A Spectral Clustering Approach to Application-Specific Network-on-Chip Synthesis

Vladimir Todorov†, Daniel Mueller-Gritschneder‡, Helmut Reinig†, Ulf Schlichtmann‡
† Intel Mobile Communications GmbH. ‡Technische Universitaet Muenchen
vladimir.todorov@intel.com daniel.mueller@tum.de helmut.reinig@intel.com ulf.schlichtmann@tum.de

***Abstract*—Modern System-on-Chip (SoC) design relies heavily on efficient interconnects like Networks-on-Chip (NoCs). They provide an effective, flexible and cost efficient way of communication exchange between the individual processing elements of the SoC. Therefore, the choice of topology and design of the NoC itself plays a crucial role in the performance of the system. Depending on the field of application, standard topologies like meshes, fat-trees, and tori might be suboptimal in terms of power consumption, latency and area. This calls for a custom topology design methodology, which is based on the requirements imposed by the application, function and the use-cases of the SoC in question. This work proposes a fast approach, which uses spectral clustering and cluster ensembles to partition the system using normalized cuts and insert the necessary routers. Then, by using delay-constrained minimum spanning trees, links between the individual routers are created, such that any present latency constraints are satisfied at minimum cost. Results from applying the methodology to a smartphone SoC are presented.**

I. Introduction

Network-on-Chip (NoC) interconnects play an increasingly important role in the design and the performance of modern Systems-on-Chip (SoCs) [1]. As technology scales, the number of integrated circuit components on chip increases. This high integration of multiple processing elements (PEs) within the same die, the complex communication patterns between them, and the relatively long distances make point-to-point interconnects and buses infeasible due to scalability issues. Thus, NoCs have emerged as a new design concept, which is able to tackle this problem. The NoC is a packet switched network consisting of multiple routers and links. The PEs are connected to the routers via network interfaces, which encapsulate and decapsulate packets. Scaling of the NoC is easily done by either insertion of more routers or by increasing the sizes of the current ones.

Most commonly the NoC interconnects can be encountered as standard regular structures, such as meshes, trees, and tori. These regular topologies are well suited for general purpose platforms, where the application and its traffic patterns between the PEs, are unknown at design time and cannot be predicted. The regular topologies, however, could prove suboptimal for SoCs with a specific purpose, resulting in a diminished performance and an unnecessary area and wire overhead [2]. Therefore, custom designed NoC topologies are the preferred design choice. However, creating such topologies is not a straightforward task, but a complex combinatorial problem.

The information and requirements provided by the use cases for the SoC can be used both as testing criteria for the custom topology as well as a guide for creating it. Therefore, it is advantageous to make use of this information when constructing the topology. For example, in a mobile phone SoC one use case is as follows. The long term evolution (LTE) subsystem is used together with the video decoder, the memory controller and the sound module to stream and play a video off the Internet. Another is the usage of the camera interface together with the memory controller and the video encoder to capture and store a video. These use cases indicate that the memory controller is a central component and that the LTE communicates less with the video encoder than with the video decoder.

This work proposes a novel, fast and deterministic approach for constructing custom NoC topologies based on a set of use cases with latency constraints. It infers a cost-optimal partitioning from the use cases and determines the appropriate number of routers needed for interconnecting the SoC. This is done by applying spectral clustering, a partitioning algorithm based on eigenvalue decomposition [3]. It is better than a min-cut algorithm as it avoids constructing clusters with just a single member. The clusters for the different use cases are then combined into a final cost-optimal partitioning of the SoC via ensemble learning [4]. The routers are instantiated according to the final partitioning. Finally, delay-constrained minimum spanning trees are used to insert the links between the routers in a cost effective way, which also satisfies the latency constraints, e.g. number of hops. The results for an industrial smartphone SoC show that the algorithm substantially reduces the communication costs, latencies, and the hardware resources in a negligible time (≈ 0.15 seconds).

The paper is structured in the following way: Section II introduces the already existing work on the problem, Section III provides a detailed overview of the synthesis flow, Section IV shows how a use case is partitioned, Section V describes how the final system partitioning is derived, Section VI discusses the delay constrained insertion of links between the resulting routers. Finally, Section VII presents the results obtained from applying the methodology to a smartphone SoC.

