The ReNoC Reconfigurable Network-on-Chip

Architecture, Configuration Algorithms, and Evaluation

Stuart, Matthias Bo; Stensgaard, Mikkel Bystrup; Sparsø, Jens

Published in:
A C M Transactions on Embedded Computing Systems

Link to article, DOI:
10.1145/2043662.2043669

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

The ReNoC Reconfigurable Network-on-Chip: Architecture, Configuration Algorithms, and Evaluation

MATTHIAS BO STUART, MIKKEL BYSTRUP STENSGAARD, and JENS SPARSØ,
Technical University of Denmark

This article presents a reconfigurable network-on-chip architecture called ReNoC, which is intended for use in general-purpose multiprocessor system-on-chip platforms, and which enables application-specific logical NoC topologies to be configured, thus providing both efficiency and flexibility. The article presents three novel algorithms that synthesize an application-specific NoC topology, map it onto the physical ReNoC architecture, and create deadlock-free, application-specific routing algorithms. We apply our algorithms to a mixture of real and synthetic applications and target three different physical architectures. Compared to a conventional NoC, ReNoC reduces power consumption by up to 58% on average.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design*; B.4.3 [**Input/Output and Data Communications**]: Interconnections—*Topology*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: System-on-chip, network-on-chip, routing, configuration, synthesis, mapping

ACM Reference Format:
Stuart, M. B., Stensgaard, M. B., and Sparsø, J. 2011. The ReNoC reconfigurable network-on-chip: Architecture, configuration algorithms, and evaluation. ACM Trans. Embed. Comput. Syst. 10, 4, Article 45 (November 2011), 26 pages.
DOI = 10.1145/2043662.2043669 http://doi.acm.org/10.1145/2043662.2043669

1. INTRODUCTION

Every new CMOS technology generation enables the implementation of larger and more complex systems on a single integrated circuit. This evolution is accompanied by several challenges: The design effort is increasing, the time-to-market is increasing, and the production cost (in particular the nonrecurring engineering cost) is increasing as well.

As envisioned in Magarshack and Paulin [2003], this trend seems to make application-specific integrated circuits infeasible for the main bulk of applications: The development time will simply be too long and the development cost too high.

The article combines and extends two of our previous papers [Stensgaard and Sparsø 2008; Stuart et al. 2009] that respectively appeared in *Proceedings of the 2nd ACM/IEEE International Symposium on Networks-on-Chip* and *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis 2009*. This article provides a collected overview of the ReNoC architecture, challenges in design automation posed by the architecture, solutions to those challenges, and extends the previous papers with more elaborate explanations, more results, and a discussion on the use of ReNoC in a wider range of systems.

Authors' address: M. B. Stuart, M. B. Stensgaard, and J. Sparsø (corresponding author), Technical University of Denmark, Department of Informatics and Mathematical Modeling, Richard Petersens Plads, Building 322, 2800 Kgs. Lyngby, Denmark; email: jsp@imm.dtu.dk.

DOI 10.1145/2043662.2043669 http://doi.acm.org/10.1145/2043662.2043669

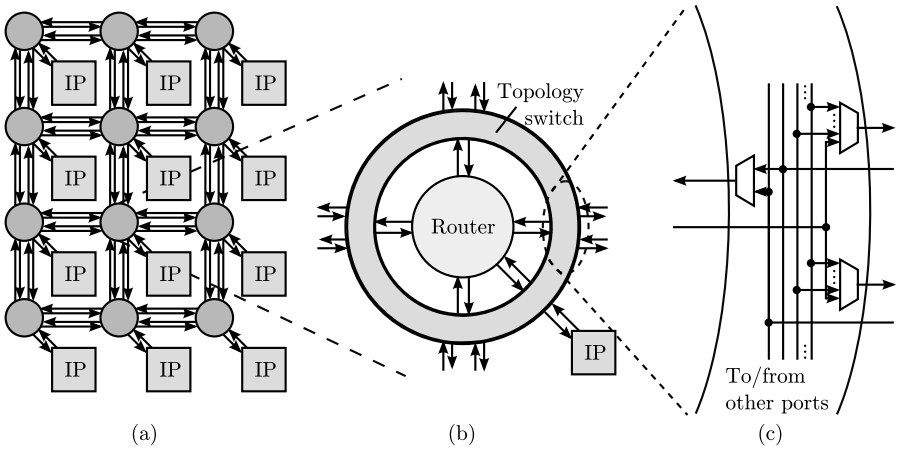

Fig. 1. From left to right: A 3 × 4 2D mesh with double links, a detailed view of a network node with an IP core and a router encircled by a topology switch, and a close-up view of a TS port.

In response to these challenges, today's Systems-on-Chips (SoC) are increasingly being designed in a modular fashion by integrating predesigned components such as general-purpose processors, digital signal processors, hardware accelerators, programmable logic blocks (i.e., FPGA-blocks), memories, and input-output units on a single chip. The term IP core (intellectual property) is commonly used to denote such predesigned blocks that may be provided by several independent vendors.

The interconnect in these IP-based systems has traditionally been implemented as one or more busses, but such ad hoc solutions do not scale well when the number of IP cores increases, and packet switched Networks-on-Chips (NoC) [Benini and De Micheli 2002; Dally and Towles 2001] have emerged as a structured and scalable approach to implementing the interconnect. Figure 1(a) shows such a NoC-based SoC using a 2D mesh network topology. Such a NoC-based and IP-based design approach makes the design effort more manageable, and a large body of research has addressed synthesizing application-specific NoC topologies [Chan and Parameswaran 2008; Murali et al. 2006].

However, in order to increase the production volume, it is necessary to consider more generic platform chips, which can be used (i.e., programmed and/or configured) to implement a range of application-specific systems. A likely scenario is that platform chips will be developed for particular application domains or classes of applications. Using conventional methods, such platform chips will trade off power for the required flexibility in the interconnect: In order to satisfy very varying application requirements, the NoC needs to be oversized and have a regular topology which is less power efficient than a custom or application-specific topology.

The work presented in this article represents an attempt to find a more efficient compromise between regularity and generality on one side and efficiency (area, speed, and power) on the other. The article presents a reconfigurable network-on-chip architecture – called ReNoC – which combines packet switching and physical circuit switching techniques, and which allows (logical) application-specific network topologies to be configured on top of a (physical) platform, whose network topology is more regular. This allows us to provide the required flexibility at the same time as the power consumption in the interconnect reflects the actual needs of the IP cores.

In the ReNoC architecture, the NoC consists of network nodes and network links as shown in Figure 1(a). One of the novel features of ReNoC is that a network node

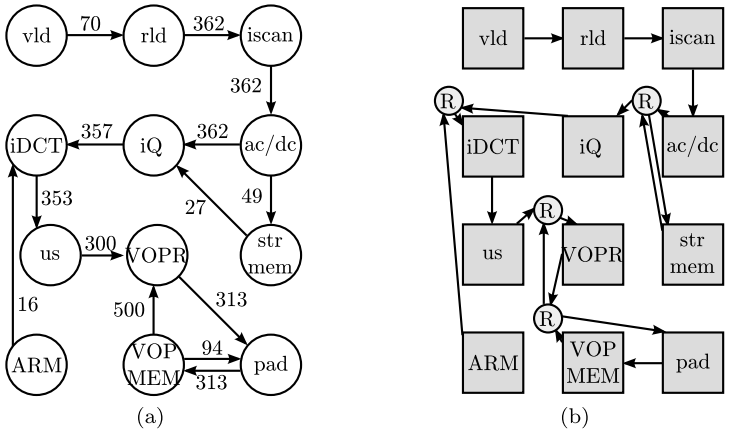

Fig. 2. (a) The VOPD application from Murali et al. [2005] and (b) a logical topology configured on a 3×4 physical architecture.

consists of a traditional packet switched router wrapped in a layer of (reconfigurable) FPGA-style switch boxes, which we denote Topology Switches (TS) due to their functionality, Figure 1(b) and Figure 1(c). In this way, links to and from a network node may be connected to ports on the router or directly to other links, effectively bypassing the router in the node. Figure 2 shows an example of a logical network topology which can be configured on the platform illustrated in Figure 1(a). Unused routers and IP cores are assumed powered down. It should be noted that even though we use a specific router for evaluating ReNoC in this article, the ReNoC architecture is orthogonal to the choice of router architecture; the ReNoC architecture can be combined with any router. The router used in this article is a simple, low-power one in order to keep the comparison between systems with and without ReNoC fair. More details about the router are given in Section 6.

In order to exploit the ReNoC architecture, tools are needed to synthesize configurations for the applications that will run on platforms using ReNoC. In this article, we present three algorithms for solving this configuration problem or optimizing existing solutions. The configuration problem includes synthesizing a logical topology, mapping the synthesized topology to the platform, and deadlock-free routing in the synthesized topology. These parts are described in more detail and formalized in later sections. We evaluate the ReNoC architecture and our optimization algorithms using a mixture of real and synthetic benchmark applications varying in size from 12 to 64 IP cores. Note again that ReNoC is orthogonal to the architecture of the packet switched routers it is combined with. Therefore, issues such as ordering of packets and message-dependent deadlocks are also orthogonal to ReNoC and are not considered in this article.

