topology synthesis for 3-d systems on chips. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 29*, 12, 1987–2000.

SHI, Z. AND BURNS, A. 2008. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the 2nd ACM/IEEE International Symposium on Networks-on-Chip (NOCS'08)*. IEEE, 161–170.

SRINIVASAN, K., CHATHA, K. S., AND KONJEVOD, G. 2005. An automated technique for topology and route generation of application specific on-chip interconnection networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'05)*. IEEE, 231–237.

STERGIOU, S., ANGIOLINI, F., CARTA, S., RAFFO, L., BERTOZZI, D., AND MICHELI, G. D. 2005. ×pipes lite: A synthesis oriented design library for networks on chips. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*, Vol. 2. IEEE, 1188–1193.

STMICROELECTRONICS. 2004. ST nomadik multimedia processor. http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm.

TI INSTRUMENTS. 2004. TI's omap platform. http://focus.ti.com/omap/docs/.

XU, J., WOLF, W., HENKEL, J., AND CHAKRADHAR, S. 2006. A design methodology for application-specific networks-on-chip. *ACM Trans. Embed. Comput. Syst. 5*, 263–280.

ZHU, X. AND MALIK, S. 2002. A hierarchical modeling framework for on-chip communication architectures. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'02)*. ACM, New York, 663–671.

Received March 2011; revised July 2011; accepted September 2011