II. Related Work

Several works on NoC topology generation already exist. The authors of [1] and [2] present an iterative algorithm, where different parameters such as link-width and number of routers are varied until a topology that optimizes the user objectives is produced. The authors of [5] use linear programming (LP) to approximate the solution to an integer LP formulation for the topology synthesis. The approach is extended in [6] by

978-3-9815370-0-0/DATE13/©2013 EDAA

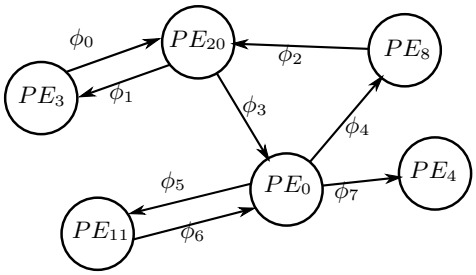


Fig. 1. A use case showing the flows between several PEs

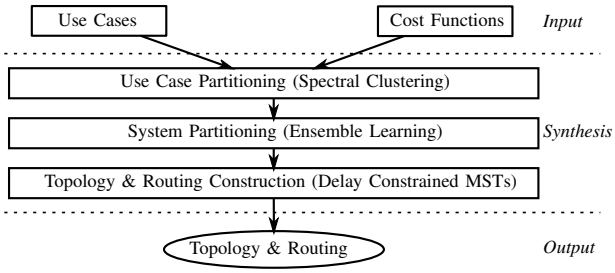


Fig. 2. The proposed synthesis methodology

applying mixed-integer LP. These approaches, however, do not address the issue of multiple use cases and possible latency constraints. An algorithm, which considers multiple use cases, but no latency constraints is presented in [7]. It solves an LP shortest path problem such that the requirements of all use cases are satisfied.

In contrast, the proposed methodology does not only aim at satisfying the requirements of the provided use cases. It exploits them to structure the yet uncoupled PEs in communication clusters. These clusters are used to instantiate the routers and to construct the NoC in a cost optimal way, while keeping the all latency constraints.

III. Synthesis Flow

A use case describes a particular application scenario of the SoC. It can be depicted as a graph $V = (S, \Phi)$, where the nodes S represent the different system PEs and the edges Φ show the flows of information between them (Fig. 1). A flow $\phi \in \mathcal{B} \times \mathcal{L}$ is a tuple described by a maximum bandwidth $\beta \in \mathcal{B}$ and a latency constraint $\ell \in \mathcal{L}$. The new methodology for synthesizing custom NoC topologies is shown in Fig. 2. As an input the methodology takes the different use cases and the applicable cost functions. For each use case the system is partitioned in a cost optimal way using spectral clustering. For each use cases there might be different number of clusters and PEs. Therefore, a consensus on the system partitioning is found by using ensemble learning applied to the results from the previous step. The final step instantiates a router for each partition and connects it to the PEs there. Then it interconnects the routers by using delay constrained minimum spanning trees, such that the latency constraints from the use cases are kept at a minimal cost. In addition, the routing is constructed at the time of link insertion.

IV. Use Case Partitioning

To construct the NoC each use case is partitioned into clusters based on the communication demands between its PEs. Thus, clusters containing PEs with high communication demands between each other are produced. This work approaches the problem of use case partitioning by applying spectral clustering (SC). It is a graph partitioning algorithm, exhaustively used in the field of artificial intelligence. Some of its uses include image segmentation in computer vision [8], [9], blind source separation in signal processing [10] and identifying clusters in bioinformatics [3]. The algorithm partitions an undirected graph, such that the connectivity between the nodes in a cluster is maximized, while the connectivity between the clusters is minimized. It uses the eigen decomposition (spectrum) of the graph's Laplacian matrix. The SC algorithm can be found in various forms [3], [9], [11]. The form used in this work approximates the solution of finding the normalized cuts in a graph (an NP–complete problem [9]) by analyzing its random-walk Lapacian (Sec. IV-B). The advantage of this algorithm is that it approximates the global optimum, rather than a local one [11]. The normalized cut between two clusters (κ_a and κ_b) is defined by Eq. 1, where $cut(\kappa_a, \kappa_b)$ is the total weight of the edges between κ_a and κ_b, and $vol(\kappa)$ is the total weight of the edges within κ. The algorithm approximates the solution of Eq. 2, where $\mathcal{C}$ is the resulting clustering that maximizes the weight within the individual clusters and minimizes the weight between them.

$$Ncut(\kappa_a, \kappa_b) = \frac{cut(\kappa_a, \kappa_b)}{vol(\kappa_a)} + \frac{cut(\kappa_a, \kappa_b)}{vol(\kappa_b)} \tag{1}$$

$$\mathcal{C} = \arg\min_{\hat{\mathcal{C}}} \sum_{\kappa_a \in \hat{\mathcal{C}}} \sum_{\substack{\kappa_b \in \hat{\mathcal{C}} \\ \kappa_b \neq \kappa_a}} \frac{cut(\kappa_a, \kappa_b)}{vol(\kappa_a)} \tag{2}$$