The preceding discussion focused on MultiProcessor System-on-Chips (MPSoCs) with a NoC interconnect (and this is what originally inspired the work) but ReNoC is equally relevant for more general-purpose single-chip multiprocessors, CMP, [Asanovic et al. 2006; Hammond et al. 1997; Olukotun et al. 1996], and Figure 1 may also be taken as an illustration of a single-chip multiprocessor (CMP). In this context, ReNoC may be seen as extending programmability to include the network topology. This may be used to minimize power consumption as well as communication latency.

Another issue, which is probably more important, and equally important in both CMPs and MPSoCs, is real-time requirements and analysis of timing aspects of a design. In a CMP, as well as in a MPSoC, it is to be expected that several independent applications may be running at the same time, each application possibly running

on its own subset of IP cores. As the NoC is a shared resource, the network traffic related to one application may negatively impact the network traffic related to another application, and this may make it impossible to analyze timing behavior and to guarantee real-time requirements. Here, ReNoC offers the possibility of providing direct, circuit switched connections. Alternatively it is possible to form partially or fully disjoint (logical) subnetworks.

A final issue, which is also important in both CMPs and SoCs, is dynamic runtime reconfiguration. We have just started exploring this issue and the article offers a discussion of the issue.

This article combines and extends two of our previous articles [Stensgaard and Sparsø 2008; Stuart et al. 2009]. We provide additional and more in-depth explanations and extend on the results from these previous papers. We also give a discussion of future uses of ReNoC.

The article is organized as follows: Section 2 discusses related work and outlines our contributions. Section 3 provides a detailed description of the ReNoC architecture, while Section 4 presents our models of ReNoC platforms and applications. Our algorithms for solving the configuration problem are presented in Section 5. Section 6 describes the experiments we use to evaluate the ReNoC architecture and our algorithms, and Section 7 provides the results of these experiments. Section 8 discusses the use of ReNoC in systems, where runtime reconfiguration is required, and the configurations also need to be found at runtime. This section also discusses future work in general. Finally, Section 9 concludes.

2. RELATED WORK

Improving the energy efficiency in NoCs and automating the process of doing so has been considered previously in literature; the novelty of our approach is the combination of energy efficiency and flexibility achieved through a mixture of physical circuit switching and packet switching.

In Ogras and Marculescu [2006], the authors show how to improve the energy efficiency and decrease the traffic congestion in a regular mesh by inserting long, application-specific links. Similarly, in Chan and Parameswaran [2008], point-to-point links are inserted directly between IP cores in an application-specific topology. In both cases, the application-specific topologies are an integral part of the platform design, whereas in ReNoC they are configured at initialization or runtime.

ReNoC's flexibility is achieved through a mixture of packet switching and physical circuit switching. Physically circuit switched NoCs have been proposed in literature [Lee et al. 2005; Wolkotte et al. 2005], but with the restriction that circuit switched connections are used only to make end-to-end connections directly between two IP cores. In ReNoC, a circuit switched connection may also be used to connect IP cores directly, but we also allow circuit switched connections to form long links between routers, that is, combining the best of packet and circuit switching.

An extensive body of research exists on automating design space exploration in NoC-based SoCs with different optimization goals. Researchers have considered the problem of mapping applications onto static NoC topologies [Ascia et al. 2004; Hansson et al. 2005; Hu and Marculescu 2005], where energy consumption is reduced by trying to minimize the average distance in the topology between communicating IP cores. Although this mapping also has an impact on the power consumption in ReNoC, automating the process in the context of ReNoC is left for future work. The methods from literature are not directly applicable, as the mapping has interdependencies with the ReNoC configuration problem that is described later. Other researchers have considered selecting the best topology from a range of topologies or synthesizing a custom topology for a given application [Chan and Parameswaran 2008; Murali et al.

2006; Ogras and Marculescu 2006; Stuart and Sparsø 2007]. These approaches however, can not exploit the high level of flexibility provided by ReNoC. The combination of the application mapping and topology selection or synthesis problems has also been considered [Hansson et al. 2007; Murali et al. 2005]. These approaches have the same weaknesses outlined before when it comes to ReNoC.

Another research direction for reducing power consumption in NoCs considers designing router architectures for low power consumption, for example, Al Faruque et al. [2007] and Modarressi et al. [2009]. These routers may be combined with our work, as the ReNoC architecture is independent of and orthogonal to the router architecture.

Methods for creating deadlock-free routing algorithms in both regular and application-specific networks have been considered previously [Duato 1993; Palesi et al. 2006; Starobinski et al. 2003]. While these methods are not directly applicable to our work, we use a very similar approach to achieve deadlock-free routing.

The previous paragraphs presented the related work. In the following, we outline the contributions of this work. Given: (1) an abstract description of an application, (2) a ReNoC-based SoC platform, and (3) a mapping of the application to IP cores, we solve the following problems: (1) synthesis of a suitable application-specific topology, (2) mapping of this topology onto the given SoC platform, and (3) generating an application-specific, deadlock-free routing algorithm, all three with the aim of minimizing power consumption while satisfying the application's bandwidth requirements. Mapping an application-specific topology to a ReNoC-based SoC platform is a new problem in NoC research. We also refer to the collection of these problems as the configuration problem. We solve the three parts of the configuration problem in one pass in order to avoid situations where, for example, no feasible mapping of the synthesized topology on the SoC platform exists. An example of a ReNoC-based SoC platform is shown in Figure 1, an application in Figure 2(a), and a configuration, a solution to the problem, in Figure 2(b).

The article also presents an extended version of the original ReNoC architecture, in which there are more links than router ports. This allows long-range physically circuit switched and short-range packet switched connections to coexist at the same network node. We evaluate our algorithms and platforms using a mixture of real and synthetic benchmark applications.

Note that we use the term "routing algorithm" in its most general sense of "how to get from A to B" rather than an algorithmic description of a route between any two points. Such a description is generally not possible for application-specific network topologies.

3. SYSTEM ARCHITECTURE

The ReNoC architecture [Stensgaard and Sparsø 2008] was briefly introduced in Section 1 and in Figure 1. In this section, we give a more detailed description of the relevant details, first of the generic ReNoC architecture and then of the specific implementation that we use in our evaluation of the architecture and the configuration algorithms.

ReNoC is a generic architecture for NoCs that is orthogonal to the router architecture. It can be used to add a layer of circuit switching to any packet switched NoC such as MANGO [Bjerregaard and Sparsø 2005a] or ×pipes [Dall'Osso et al. 2003].

By configuring the Topology Switches (TSs) (setting the multiplexers' control signals in Figure 1), long, logical links bypassing routers can be formed, allowing single-hop communication between nonneighboring routers. It is even possible to directly connect two IP cores with such a logical link, giving a physical point-to-point connection. In this way, an application can configure a logical, application-specific network topology. We make the distinction between the *physical architecture* that is the topology in which

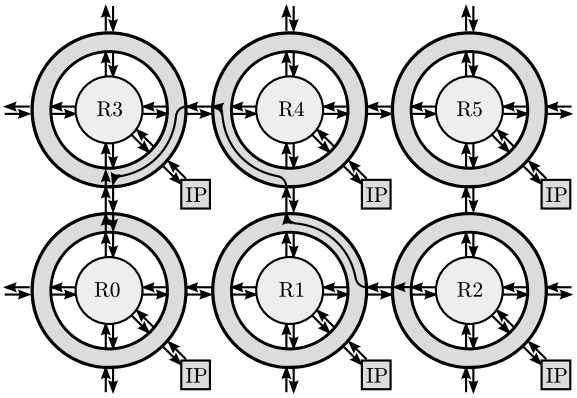

Fig. 3. A logical topology needs not have bidirectional links. Observe the north port of router R0, where the output goes to the south port of router R3, but the input comes from the west port of router R2.

network nodes are connected and the *logical* or *application-specific topology* that is configured on top of the physical architecture. An example of such a configuration is shown in Figure 2 where an irregular, logical topology is configured on top of a 3×4 mesh like the one shown in Figure 1(a).

When configuring the logical topology, care must be taken that the latency of the slowest, long, logical link does not exceed the clock period. Pessimistic models of the links indicate a latency of 120ps for a flit on a 1mm link, thus (with a 100MHz clock) very long, logical links can be formed with no need for pipelining [Stensgaard and Sparsø 2008]. If needed, state holding repeaters can be inserted in all or a subset of the TSs to allow very long, logical links to be pipelined. As NoCs typically employ flow control at the flit level, synchronous latency-insensitive or elastic circuits [Carloni and Sangiovanni-Vincentelli 2002; Carmona et al. 2009] may be used to arbitrarily add pipeline registers without changing the circuits' functionality. If the NoC is implemented using asynchronous techniques [Sparsø and Furber 2001], such insensitivity to the addition of pipeline registers is typically already present.