A. Combining Flows

As SC works on undirected graphs, first the directed use case graph (e.g. Fig. 1) has to be converted to an undirected one. This is done by combining all flows ϕ between every two PE_x and PE_y. The combination of flows is done according to Eq. 3–5. If two flows cannot exist at the same time ($\phi_i \perp \phi_j$), the combination of their bandwidths results in maximum of both. In the converse case ($\phi_i \parallel \phi_j$) the resulting bandwidth is the sum of both. The latency is always taken to be the minimum of all combined latencies.

$$\phi_{sum} = \phi_i + \phi_j = (\beta_{sum}, \ell_{sum}) \tag{3}$$

$$\beta_{sum} = \begin{cases} \max(\beta_i, \beta_j), \text{ if } \phi_i \perp \phi_j \\ \beta_i + \beta_j, \text{ if } \phi_i \parallel \phi_j \end{cases} \tag{4}$$

$$\ell_{sum} = \min(\ell_i, \ell_j) \tag{5}$$

Fig. 3(a) shows the directed flows in the use case. In Fig. 3(b) these flows are combined into a single undirected edge.

B. Affinity Matrix and Graph Laplacian

Next, the affinity function of Eq. 6 is applied on the resulting sums of flows. The function is used to compute the attraction of the PEs to each other. It is based on both the cumulative

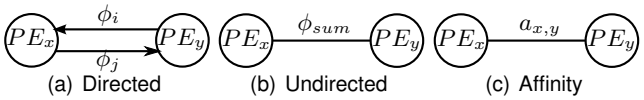


Fig. 3. Use case to undirected graph conversion

bandwidth and latency between two PEs. The coefficients c_0 and c_1 allow scaling the weight of respectively bandwidth and latency.

$$f_a(\phi_{sum}) = c_0\beta_{sum} + \frac{c_1}{\ell_{sum}} \tag{6}$$

An affinity matrix A is constructed by using f_a. The entries $a_{x,y} = f_a(\phi_{sum})$ represent the affinity between PE_x and PE_y within the context of the current use case (Fig. 3(c)).

From the affinity matrix A a diagonal matrix D, representing the weighted degree of each PE, is computed. The entries of D, containing the integrated affinity for the PEs, are computed according to Eq. 7.

$$D(i,i) = \sum_j a_{i,j} \tag{7}$$

Finally, the random-walk graph Laplacian $L_{rw} = D^{-1}A$ is constructed. The name derives from the fact that the entries of L_{rw} can be seen as PE*–to–*PE transition probabilities. The eigen-decomposition $L_{rw} = U\Lambda U^T$ is computed, where U is the matrix containing all the eigenvectors of L_{rw} and Λ is the matrix, which has the corresponding eigenvalues on its diagonal. The mapping $\psi : \Lambda \rightarrow U$ preserves the association of an eigenvalue with its corresponding eigenvector.

C. Number of Clusters and Partitioning

The number of clusters, to be produced by the partitioning, is determined by finding the position of the eigen-gap of L_{rw}'s spectrum. It shows the natural number of clusters χ that occur in the graph [10], [12]. The eigen-gap is found by inspecting the eigenvalues Λ. First, the entries in Λ are sorted in descending order into the vector $\boldsymbol{\lambda}$. Then $\boldsymbol{\lambda}$ is iterated until the eigen-gap $\max(\lambda_{i-1} - \lambda_i)$ is found. Hence, number of clusters χ is computed by Eq. 8.

$$\chi = \arg\max_{i\in[2;n]}(\lambda_{i-1} - \lambda_i) \tag{8}$$

The matrix P is constructed using the χ eigenvectors corresponding to the largest χ eigenvalues. Each row of P represents the $\mathbb{R}^\chi$-space coordinates of a PE from the use case. The final clusters $\mathcal{C}$ are obtained by clustering the rows of P using harmonic K-means [13], deterministically initialized using Kaufman initialization [14] and by using χ as the number of clusters. Algorithm 1 summarizes the steps used to cluster a use case V. If the latency constraint between two PEs does not allow them to reside in different clusters, they are treated as a single PE. Thus, latency constraints are kept irrespective of the affinity function used.