The detailed view of one of a TS' ports in Figure 1(c) shows a conceptual implementation using multiplexers, but in actual implementations other possibilities exist.

— If the links use low-swing signaling, it is also possible to implement the topology switches using low-swing switches as presented in Dally [2007].
— If reconfiguration is expected to occur infrequently or only at initialization of the SoC platform, implementation styles similar to those used in FPGA switch boxes can be used for the TSs, such as pass-gates, tristate buffers, or multiplexers as shown in Figure 1(c).

As can be seen in the logical topology in Figure 2, the ReNoC architecture does not impose any requirement on bidirectional connections through the NoC. For example, considering R0 in Figure 3, a router's north output port may be connected to the neighboring router's south input port, while its north input port is connected to a long, logical link originating much further away on the SoC platform. Well-known deadlock-free routing algorithms such as up-down routing or turn-prohibition [Starobinski et al. 2003] rely on bidirectional topologies and can thus not be applied to ReNoC in general. We describe our approach to avoiding deadlocks in the next section.

The preceding paragraphs concerned the generic ReNoC architecture. In the following, we present the specific implementation that is used in the evaluation. In this work, we use mesh-based physical architectures, but keep in mind that the ReNoC

architecture is not limited to these. While allowing any circuit switched connection to be established in the TSs would provide the highest amount of flexibility, we impose a few restrictions on the possible connections in order to minimize the overhead. The following circuit switched connections are the ones allowed.

(1) Any link *input* can be connected straight to any link *output* except back in the direction of the link input; no U-turns are allowed. This effectively bypasses the router.
(2) A port on a router may be connected only to the link in the corresponding direction, that is, the router's north port may only be connected to the links on the TS' north port. This goes for both in- and output ports.

The connection to the IP core's network interface is considered a link similar to those connecting neighboring network nodes for this purpose, that is, the IP core can only be connected to the local router on the router's port in the direction of the IP core. However, an IP core may use a long link to connect to any port on a different router, except for the IP port, which can exclusively be used by the local IP core, refer to the preceding item number (2). If an application decides not to use a given TS port, an enable-bit prevents the port from forwarding flits.

4. SYSTEM MODEL

This section describes the models we use for representing applications, physical architectures, etc., and formalizes the configuration problem. Applications are characterized by bandwidth graphs that describe the bandwidth requirements between sets of tasks, where all tasks in a set are mapped to the same IP core. We consider a fixed mapping of the application to IP cores. Using O for the set of IP cores, we have the following.

Definition 1. A bandwidth graph is a directed graph $BG = (T, C)$, where each vertex $t_i \in T$ represents a task set and each directed edge $c_{i,j} = (t_i, t_j) \in C$ represents a connection from t_i to t_j. Each edge $c_{i,j}$ has a weight $b_{i,j}$ that indicates the connection's bandwidth requirement.

Definition 2. A mapping $M : T \to O$ maps a task set $t \in T$ on an IP core $o \in O$. M is assumed to be fixed and given as input for each run of the algorithms.

We also use a graph representation to represent the physical architecture. Most graph representations of networks use vertices to represent routers and edges to represent links between routers. We need to model at a finer level of granularity due to the nature of ReNoC where the output ports a packet can leave a network node on depends on what input port it arrived on. Therefore, we use vertices to represent *ports* on both routers, TSs, and IP cores. We call the sets of router, TS, and IP core ports U, S and O respectively. These are also indicated in Figure 4. We find it useful to distinguish between these three sets, as edges internal to TSs need to be handled differently from other edges in the algorithms: An edge between two router ports indicates that it is possible to come from one port to the other, while an edge between two TS ports indicates that it is possible to come from one input port to the multiplexer on the output port (see Figure 1), but the setting of the multiplexer control signal determines which input port is actually connected to the output port. These multiplexer control signals are what are actually set in order to configure a logical topology.

Definition 3. A network graph is a directed graph $NG = (P, L)$ where the set of vertices $P = U \cup S \cup O$ is the union of the sets of router, TS, and IP core ports, and each directed edge $l_{i,j} = (p_i, p_j) \in L$ represents a link from p_i to p_j. Note that we

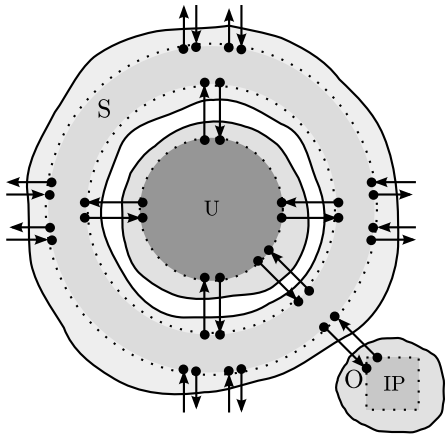

Fig. 4. The vertices contributing to the sets O, S, and U from a single network node. The full sets are found by taking the union of these subsets over all network nodes. Edges internal to the TS and the router are not shown.

use the term "link" to describe any edge in the *NG*; when we are referring to links between network nodes, we will use the term "NoC links." For IP cores, we discern between input and output ports using subscripts, for example, $o_{i,in}$. We associate two parameters with each link $l \in L$.

(1) An energy per packet, e, that denotes the amount of energy expended in transmitting a packet along l. This not only covers the energy consumption in inter-node links but also the energy consumed internally in routers and TSs, that is, in the buffers and the switch in routers and in the multiplexers in the TSs.
(2) A capacity q that denotes the sustainable throughput of the link, which is a fraction of the peak throughput, $q = \alpha \times \texttt{peak}, 0 < \alpha \leq 1$. In practice, α represents the saturation load of the network. By setting the available capacity on a link to α, we ensure that the saturation point is never reached. This approach is equivalent to the one used in Murali et al. [2006], where the bandwidth requirements of connections in the application model are increased until simulations show that the synthesized NoC can support the actually required bandwidth.

In general, for a graph $G = (V, E)$, given two vertices $u, v \in V$ and an edge between them $e = (u, v) \in E$, we specify that u is the source and v the destination of e, $\text{src}(e) = u$, $\text{dst}(e) = v$.

Routes are paths in the network between pairs of IP cores. In the context of our graph representations, a route contains all ports that a packet passes through, not only the ones where actual routing decisions are made. The routes will be trimmed down to the necessary parts as a postprocessing step.

Definition 4. A route $\mathcal{R}(o_i, o_j)$ between the IP cores $o_i, o_j \in O$ is a path $\langle p_0 p_1 \dots p_{n-1} \rangle$ where $p_0 = o_{i,out}$ and $p_{n-1} = o_{j,in}$. We define the route servicing a connection $\mathcal{R}(c) = \mathcal{R}(M(\text{src}(c))_{out}, M(\text{dst}(c))_{in}), c \in C$, that is, as the route originating at the IP core the source of the connection is mapped to and terminating at the IP core the destination of the connection is mapped to. The notation $\mathcal{R}_i$ indicates the ith element in $\mathcal{R}$. The set of all routes is denoted $R = \{\mathcal{R}(c) | c \in C\}$.

A routing deadlock is characterized by a cyclic dependency of flits in the network. We can determine if a deadlock is possible by analyzing if the set of all routes form a

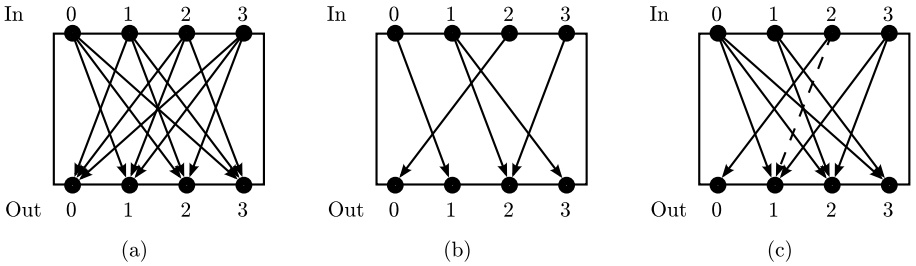

Fig. 5. Care must be taken when unconfiguring links. In (c), the dashed link should not be added when unconfiguring (0, 1), as this would inadvertently unconfigure (2, 0).

cycle in a graph. To do so, we make use of a dependency graph that is similar to the application-specific channel dependency graph in Palesi et al. [2006].

Definition 5. A dependency graph is a directed graph $DG = (P, D)$ where the set of vertices P is identical to that in the NG, but each directed edge $d_{i,j} = (p_i, p_j) \in D$ represents a dependency by p_i on p_j. A dependency $d_{i,j}$ signifies that $\exists \mathcal{R} \in R \exists i. \mathcal{R}_i = p_i \wedge \mathcal{R}_{i+1} = p_j$, that is, there exists a route where p_j is the immediate successor of p_i: If a flit arrives at p_i, it may need to proceed to p_j.