V. System Partitioning

The number of clusters depend on the use case. Thus, different use cases may result in different clusters composed of different PEs. Hence, a *consensus* solution is constructed

Algorithm 1 Use Case Partitioning

```
PROCEDURE: UseCaseSpectralClustering(V = (S, Φ))
BEGIN
for ∀PE_i, PE_j ∈ S do                    //Iterate over all PEs ∈ S
    a_{i,j} ← affinity(PE_i, PE_j, Φ)     //Compute affinity
end for
for i ∈ rows(A) do                        //Compute the diagonal matrix
    D_{i,i} ← Σ_{j∈rows(A)} a_{i,j}
end for
L_rw ← D^{-1}A                            //Compute the random walk Laplacian
[U, Λ] ← eigen(L_rw)                      //Eigenvalue decomposition
λ ← sort(diag(Λ))                         //Sort eigenvalues
χ ← eigengap(λ)                           //Find eigengap
for i ∈ [0, χ − 1] do                     //Get the interesting eigenvectors
    P ← [P, ψ(λ_i)]                       //Create an R^{n×χ} representation
end for
C = HarmonicKmeans(P, χ)                  //Perform clustering
return C
END
```

based on the individual solutions for the use cases. This is achieved by treating the clustering as a multi-learner system and by using cluster ensembles which improve the quality and the robustness of the results [4]. In this work each use case partitioning is treated as a single learner. The system partitioning is achieved by combining the different solutions.

Given a set $\boldsymbol{\mathcal{C}}$ of N clusterings of a system with n PEs, a final consensus solution $\mathcal{C}_F$ is obtained from $\boldsymbol{\mathcal{C}}$, so that the normalized mutual information between $\mathcal{C}_F$ and $\forall\mathcal{C} \in \boldsymbol{\mathcal{C}}$ is maximized. The mutual information is a measure which quantifies the statistical information shared between two distributions [15]. Equation 9 provides an expression for the normalized mutual information between two clusterings $\mathcal{C}_a$ and $\mathcal{C}_b$. In the expression n is the total number of unique PEs in $\boldsymbol{\mathcal{C}}$, n_h^a is the number of PEs in cluster h according to $\mathcal{C}_a$, $n_{a,b}$ is the number of common PEs between $\mathcal{C}_a$ and $\mathcal{C}_b$, n_l^b is the respective number of PEs in cluster l in $\mathcal{C}_b$ and $n_{h,l}$ is the number of shared PEs between clusters h and l.

$$\eta(\mathcal{C}_a, \mathcal{C}_b) = \frac{\sum_{h\in\mathcal{C}_a}\sum_{l\in\mathcal{C}_b} n_{h,l} log\left(\frac{n\cdot n_{a,b}}{n_h^a n_l^b}\right)}{\sqrt{\left(\sum_{h\in\mathcal{C}_a} n_h^a log\frac{n_h^a}{n}\right)\left(\sum_{l\in\mathcal{C}_b} n_l^b log\frac{n_l^b}{n}\right)}} \tag{9}$$

Therefore, Eq. 10 describes the objective function. The final clustering $\mathcal{C}_F$ is the one, which maximizes the sum of its normalized mutual information to the clusterings in $\boldsymbol{\mathcal{C}}$.

$$\mathcal{C}_F = \arg\max_{\hat{\mathcal{C}}} \sum_{z=0}^{|\boldsymbol{\mathcal{C}}|-1} \eta(\hat{\mathcal{C}}, \mathcal{C}_z) \tag{10}$$

Direct optimization of $\mathcal{C}_F$ is infeasible because it is a combinatorial problem. Therefore, a heuristic is used. The cluster-based similarity partitioning algorithm (CSPA) [4] solves the problem by inducing a pairwise similarity measure from $\boldsymbol{\mathcal{C}}$, which is used within a clustering algorithm to produce $\mathcal{C}_F$.

First in CSPA, each partitioning $\mathcal{C} \in \boldsymbol{\mathcal{C}}$ is represented as a matrix. The rows of the matrix represent the different PEs in the system and the columns the different clusters from the partitioning in question. For each row, a 1 indicates a membership of the corresponding PE to a cluster. If the

use cases have different probability weights, the appropriate weights can be used instead of 1. A row of zeros indicates that the PE is not present in the use case. For example, the use case partitioning $\mathcal{C}_i = \{\{PE_0, PE_1, PE_2\}, \{PE_3\}, \{PE_5\}\}$ of a system with 6 PEs is represented by the matrix H_i in Fig. 4. PE_4 is not part of the use case and, thus, it is not part of any cluster.

$$\begin{array}{c} \\ PE_0 \\ PE_1 \\ PE_2 \\ PE_3 \\ PE_4 \\ PE_5 \end{array} \begin{array}{c} \begin{array}{ccc} \kappa_a & \kappa_b & \kappa_c \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

Fig. 4. A matrix H_i of a system consisting of 6 PEs grouped in 3 clusters

Such a matrix is constructed for each $\mathcal{C} \in \boldsymbol{\mathcal{C}}$, corresponding to the solution to a specific use case. All the matrices are then concatenated into a single one $H = [H_0, H_1, \ldots, H_{|\boldsymbol{\mathcal{C}}|-1}]$. This results in $H \in \mathbb{R}^{n \times k}$, with k equal to the total number of clusters in $\boldsymbol{\mathcal{C}}$.

$$S = \frac{1}{||\boldsymbol{\mathcal{C}}||} H H^T \tag{11}$$

From H a new similarity matrix S is computed by multiplication (Eq. 11). The product HH^T provides a measure of how many times two PEs have been grouped in the same cluster. All the diagonal entries of S are set to 0 so that the self-similarity is removed. To obtain the number of clusters and the final partitioning of the system (C_F), SC from the previous section is used with $A = S$. Thus, the clusterings based on the individual use cases are effectively combined into one final partitioning. The solution approximates the optimum of Eq. 10. Algorithm 2 lists the whole process of partitioning a system using a set of use cases $\mathcal{V}$.