In order to describe a solution to the configuration problem we use a configuration graph that has the same vertices as the NG but whose edges are a subset of those in the NG. Specifically, internally to TSs we only include those edges that correspond to the connections between ports, that is, an output port has multiple incoming edges but in the configuration graph we only include the one from the input port that is selected by the multiplexer shown in Figure 1(c). We allow partial configurations meaning that the configuration graph contains multiple edges for one TS port. This is useful when building solutions iteratively.

Definition 6. A configuration graph is a directed graph $CG = (P, A)$. The two edge properties are copied from the NG for each link $a \in A$.

An edge $a \in A$ in a TS is said to be *configured* by removing the other edges incident on src(a) or dst(a) and internal to the TS (the links that are not selected with the multiplexer control signal), and to be *unconfigured* by adding (some of) these edges back. When unconfiguring, care must be taken to not inadvertently unconfigure a different edge than the one actually being unconfigured. Consider Figure 5, where a TS with four input and output ports is shown. In Figure 5(a), the unconfigured TS is shown, in Figure 5(b), the links (0, 1) and (2, 0) have been configured, while Figure 5(c) shows what will happen if (0, 1) is then unconfigured. Note the dashed line that, if added, would cause (2, 0) to be unconfigured as well. Therefore, this dashed line should not be added to A when unconfiguring (0, 1).

The energy per packet of a route is calculated by summing the energy per packet of each edge, $e_{\mathcal{R}_i,\mathcal{R}_{i+1}}$, in the route.

$$E_{\mathcal{R}} = \sum_{i=0}^{|\mathcal{R}|-2} e_{\mathcal{R}_i,\mathcal{R}_{i+1}}$$

The power consumption of the connection serviced by the route is found by multiplying the energy per packet with the bandwidth of the connection.

$$P_c = E_{\mathcal{R}(c)} \times b_c$$

Additionally, routers have an idle power, P_{idle}, and both routers and TSs have a leakage power, P_{leak}. If a router is unused in a configuration (no routes contain any of the router's ports), it is assumed to be power gated, reducing both its idle and leakage power to approximately zero.

The total power consumption in the interconnect of a platform SoC executing an application is made up of leakage, idle, and communication power.

$$P_{\text{total}} = \sum_{\text{routers}} (P_{\text{leak}} + P_{\text{idle}}) + \sum_{\text{TSs}} P_{\text{leak}} + \sum_{c \in C} P_c$$

The configuration problem can now be formalized: Given BG, NG, and M, synthesize CG with the objective of minimizing P_{total}, subject to

$$\forall c \in C : \mathcal{R}(c) \in R \tag{1}$$

$$\forall \mathcal{R} \in R \forall i < |\mathcal{R}| - 1 : (\mathcal{R}_i, \mathcal{R}_{i+1}) \in A \tag{2}$$

$$\forall p_i, p_j \sum_{\{c \in C | (p_i, p_j) \in \mathcal{R}(c)\}} b_c \leq q_{p_i, p_j} \tag{3}$$

$$\forall \langle \dots p_{i-1} p_i p_{i+1} \dots \rangle \in DG : \nexists i. p_i = p_j \wedge i \neq j. \tag{4}$$

The first requirement states that all connections in the BG have a route, while the second requirement is that for all of these routes, the edges that comprise the route exist in A, that is, the routing and the configuration of the network correspond to each other. Together, these two requirements state that the configured network can service all requests the application will make. The third requirement states that for each link, the sum of the bandwidth requirements of the connections routed on that link does not exceed the link's capacity. The last requirements states that no cycles exist in the DG. As the edges in a DG is the collection of all edges used in all routes, the absence of cycles guarantees freedom from deadlocks. A solution that fulfills all four requirements is said to be valid.

5. OPTIMIZATION ALGORITHMS

In this section, we present our algorithms for solving the configuration problem. The algorithms all work at application design time (as opposed to SoC platform design time) and take different approaches: The *constructive* algorithm starts from an unconfigured CG and configures routes for one connection at a time in a greedy manner, while the *specializing* algorithms start from an already configured CG with deadlock-free routes and makes modifications (*specializations*) to these.

In general, these algorithms work from the realization that the function of routers is to split and merge traffic streams, while simply moving packets along may be done much more efficiently in long, logical links. The objective of the algorithms is to minimize the power consumption, which may lead to some counter-intuitive situations where the algorithms prefer a long, logical link with a nonminimal hop count to a minimal route that passes through a router. Depending on the relative power consumption of links and routers, these longer routes may have lower power consumption than the minimal ones. Due to their greedy nature, the algorithms may find locally optimal solutions instead of the globally optimal solution. Finding the optimal solution with minimal energy consumption would require an exhaustive search.

5.1 Constructive Algorithm

The constructive algorithm is a greedy algorithm that starts from an unconfigured CG. The pseudocode is given in Figure 6. For each connection in the BG, the route with the lowest $E_{\mathcal{R}}$ and with sufficient (residual) capacity is found. As this route may be a

ConstructiveAlgorithm(BG, NG, M)
1: CG=DG=NG; D=$\emptyset$;
2: Sort C in decreasing order according to b
3: **for all** $c \in C$ **do**
4: Find $\mathcal{R}(c)$ in CG
5: **if** (out-degree(src(c)) $>1 \vee$ in-degree(dst(c)) $>1) \wedge \{\mathcal{R}(c)_i | \mathcal{R}(c)_i \in \mathcal{R}(c)\} \cap U = \emptyset$ **then**
6: **if** out-degree(src(c)) $=1$ **then**
7: Configure path to $M(\text{dst}(c))_{in}$ from closest router
8: **else if** in-degree(dst(c)) $=1$ **then**
9: Configure path from $M(\text{src}(c))_{out}$ to closest router
10: **else**
11: Configure a path between o and closest router port, where o is the result of M on the element from $\{\text{src}(c), \text{dst}(c)\}$ with highest total bandwidth over outgoing and incoming connections respectively;
12: Find $\mathcal{R}(c)$ in CG
13: In CG, configure all edges in $\mathcal{R}$ and add these edges to D; Add $\mathcal{R}$ to R
14: **if** DG is cyclic $\vee$ no route found **then**
15: Fail

Fig. 6. The constructive algorithm.

long link directly between IP cores, it is necessary to consider if this is allowed for the given connection, that is, if the source IP core is not the source of any other connection, and the destination IP core is not the destination of any other connection, and if not, to make sure the route passes through a router. This is done in lines 5–12. Finally, the found route is configured, the residual capacity of the links in the route is updated, and the *DG* is tested for possible deadlocks. The constructive algorithm is unable to handle these for now, but in the future we plan to implement either backtracking or rerouting to resolve deadlocks. For now, the algorithm fails to find a solution. Another scenario in which the algorithm fails is if no route is found between the specified endpoints. This may happen from a combination of edges removed from A by configurations in previous iterations and edges excluded because of insufficient residual capacity.

When finding routes in the *CG*, Dijkstra's algorithm is used with the energy per packets as weights on the edges. Edges with insufficient residual capacity are excluded from the search. Doing so finds the route with the lowest $E_{\mathcal{R}}$ and ensures that the capacity of individual links are not exceeded.

As mentioned before, the found route may be an end-to-end circuit switched link directly between the two IP cores, which may not be allowed. In case such a disallowed, direct link is formed, a change in the route to include a router is required. The if-statement in line 5 first tests if either endpoint of c is involved in multiple connections, thus requiring merging or splitting of traffic streams, and then if the route does not include any routers (the route is a end-to-end circuit switched link). If both are true, three cases can occur.

(1) This is the only connection from src(c), but dst(c) is the destination of multiple connections. $M(\text{dst}(c))_{in}$ is connected to the closest router such that the traffic streams are merged as close to the destination IP core as possible.
(2) Multiple connections originate in src(c), but this is the only connection to dst(c). $M(\text{src}(c))_{out}$ is connected to the closest router, such that the traffic streams are split as close to the source IP core as possible.
(3) src(c) is the source of multiple connections and dst(c) the destination of multiple connections. We find the sum of the bandwidth requirement of the connections originating and terminating in src(c) and dst(c) respectively. The task set with the greater sum has its traffic streams split or merged as close to the IP core it is mapped to as possible. Alternatively, the sources and destinations of these

other connections could be taken into consideration by finding a router closer to the middle of the route. Exploring different methods to optimizing the splitting and merging points in this case is left for future work.

After a router has been inserted in the route, a new route is found in line 12 that this time passes through a router. The final steps are to configure the route in CG, add it to the set of routes R, and update the dependency graph, DG. The algorithm actively fails if no route was found or a deadlock has become possible.

Unless the algorithm fails, the generated solutions satisfy the requirement that all connections have routes in Eq. (1) as the loop is over all $c \in C$.

The routability requirement in Eq. (2) is also satisfied. To realize this, first assume that the requirement is violated, that is, $\exists \mathcal{R}_i.(\mathcal{R}_i, \mathcal{R}_{i+1}) \notin A$. For this to occur, either the edge $(\mathcal{R}_i, \mathcal{R}_{i+1})$ was not configured in the first place or it was unconfigured at a later place in the algorithm. As the algorithm never unconfigures an edge, the second case can be readily dismissed. The first case does not occur either, as can be realized by tracing all paths through the body of the for-loop: In all paths, a route is first found and then configured.