Algorithm 2 System Partitioning

```
1: PROCEDURE: ClusterUseCases(V)
2: BEGIN
3: for all V ∈ V do                          //Iterate over all use cases
4:     C_V ← UseCaseSpectralClustering(V)    //Cluster use case
5:     H_V ← ToMatrix(C_V)                   //Convert to matrix
6:     H ← [H, H_V]                          //Extend H
7: end for
8: S ← 1/||C|| H H^T                        //Compute similarity matrix
9: A ← S − diag(S)                           //Convert S to affinity matrix
10: C_F ← SpectralClustering(A)              //Get final clustering
11: END
```

VI. Topology & Routing Construction

For every partition $\kappa_i \in C_F$ a router r_i is instantiated and connected to the PEs in κ_i. The links between the routers are constructed so that the communication cost is minimized without violating the latency constraints in the system. This work uses delay-constrained minimum spanning trees (DCMSTs) to insert links. A DCMST is a MST which satisfies given latency constraints. The general DCMST problem is NP–complete [16]. However, its solution can be efficiently approximated using different heuristics [16], [17]. The algorithm presented here is based on the Kruskal heuristic from [17], but it is able to work with heterogeneous latency constraints.

A minimum of $|\mathcal{C}_F| - 1$ links are needed to interconnect $|\mathcal{C}_F|$ routers in a tree structure. A single tree, however, is not guaranteed to meet all latency constraints. A DCMST solves the interconnection problem for at least one source node (router) and multiple target nodes. Therefore, as set $\mathcal{T}$ of $|\mathcal{C}_F|$ DCMSTs is constructed. Each $T \in \mathcal{T}$ is spanned with a different router r_s acting as a source node. Finally, $|\mathcal{C}_F|$ topologies are constructed by combining the trees in $\mathcal{T}$. The final topology is selected to be the cheapest among the available ones. The following subsections present the detailed steps of the algorithm.

A. Set of Possible Links and Link Costs

The latency constraint and bandwidth requirement between two routers r_a and r_b are computed by combining the flows using Eq. 3–5 between the PEs connected to each router over all use cases (Eq. 12).

$$\phi(r_a, r_b) = \sum_{V \in \mathcal{V}} \sum_{x \in \kappa_a} \sum_{y \in \kappa_b} \sum_{\phi(x,y) \in \Phi} \phi(x, y) \tag{12}$$

A bidirectional link $\zeta(r_a, r_b)$ can only be inserted between the routers r_a and r_b if there exists an inter-router flow $\phi(r_a, r_b) = (\beta(r_a, r_b), \ell(r_a, r_b))$. The set Z contains all these possible links. Each possible link has a delay $d(\zeta) = 1$ because this work assumes the usage of hop-delay. The presented algorithm, however, can work with any delay measure. The cost of each link is computed by Eq. 13. It assigns lower cost to links corresponding to high bandwidth flows.

$$f_{cost}(\zeta(r_a, r_b)) = 1/\beta(r_a, r_b) \tag{13}$$

B. Spanning a DCMST

The Kruskal-based heuristic DCMST algorithm has two stages. Algorithm 3 lists the first stage, where low-cost links are inserted first, if they do not cause latency violations. First, a source router r_s is selected. Next, the shortest path ρ_s between r_s and each other router r_i is computed. This is done by applying Dijkstra's shortest-path algorithm (line 4) on a graph containing all routers as nodes and all possible links in Z as edges. The estimated delay $d_e(r_i)$ between each router r_i and r_s is initialized with the delay of the shortest path ρ_s between the two routers (line 5).

After this, the DCMST graph is set up. Initially, it contains all routers as nodes and no edges (links). Thus, each node r_i is a separate graph component K_i, having a local root $\nu(K_i)$ equal to the node r_i itself (line 6).

Next, all links in Z are sorted in ascending order based on their cost (line 9). The algorithm proceeds by iterating through the sorted links and tries to merge the graph components by inserting the cheapest links. Thus, to minimize the cost, it first tries to put direct connections for the high bandwidth flows. A link $\zeta(r_u, r_v)$ between two routers $r_u \in K_u$ and $r_v \in K_v$ is inserted into the DCMST graph, if $K_u \neq K_v$ and if the two nodes r_u and r_v satisfy the condition of Eq. 14, which assures that no latency constraint is violated.

$$d_s(r_u, r_v) \geq 0 \quad \text{with} \tag{14}$$

$$d_s(r_u, r_v) = \Delta_{min}(r_v) - d(\zeta(r_u, r_v)) - d_e(r_u) \tag{15}$$

The measure $d_s(r_u, r_v)$ shows the worst delay slack left between r_s and the nodes from K_v by inserting the link $\zeta(r_u, r_v)$, under the assumption that r_s will be connected through the local root of K_u. A negative value of d_s indicates a constraint violation. The delay margin $\Delta_{min}(r_v)$ is the minimum available delay slack, left on the paths from r_v to the rest of the nodes in its graph component K_v (Eq. 16).

$$\Delta_{min}(r_v) = \min_{r_h \in K_v} (\ell(r_s, r_h) - 1 - d(r_v, r_h)) \tag{16}$$

In the equation, $d(r_v, r_h)$ is the delay of the path between nodes r_v and r_h in the DCMST graph. At the beginning of the algorithm, $\Delta_{min}(r_v)$ is initialized to $\ell(r_s, r_v) - 1$ if $\phi(r_s, r_v)$ exists, else to ∞ (line 7).

Equation 14 is checked for both directions ((r_u, r_v) and (r_v, r_u)). If Eq. 14 is satisfied in at least one direction, the link $\zeta(r_u, r_v)$ is inserted into the DCMST graph. In this case, the graph components K_u and K_v are merged into a new one $K_w = K_u \cup K_v$ (line 12). For the graph component K_w, resulting from the merging, a new local root $\nu(K_w)$ is selected. It is chosen to be $\nu(K_u)$ if $d_s(r_u, r_v) \geq d_s(r_v, r_u)$, else it is set to $\nu(K_v)$. Thus, the local root providing the better delay slack between r_s and the nodes in K_w is preferred. After that, $\Delta_{min}(r_w)$ and the estimated delay $d_e(r_w)$ are updated for every node $r_w \in K_w$ by, respectively, Eq. 16 and Eq. 17.

$$d_e(r_w) = d(\nu(K_w), r_w) + d_e(\nu(K_w)) \tag{17}$$

Algorithm 3 Insertion of cheap links

```
1: PROCEDURE: DCMST_STAGE1(r_s)
2: BEGIN
3: for ∀r do                                   //Initialize graph components
4:     ρ_s(r) ← Dijkstra(r_s, r)               //Get shortest delay paths from r_s
5:     d_e(r) ← d(ρ_s(r))                      //Set the estimated delay from r_s to r
6:     ν(K_r) ← r                              //Initialize the local root of the component
7:     Δ_min(r) ← ℓ(r_s, r) − 1                //Initialize the worst delay margin
8: end for
9: ζ ← sort(Z)                                 //Sort all possible links
10: for ζ ∈ ζ do                               //Iterate over the sorted links
11:     if d_s(r_u(ζ), r_v(ζ)) ≥ 0 ∨ d_s(r_v(ζ), r_u(ζ)) ≥ 0 then
12:         merge(K_u, K_v)                    //Merge components by inserting the link
13:     end if
14: end for
15: END
```

The second stage (Algorithm 4) merges all the graph components that the first stage was not able to merge. The remaining graph components are connected to r_s by using the shortest delay paths ρ_s. $\mathcal{D}$ is the set of all the local roots of the disconnected graph components. For each of the local roots $\nu \in \mathcal{D}$, the path $\rho_s(\nu)$ between the source r_s and ν is backtracked and the corresponding links are inserted. If loops result, they are destroyed by removing the link to the predecessor (in the DCMST graph) of the node on ρ_s where the loop occurred.

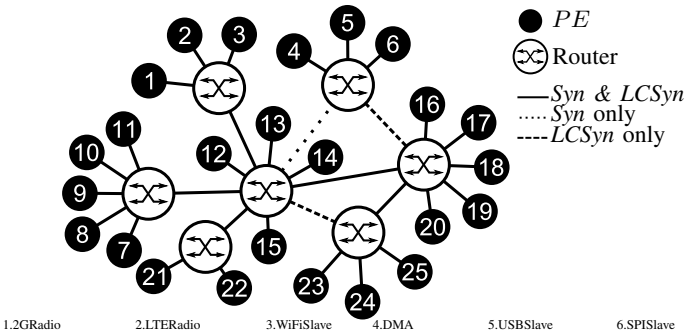


Fig. 5. The synthesized *Syn* and *LCSyn* topologies

C. Constructing the Topologies & Routing

The DCMST algorithm is repeated with each router acting as a source node. Then, for each tree T in the set of solutions $\mathcal{T}$, the remaining trees in $\mathcal{T}$ are sorted according to their difference (in term of links) to T in increasing order. Next, the initial topology is determined by T. If any latency constraints are violated in T, the rest of the trees (in order) are used to insert the alternate links and correct the issue. As the DCMST problem is solved for each router separately, it is guaranteed that the set $\mathcal{T}$ satisfies all constraints.