The requirement of not exceeding any link capacity in Eq. (3) is satisfied as edges with insufficient residual capacity are excluded when searching for a route. Therefore, it is impossible for an edge's capacity to be exceeded. Finally, the deadlock-free routing requirement in Eq. (4) is satisfied, as the algorithm actively fails in case of a deadlock, that is, no solution is generated in case a deadlock may occur.

The complexity of the constructive algorithm is $O(|C|^2 + |C|(|P| \log |P| + |L|))$, where $|C||P| \log |P|$ dominates. Quicksort is used for sorting the connections according to bandwidth requirements, which requires $O(|C|^2)$ in the worst case, but only $O(|C| \log |C|)$ in the average case. Then, the loop is over all connections, $O(|C|)$, while Dijkstra's algorithm, $O(|P| \log |P|)$, is run for each of these. The found route then needs to be configured, $O(|L|)$, thereby producing the second term.

An example execution of the algorithm on a small problem is shown in Figure 7. The BG is shown in (a). First, a route for the connection (t_0, t_3) is found and configured in (b). As this is the only connection out of t_0 and the only one into t_3, the route in this case is simply a long, logical link connecting the two network interfaces directly. In (c), the connection (t_3, t_2) is being routed. In line 4, an end-to-end circuit switched route is found, but both the out-degree of t_3 and the in-degree of t_2 are greater than 1. In line 11, it is found that t_2 has greater incoming bandwidth than t_3 has outgoing. Therefore, IP2's network interface's input terminal is connected to R2. Then a new route is found, this time going through R2. The final configuration is shown in (d).

We have also implemented a small variation of the algorithm in which paths are initially configured between those IP cores that are the source or destination of multiple connections and their closest router, but no paths are configured between routers. This preprocessing step makes the algorithm generate valid solutions in some cases where it otherwise fails. The variation has no impact on the validity of the generated solutions.

5.2 Specializing Algorithms

The specializing algorithms take a significantly different approach to solving the configuration problem compared to the constructive algorithm. Instead of constructing a solution from scratch, they make modifications (*specializations*) to an existing solution. These specializations are designed to exploit the unique combination of packet and circuit switching in the ReNoC architecture.

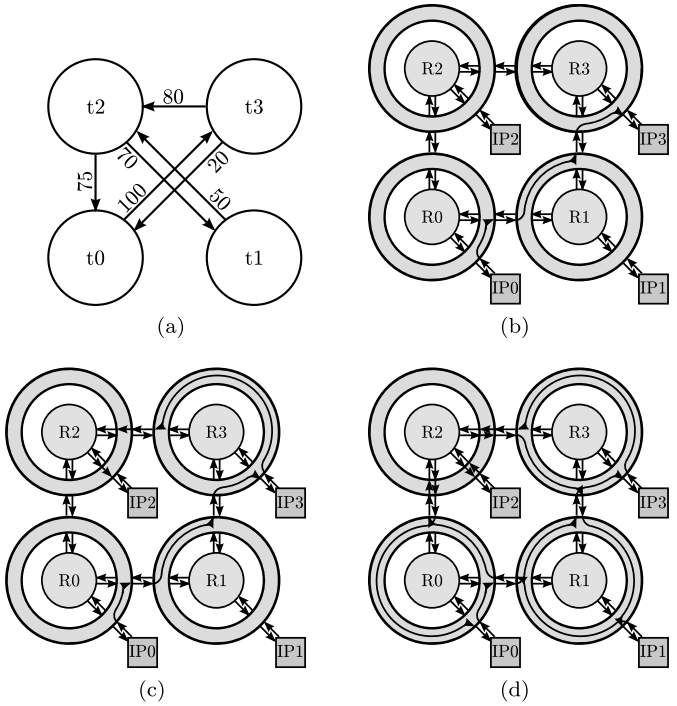

Fig. 7. An example of the constructive algorithm's execution. The BG is shown in (a), intermediate steps in (b) and (c), and the solution in (d).

RouteConnections(BG, NG, M)
1: CG =logical mesh
2: Remove prohibited edges from A to form the routing function
3: Sort C according to bandwidth requirement
4: **for all** $c \in C$ **do**
5: Find $\mathcal{R}(c)$ in CG
6: **if** No route found **then**
7: Fail
8: Add $\mathcal{R}(c)$ to R

Fig. 8. Pseudocode for generating an initial configuration that matches the physical architecture, here a mesh.

5.2.1 Initial Configuration. The starting point of the specializations, the *initial configuration*, can be any valid solution. The idea of the specializations is to make modifications that maintain the validity of the solution. In this article, we use two ways of acquiring the initial configuration: (1) Let the greedy algorithm generate the initial configuration and (2) generate an initial configuration in which the TSs are configured such that the logical topology matches the physical architecture, in our case a 2D mesh. In the second case, the TSs simply constitute an overhead compared to a static mesh.

For the routing in such a mesh, we make use of two different routing functions: dimension ordered XY- and YX-routing and north-, south-, east-, and west-first routing. These routing functions are implemented by removing the prohibited edges from A and L. Note that our earlier comment about not being able to use deadlock-free routing

BypassRouter(CG, NG, R, DG)
1: **for all** $\{(u_i, u_o) \in D | \text{out-degree}(u_i) = 1 \wedge \text{in-degree}(u_o) = 1\}$ **do**
2: Find s_i by going backwards from u_i until input port on TS
3: Find s_o by going forwards from u_o until output port on TS
4: Unconfigure the link going out of s_i
5: Unconfigure the link coming in to s_o
6: Configure the link (s_i, s_o)
7: Update all routes that contain (u_i, u_o)

Fig. 9. Pseudocode for specialization A: Bypass Router.

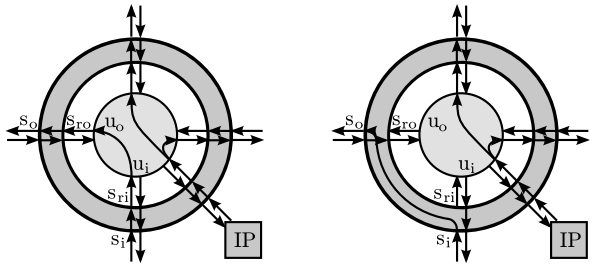

Fig. 10. An example of specialization A: (a) before and (b) after inserting a bypass.

algorithms from literature does not apply here, as we in fact do have a regular topology with bidirectional links in this case. A greedy algorithm as shown in Figure 8 is used to select the routes for all connections. In dimension ordered routing, only one route exists for each connection, whereas in, for example, north-first routing, each connection can select from multiple routes. As in the constructive algorithm, Dijkstra's algorithm is used to find the route with sufficient capacity and lowest energy consumption per packet.

The algorithm in Figure 8 is quite similar to the constructive algorithm with the exception that we are not required to check if an IP core is connected to a router, because of the initial configuration. The complexity analysis is also similar to that of the constructive algorithm. Configuring a logical mesh and removing the prohibited edges requires inspecting each edge in the CG once, $O(|L|)$. Sorting C is $O(|C|^2)$ in the worst case, while the loop is over all elements in C and contains an $O(|P| \log |P|)$ search. The total complexity is $O(|L| + |C|^2 + |C||P| \log |P|)$.

Solutions generated by this algorithm satisfy the requirements that all connections have routes and the link capacities are not exceeded in Eqs. (1) and (3) respectively with identical arguments to those for the constructive algorithm. As all modifications to the CG are done before routing commences, the routability requirement in Eq. (2) is also satisfied, and the chosen routing functions are known from literature to be deadlock-free. With the initial configuration now in place, we can start considering the specializations.

5.2.2 Specialization A: Bypass Router. The pseudocode for the first specialization is given in Figure 9 with an example shown in Figure 10. The specialization consists of detecting cases where all traffic entering a router on one port, $u_i \in U$, exits on one other port, $u_o \in U$, and all traffic exiting through u_o originate in u_i. As DG describes which edges in A are actually used, this can be formalized as $(u_i, u_o) \in D \wedge \text{out-degree}(u_i) = 1 \wedge \text{in-degree}(u_o) = 1$. Whenever this situation occurs, u_i and u_o are not involved in

merging or splitting traffic streams and there is thus no reason for the traffic stream to pass through the router. Therefore, we insert a bypass using the TS.

Referring to the example in Figure 10, consider a subpath, ϕ, in CG that is used by at least one route $\{\mathcal{R} \in R | \phi = \langle s_i s_{ri} u_i u_o s_{ro} s_o \rangle \in \mathcal{R}\}$ as seen in Figure 10(a). By unconfiguring the two edges (s_i, s_{ri}) and (s_{ro}, s_o), the edge (s_i, s_o) is restored in A among others. By configuring (s_i, s_o) and modifying ϕ such that $\phi' = \langle s_i s_o \rangle$ we have bypassed the router as shown in Figure 10(b). Corresponding changes are made to D as well. No further bypasses can be made in the example in Figure 10 because the traffic entering the router on the IP core's port exits the router on two different ports. By applying this specialization across the entire CG, multiple bypasses may be inserted, taking advantage of the TSs' much lower power consumption compared to that of the routers. This specialization's complexity is $O(|L||S'|^2)$, where S' is the set of ports in a single TS. The exponent comes from the fact that all combinations of ports need to be examined to unconfigure an edge safely, as described previously.