The routing for the individual solutions is first constructed by the unique paths in T. New paths, added to the solution, are also used to extend and adapt the routing.

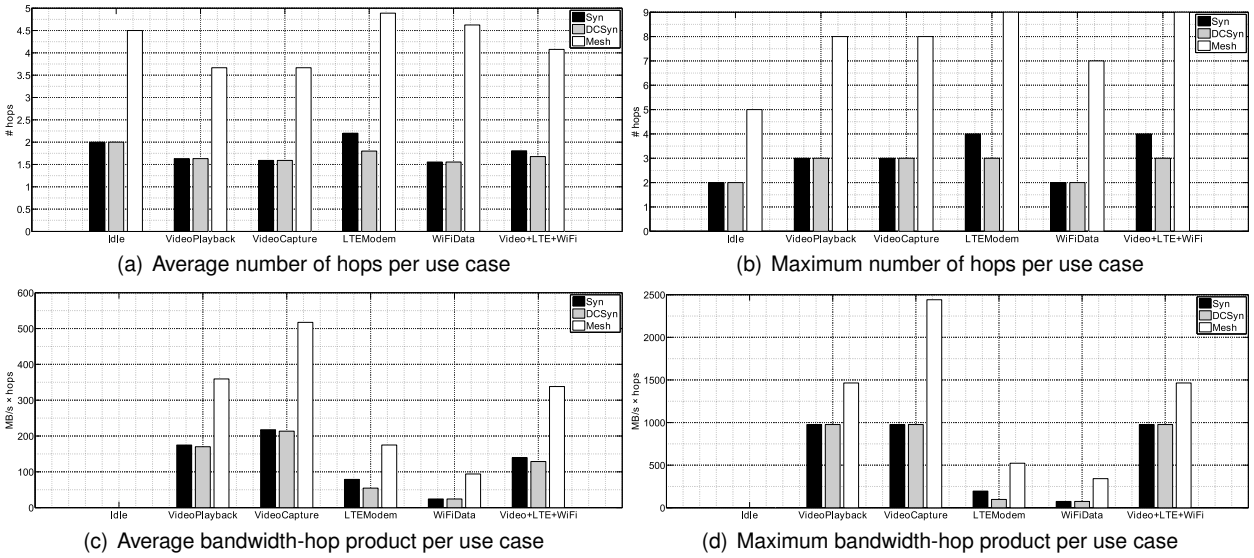
Algorithm 4 Fallback to shortest delay path

```
1: PROCEDURE: DCMST_STAGE2(D)
2: BEGIN
3: for ∀ν ∈ D do                               //Iterate over local roots
4:     ρ = ρ_s(ν)                              //Get lowest latency path to ν
5:     for i ∈ [|ρ| − 2; 0] do                 //Backtrack path
6:         add_link(ρ[i], ρ[i + 1])            //Add new link
7:         resolve_loop()                      //If there is a loop, remove it
8:         if d(ρ_s(ρ[i])) = d(r_s, ρ[i]) then //Reached min. latency?
9:             break                           //Yes, then ν is connected
10:        end if
11:    end for
12: end for
13: END
```

VII. Results

The synthesis approach is applied to an industry design of a smartphone SoC with 25 PEs. The SoC is described by 6 different use cases, depicting the scenarios: idle phone, high-definition (HD) video playback, video capture, LTE communication, Wi-Fi communication, and a combination of HD video playback with LTE and Wi-Fi communications. Two synthesis results with (*LCSyn*) and without (*Syn*) latency constraints are obtained (Fig. 5) and compared to the same system mapped on a 5×5 mesh topology. In both cases f_a is used with $c_0 = 1$ and $c_1 = 0$. Thus, the clustering is performed on bandwidth basis.

Table I lists the resource utilizations of the different solutions in terms of number of routers, number of ports, estimated area, and maximum and average ports per router. *Syn* and *LCSyn* use fewer routers and ports than the mesh. Thus, the synthesized topologies consume much less area than the mesh.