When considering the validity of the generated solutions, first recall that we assume a valid solution *before* the specialization is applied. Thus, we only need to prove that the specialization does not violate any of the requirements. As no elements are removed from R, all connections have routes, and the first requirement thus remains satisfied. The routability requirement is also satisfied, because we replace ϕ with ϕ' in all routes where ϕ occurs and make changes to A that correspond to this replacement. In other words, all routes previously using ϕ will use ϕ' after the specialization has been applied and A has been updated to include ϕ' instead of ϕ.

For the capacity requirement, we utilize the fact that the capacity q of all links belonging to TSs are identical: The number of flits per time unit that can be moved through a TS is independent of the path taken through the TS. Thus, as we assume that the capacity of (s_i, s_{ri}) is sufficient before applying the specialization, the capacity of (s_i, s_o) is sufficient after applying the specialization.

Finally, the specialization does not introduce deadlocks (cycles in the DG). Before the specialization was applied, we knew that the out-degree of all vertices in ϕ was one (the very reason we could introduce the bypass). Thus, there existed one and only one path from s_i to s_o, and this path did not form any cycles in the DG. After the specialization has been applied, there is still only one path from s_i to s_o (ϕ' instead of ϕ) which neither forms any cycles in the DG. No cycles can possibly be introduced in the DG by this specialization.

5.2.3 Specialization B: Insert Long Links. This specialization takes a route and modifies it by trying to insert the longest link possible. The pseudocode can be found in Figure 11.

Given $S' \subseteq S$ is the set of ports on one TS, two sets X_{in} and X_{out} of vertices on the route are found, where:

$$X_{in} = \{\mathcal{R}_i | \mathcal{R}_i, \mathcal{R}_{i+1} \in S'\}$$
$$X_{out} = \{\mathcal{R}_i | \mathcal{R}_i, \mathcal{R}_{i-1} \in S'\}.$$

That is, X_{in} is the set of TS input ports in $\mathcal{R}$, while X_{out} is the set of TS output ports in $\mathcal{R}$. By finding all the pairs with one element $x_{in} \in X_{in}$ and the other element $x_{out} \in X_{out}$ where $x_{in} = \mathcal{R}_i, x_{out} = \mathcal{R}_j, j > i$ and sorting them according to their distance $j - i$, we can make an exhaustive search of the possible long links to insert, terminating when we find the longest possible one. We call this sorted set of pairs Y. Finding a long link requires unconfiguring the two edges $(\mathcal{R}_i, \mathcal{R}_{i+1})$ and $(\mathcal{R}_{j-1}, \mathcal{R}_j)$. This naturally impacts all routes utilizing these two edges. The set of these routes is denoted R_{upd}. When considering a pair $y \in Y$ in a route $\mathcal{R}(c)$, if $\exists \mathcal{R}(c_i) \in R_{upd}.b_{c_i} > b_c$, then y is ignored, that is, if inserting a long link for a connection requires another connection with a

InsertLongLinks(BG, CG, NG, M, R)
1: Sort C according to b
2: **for all** $c \in C$ **do**
3: Find X_{in}, X_{out} and Y
4: **for all** $y \in Y$ **do**
5: Unconfigure outgoing edge of x_{in}, incoming edge to x_{out}, and all edges internal to TSs in between that are only used by $\mathcal{R}(c)$; Find R_{upd}
6: Find a path between x_{in} and x_{out}
7: **if** No path found **then**
8: Undo unconfigurations in line 5; Continue with next $y \in Y$
9: Configure path in CG; Update DG; Update $\mathcal{R}(c)$
10: **for all** $\mathcal{R} \in R_{upd}$ **do**
11: Find new route for $\mathcal{R}$
12: **if** No route found **then**
13: Undo changes to CG, DG and all $\mathcal{R}$; Continue with next $y \in Y$
14: Configure path in CG; Update DG; Update $\mathcal{R}$
15: **if** DG is cyclic **then**
16: Undo all changes in this iteration

Fig. 11. Pseudocode for inserting long links using specialization B.

higher bandwidth requirement to be rerouted, the algorithm proceeds to the next y. If all the routes in R_{upd} belong to connections with a lower bandwidth requirement, these connections are rerouted at the end of each iteration.

This specialization has a higher complexity than the other algorithms considered so far. The for-loops in lines 2 and 4 contribute with $O(|C||P|^2)$. The unconfiguring in line 5 potentially considers each edge in the CG and is $O(|L|)$, while the for-loop in line 10 is $O(|C|)$. The body of this loop is $O(|P| \log |P|)$. In total, the complexity is $O(|C||P|^2(|L| + |C||P| \log |P|))$.

The requirement that all connections have routes is satisfied, because we never remove a route from R. The routability requirement is also satisfied, as whenever we make changes to the CG, the affected routes are updated correspondingly. As in the other algorithms, edges with too little residual capacity are omitted from searches in the graph, thus the capacity requirement is also satisfied. Finally, the requirement of no deadlocks is satisfied by the final if-statement that reverts the changes to the solution in case a deadlock is possible.

An example of the execution of this algorithm is shown in Figure 12, where the route between IP0 and IP1 shares two router ports with the dashed route in (a), thus specialization A is unable to make any modifications in this situation. By unconfiguring the edges out of x_{in} and in to x_{out}, searching for a new route in line 6 yields a circuit switched route directly between the two IP cores in (b). No other routes are affected by this specialization.

6. EXPERIMENTAL SETUP

This section describes the physical architectures and the applications used along with the experiments we conduct.

As we focus on the power savings by using ReNoC to move traffic out of routers and onto long, logical links, it would be an unfair comparison to use a large router with many features and high power consumption. Therefore we use routers similar to those in Stensgaard and Sparsø [2008]: low-power, single-cycle, source routed, wormhole routers at 100 MHz with two virtual channel buffers in each input port, each buffer being four flits deep. A packet consists of four flits of which one is a header flit and the remaining three are payload.

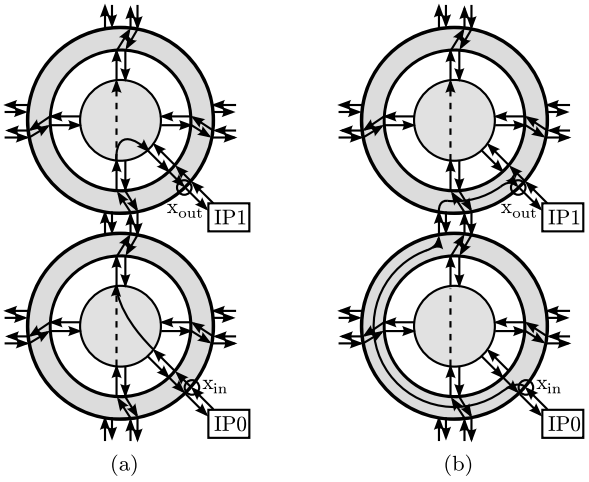

Fig. 12. An example of specialization B. (a) shows two connections before the specialization is applied, while (b) shows the two connections after the specialization has been applied.

For comparison purposes, we consider three different physical architectures, all based on a 2D mesh topology. Again, the ReNoC architecture is in no way limited to meshes.

— A conventional mesh of routers without TSs gives us a baseline to evaluate the overhead of the TSs. This physical architecture will be referred to as a static mesh. For routing in this mesh, we use both dimension ordered (XY and YX) and north-, south-, east-, and west-first routing.
— The first ReNoC-based physical architecture is, as earlier, a standard mesh of network nodes, where network nodes here consist of a router, a TS, and an IP core. This physical architecture will be referred to as the Single-Link architecture (SL). An example can be seen in Figure 3.
— The second ReNoC-based physical architecture is also a mesh of network nodes as before, but with the difference that the number of NoC links is doubled while the routers' sizes are as in the other two physical architectures. This means that there are twice as many links as router ports, which, for example, allows a long, logical link to skip unaffected through an area that is otherwise congested. This physical architecture will be referred to as the Double-Link architecture (DL). This is the architecture shown in Figure 1.

Table I presents the energy consumption per packet in the different components of a ReNoC-based NoC. Routers and topology switches have been synthesized and their power consumption determined using commercial synthesis and power characterization tools using estimated wire-load models, while link characterization is based on figures from SPICE simulations [Banerjee et al. 2007; Stensgaard and Sparsø 2008]. All figures in the table are prelayout, based on low-leakage cells from a commercial 90-nm standard cell library, using a 1V supply voltage at nominal parameters. The energy per packet is the average energy consumed when sending a packet based on random data, leakage is the leakage power consumption, and idle power is the dynamic power that is always consumed, independent of the use. Idle power accounts for clocking of clock-gates and registers that are not clock-gated. The TSs have previously been shown to add around 10% to the area of the interconnect [Stensgaard and Sparsø 2008].

Table I. Energy and Power Consumption of the Components in a ReNoC-Based NoC

Module	Energy/packet (pJ)	Leakage (μW)	Idle power (μW)
Link, 1mm	21	-	-
5x5 Router	32	8.6	136
TS SL	0.48/1.05	0.55	1.44
TS DL	0.9/1.4	2.65	1.61
4x4 Router	31	6.7	109
TS SL	0.4/0.87	0.43	1.44
TS DL	0.71/1.2	1.64	1.44
3x3 Router	30	4.7	82
TS SL	0.41/0.43	0.22	1.44
TS DL	0.72/1.05	0.55	1.44

The two energy/packet values are to a router input and to a NoC link output respectively.

Table II. Characteristics of the Applications

App. name	$\|T\|$	$\|C\|$	Phys. arch.
VOPD	12	14	3×4
R12	12	11	3×4
C12	12	12	3×4
MMS	16	30	4×4
R16	16	14	4×4
C16	16	16	4×4
S64	64	83	8×8
R64	64	62	8×8
C64	64	64	8×8

$|T|$ is the number of task sets (IP cores), and $|C|$ is the number of connections between these.

We use a mixture of real and synthetic applications to evaluate our algorithms and physical architectures. The following applications can be found in literature.

— VOPD: A multimedia application [Murali et al. 2005].
— MMS: A multimedia system [Hu and Marculescu 2005].
— Rotate and complement: Well-known traffic patterns from computer networks [Dally and Towles 2003] that have also been suggested for use in NoC micro benchmarks [Salminen et al. 2008].

We also use a larger synthetic application (S64) that incorporates some of the patterns suggested in Salminen et al. [2008]. Characteristics of the different applications can be found in Table II.

The mapping M has been generated by hand for the VOPD, MMS, and S64 applications. For the remaining applications, the mapping is determined from the addresses of task sets ($t \in T$) that are an integral part of forming the traffic patterns. The IP core with address zero has been put in the lower-left corner of the mesh, and addresses increment by one along the x-axis.

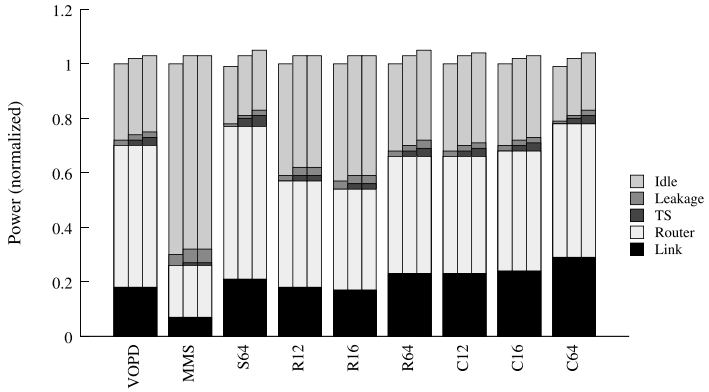

Fig. 13. ReNoC power overhead. The bars represent the power consumption normalized to the static architecture. The order of the bars is: Static, SL, and DL.

The experiments we conduct are (for all applications on SL and DL) as follows.

— Configure a logical mesh and route using the algorithm in Figure 8. This allows us to evaluate the overhead of the TSs.
— Use the constructive algorithm (Figure 6) to configure each application on each platform.
— Generate an initial configuration using the algorithm in Figure 8 and apply each of specialization A, specialization B, specialization A on the output of specialization B (BA), and vice versa (AB).
— Generate an initial configuration using the constructive algorithm and apply both specializations and both combinations of specializations to this solution.

In the static architecture, we evaluate the power consumption using both the six mentioned routing functions and also using the constructive algorithm to make an application-specific routing algorithm. For the results section, from those experiments where multiple routing functions are used, we select the best of the results, as we are not specifically interested in whether, for example, XY routing is better than YX routing.

7. RESULTS

In this section, we present and discuss our results. The power consumption in "links" shown in the graphs in this section relates to "NoC links," that is, links between network nodes.

In Figure 13, we see the overhead of the ReNoC architecture when the logical topology forms a mesh. The power consumption of each application has been normalized to that of the static architecture. As can be seen, the TSs only add between 2–5% to the power consumption. This is the worst-case scenario as the TSs' ability to move traffic out of routers is not utilized.

Figure 14 shows a comparison of the different physical architectures. For each combination of physical architecture and application, we selected the best solution from all the experiments in order to show the potential of the ReNoC architecture. The ReNoC architecture clearly leads to lower power consumption with the decrease primarily found in routers and secondarily in reduced idle and leakage power from clock- and power-gating. For SL, the reduction in power consumption averages 36%

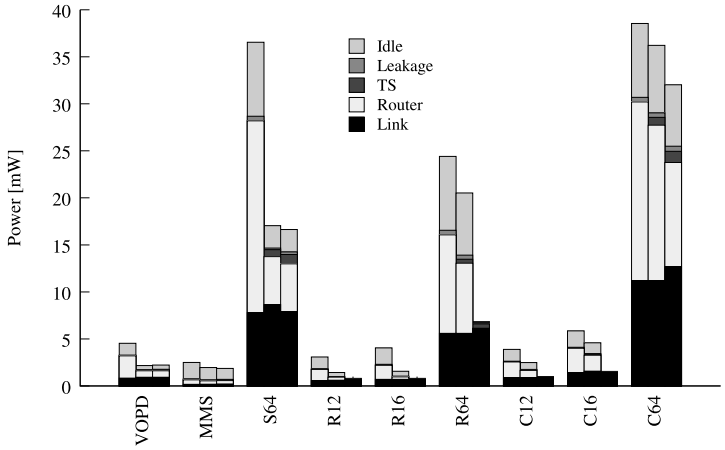

Fig. 14. Comparison of the physical architectures. The order of the bars is: Static, SL, DL.

Table III. The Number of Powered-On Routers and Total Power Consumption in the Best Configuration of All Physical Architectures and Applications

		Routers on		
App	No. routers	Static	SL	DL
VOPD	12	12	4	4
R12	12	12	4	1
C12	12	12	6	0
MMS	16	16	11	10
R16	16	16	4	0
C16	16	16	10	0
S64	64	64	19	19
R64	64	64	52	0
C64	64	64	56	51

with a minimum of 6% for C64 and a maximum of 61% for R16. DL has lower power consumption than SL due to the greater possibilities of moving traffic out of routers and on to long, logical links, which is seen from the even lower power consumption in routers. On average, DL reduces power consumption by 58% compared to the static architecture, varying between 17% for C64 and 80% for R16. For some applications (R16, R64, C12, C16) the traffic pattern can even be implemented fully using only circuit switching in DL. This clearly demonstrates the potential of reducing power consumption using the ReNoC architecture.

Table III shows the number of routers that are powered on in the same solutions whose power consumption is shown in Figure 14. The main trend to notice is that more routers can be powered off when going from SL to DL indicating that more traffic is moved out of routers and onto long links, thereby reducing the overall power consumption. The exceptions to this trend are VOPD and S64 where the same number of routers are powered on in SL and DL. For VOPD it simply does not pay off to use less than four routers. Doing so would require significant detours of the traffic streams to be split or merged, easily outweighing the gains of powering off another router. The

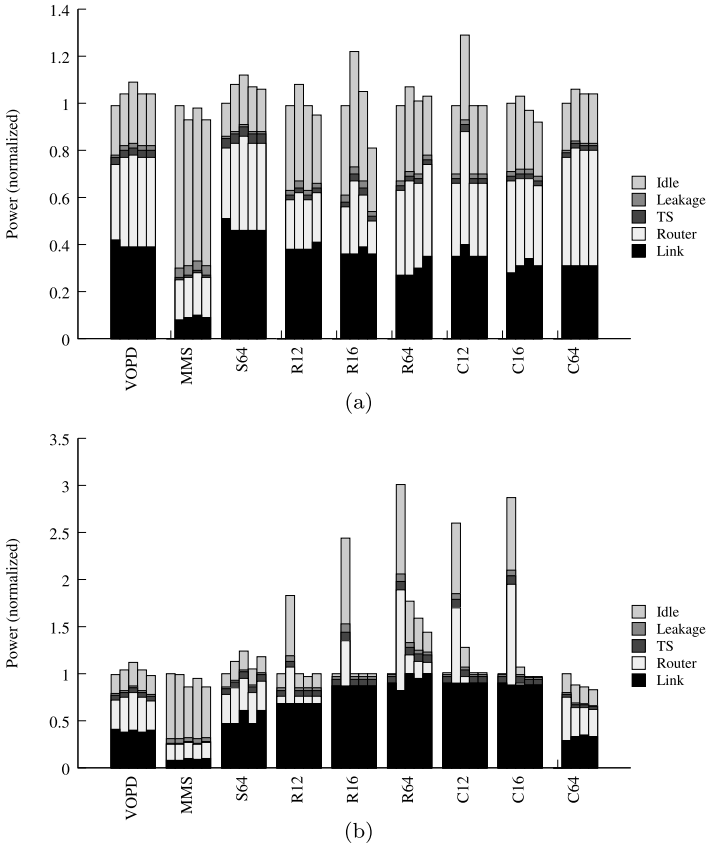

Fig. 15. Comparison of the optimization algorithms for SL (a) and DL (b). The order of columns are constructive algorithm, specialization A, specialization B, AB, and BA. Bars are normalized to the first nonzero bar for each application.

power consumption of VOPD is slightly higher on DL because the configurations on SL and DL are identical, but DL has larger, more power-consuming TSs.