(a) Average number of hops per use case

(b) Maximum number of hops per use case

(c) Average bandwidth-hop product per use case

(d) Maximum bandwidth-hop product per use case

Fig. 6. Comparison of different topologies for different use cases

TABLE I
RESOURCE UTILIZATION

	# routers	*# ports*	Area $[\mu m^2]$	*Max. ports*	*Av. ports*
Syn	7	37	186762.2	9	5.29
LCSyn	7	39	197487.4	9	5.57
Mesh	25	109	572869.4	5	4.36

The synthesized solutions contain some larger, but feasible routers. The average port count per router is comparable for all solutions. *LCSyn* uses more ports than *Syn* because it has more links due to the latency constraints.

Figures 6(a) and 6(b) show, respectively, the average and maximum number of hops for the flows in the different use cases. Both *LCSyn* and *Syn* are comparable and have a much lower hop count than the mesh. Additionally, because of the shorter paths, caused by the latency constraints, *LCSyn* has lower hop count than *Syn* for some use cases. Overall, the synthesized topologies provide better hop-latency and require a significantly lower amount of hardware resources.

To explore the benefits of the synthesized topologies, the bandwidth-hop product is computed for all flows and use cases. It shows the amount of data caused by the use case in the NoC. Figures 6(c) and 6(d) show, respectively, the average and maximum values obtained for the different topologies. Both *LCSyn* and *Syn* exhibit significantly lower bandwidth-hop products. Hence, the traffic is replicated on fewer routers and ports leading to lower power consumption. The difference between *LCSyn* and *Syn* is small because of the system partitioning. There, most of the traffic has been concentrated within the clusters, leaving only small inter-cluster bandwidths.

Finally, the execution time of the overall synthesis flow, without any compiler optimizations, has been measured to be ≈ 0.15 seconds. Compared to the execution times of various algorithms in [7] it makes the proposed approach one of the fastest topology synthesis algorithms.

VIII. CONCLUSION

This paper has presented a novel, fast and deterministic approach for custom NoC topology synthesis based on multiple use cases with latency constraints. The results have shown that using SC with DCMST to partitioning and link insertion provides excellent results, which exhibit significant advantage over a regular mesh topology. Future work will concentrate on expanding the concept and incorporating power domains, floorplanning input and link scaling.

REFERENCES

[1] L. Benini, "Application specific NoC design," in *Proceedings*, ser. DATE '06, 2006.
[2] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing application-specific networks on chips with floorplan information," in *ICCAD '06*, 2006.
[3] U. Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, Dec. 2007.
[4] A. Strehl and J. Ghosh, "Cluster ensembles – a knowledge reuse framework for combining multiple partitions," *Journal on Machine Learning Research*, vol. 3, Dec. 2002.
[5] K. Srinivasan, K. S. Chatha, and G. Konjevod, "An Automated Technique for Topology and Route Generation of Application Specific On-Chip Interconnection Networks," in *ICCAD '05*, 2005.
[6] K. Srinivasan and K. S. Chatha, "A Methodology for Layout Aware Design and Optimization of Custom Network-on-Chip Architectures," in *International Symposium on Quality Electronic Design*, 2006.
[7] G. Leary and K. Chatha, "A Holistic Approach to Network-on-Chip Synthesis," in *CODES+ISSS'10*, 2010.
[8] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Tran. on Pattern Analysis and Machine Intelligence*, vol. 22, 1997.
[9] M. Maila and J. Shi, "A Random Walks View of Spectral Segmentation," in *AI and STATISTICS (AISTATS) 2001*, 2001.
[10] N. Bassiou, V. Moschou, and C. Kotropoulos, "Speaker diarization exploiting the eigengap criterion and cluster ensembles," *Trans. Audio, Speech and Lang. Proc.*, 2010.
[11] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*. MIT Press, 2001, pp. 849–856.
[12] L. Zelnik-Manor and P. Perona, "Self-tuning spectral clustering," in *Adv. in Neural Information Processing Systems*. MIT Press, 2004.
[13] B. Zhang, M. Hsu, and U. Dayal, "K-Harmonic Means - A Data Clustering Algorithm," HP Laboratories, Tech. Rep., 1999.
[14] J. Pena, J. Lozano, and P. Larranaga, "An empirical comparison of four initialization methods for the k-means algorithm," 1999.
[15] T. M. Cover and J. A. Thomas, "Entropy, relative entropy and mutual information," in *Elements of Information Theory*, 1991.
[16] H. F. Salama, D. Reeves, and Y. Viniotis, "An Efficient Delay-Constrained Minimum Spanning Tree Heuristic," in *5th International Conference on Computer Communications and Networks*, 1996.
[17] M. Ruthmair and G. R. Raidl, "A Kruskal-Based Heuristic for the Rooted Delay-Constrained Minimum Spanning Tree Problem," in *12th International Conference on Computer Aided Systems Theory*, 2009.