In S64 when going from the static architecture to SL in Figure 14, it can be seen that both overall and router power consumption decrease while link power increases. This is caused by the algorithms sending some traffic streams out on minor circuit switched detours in order to avoid going through some routers where the streams in question are neither split nor merged. In Figure 14 the router power remains the same in DL, but the link power decreases because shorter circuit switched paths have replaced some longer ones. This demonstrates the potential of DL to move traffic that simply passes through a busy region of the network onto a long link that passes right by the busy routers.

Figures 15(a) and 15(b) give a comparison of the algorithms on SL and DL, respectively. The bars are normalized to the first nonzero bar for each application. For SL, the constructive algorithm only produces a valid solution for two of the applications. However, for those two it produces the best solution of all the algorithms. In general the algorithms' performance is almost uniform with most solutions within 12% of the reference. The outlier is specialization B that produces the worst solutions by up to 30%, which is interesting since BA often produces the best solutions. This can be

explained by B maximally inserting one long link per connection: When a change has been made for a connection, the algorithm proceeds to the next connection. If instead, the algorithm would continue trying to insert long links for the current connection, specialization B is expected to perform at least as well as BA. Investigating this is left for future work.

For DL, the constructive algorithm produces valid solutions for eight of the applications. In all but one case, these solutions have the lowest power consumption. This is explained by the greater freedom in generating solutions when starting from an unconfigured *CG* compared to starting from an already valid solution and making modifications. Considering the specializations, A on its own generally produces much worse solutions than the other specializations by up to 200%. This is explained by A not being able to exploit the extra links, as it operates on subpaths that are internal in network nodes. As for SL, BA produces the best solutions of the specializations, although the distribution again is rather uniform.

Considering the execution time of the different algorithms AB and BA are the slowest, but even for the large problems the solution is found in less than a minute.

Using the specializations to optimize the solutions produced by the variation of the constructive algorithm where IP cores are connected to the closest router in a preprocessing step gives some improvement in a few cases. However, in no case are these solutions better than the best one produced either by the constructive algorithm without the preprocessing step and without subsequent specialization or by the specializations applied to the initial configuration where a logical mesh is formed. Furthermore, the specializations are unable to improve on the solutions generated by the constructive algorithm without the preprocessing step for our benchmarks.

For VOPD on both SL and DL, specialization B improves the solutions generated by the constructive algorithm with the preprocessing step by 14% and 19%, respectively. For S64, improvements of around 4% are also seen on both SL and DL using specialization B. In both benchmarks, the improvements come from traffic being moved out of routers and onto long links, leading to lower power consumption in the routers and in some cases also unused routers that are powered down. For the remaining benchmarks, no improvements are made by the specializations on the solutions from the constructive algorithm. As mentioned before, for the applications considered here, applying the specializations to the output of the constructive algorithm never produces a better solution than that produced by the constructive algorithm itself or by the applying the specializations to the initial configuration where the logical topology matches the physical architecture. However, it cannot be ruled out that some applications exist for which applying the specializations to the output of the greedy algorithm produces the best results.

We also tried using the constructive algorithm to generate application-specific, deadlock-free routing algorithms in the static mesh. However, for our benchmark applications, only insignificant differences (less than 0.2%) are seen between the well-known routing algorithms and application-specific routing. In order for application-specific routing to be beneficial, the other routing algorithms need to break down from insufficient capacity along their restricted sets of allowed routes.

The results presented in this section have focused on ReNoC's advantage in power consumption over a conventional NoC, and on the relative performance of the algorithms for automatically solving the configuration problem. However, one of ReNoC's key features is difficult to put into numbers, namely its flexibility. By providing a reconfigurable interconnect, we can on a single chip switch between the best found configuration for the different applications. This is one of ReNoC's strengths: The interconnect in a *single* SoC platform chip can be used to provide energy efficient, intra-chip communication for a wide range of applications.

8. DISCUSSION AND FUTURE WORK

Up to now, we have focused on initialization-time configuration. In the following, we discuss some initial thoughts on runtime reconfiguration, and show how ReNoC can be used to overcome some of the issues that arise when using a regular NoC.

Let us consider the general case of a system with a heterogeneous set of IP cores. A process executing on this platform is assigned a set of resources (IP cores and network connections), and the communication between these can be described by a graph similar to the *BG* in the previous section. A process may require *exclusive* or *shared* access to IP cores. An exclusively owned IP core is allocated to one and only one process until the process releases the IP core or terminates. A shared IP core can be allocated to multiple processes, in which case some local scheduling is performed in the IP core. Furthermore, processes may request guaranteed service of the NoC (if the routers implement such a feature) or exclusive access to end-to-end connections in the NoC. Providing GS in NoCs is a major topic which has been covered in the literature and will not be discussed further here [Bjerregaard and Sparsø 2005b; Millberg et al. 2004; Rijpkema et al. 2001].

In such a system, an operating system executing on one of the IP cores will handle resource allocation requests and scheduling of processes on shared IP cores. The scheduling of processes in a multiprocessor system is a well-known subject, and will not be discussed further here. Considering the allocation of IP cores, two situations can occur: (i) A process requests an IP core of a given type, or (ii) a process releases an IP core. In the first situation, the OS needs to find a suitable instance of the specified IP core type and connect it to those parts of the requesting process that it needs to communicate with, all without interfering with other processes' exclusively owned connections. It may be acceptable to interrupt communication on shared connections, although doing so will have to be done in a safe manner [Hansson and Goossens 2007]. The second case is somewhat simpler to handle, as the OS here simply needs to tear down the no longer used connections, although the safety requirements of course still need to be satisfied.

During the execution of such a system, it may obviously not be possible to assign processes to IP cores in such a way that the IP cores assigned to the different processes form disjoint subnetworks. Therefore, in a traditional packet switched NoC, traffic relating to different processes will interfere, and this may compromise real-time properties. For virtualization purposes (running a virtual machine in a process), the lack of traffic separation may even be a security concern [Duato 2008].

In a conventional NoC, the only possibility for connecting distant IP cores is to establish a packet switched connection between them. While this is no problem in itself, when coupled with the requirement of separation of traffic, significant detours may be necessary, if it even is possible to find such detours. With a reconfigurable interconnect such as ReNoC, it is possible to take advantage of the circuit switched topology switches to overcome these issues, either by establishing end-to-end connections or by establishing a long link which passes through another processes' exclusively owned region of IP cores. Exploring these issues in online mapping and allocation in the context of ReNoC is left for future work.

We will also be investigating other specializations, including one where we try to take advantage of the multitude of possible routes in, for example, north-first routing: By making a route follow a detour, it may be possible to collect the NoC traffic in fewer routers, allowing more routers to be powered down. Concerning the existing algorithms, modifications to the constructive algorithm to better handle deadlocks will also be considered. This could, for example, be done by backtracking once the dependency graph becomes cyclic and finding alternative routes.

In this article, we made a comparison between a generic NoC (the static architecture) and ReNoC-based physical architectures. It will be very interesting to compare with the other end of the spectrum: application-specific topologies generated using state-of-the-art methods. Although these methods do not produce flexible NoCs (as ReNoC does), it will be highly interesting to determine the cost (if any) of the added flexibility of ReNoC.

9. CONCLUSION

In this article, we have presented the ReNoC architecture, a generic network-on-chip architecture which combines packet switching, physical circuit switching, and (initialization time) reconfigurability. The key idea is to augment the routers in a conventional packet switched NoC with a layer of so-called topology switches, which are very similar to the switch-boxes used in FPGAs. In this way, it is possible to implement a generic MPSoC platform with a particular NoC topology, and to configure different logical NoC topologies on this platform, thereby efficiently supporting different applications. ReNoC is a generic architectural idea which can be combined with any NoC topology and NoC router. The article presented two example architectures both using a 2D mesh NoC: one in which the number of links matches the number of router ports and one in which the number of links has been doubled. These platforms are called SL (Single Link) and DL (Double Link) respectively, and the article compares the performance of these against a conventional 2D mesh topology without topology switches (denoted "static").

The configuration of a ReNoC platform for a given application involves solving the following problems: topology synthesis, topology mapping, and routing that together constitute the configuration problem. We have presented three algorithms that solve these problems concurrently with the objective of minimizing energy consumption in the NoC. The algorithms are applied to two real and seven synthetic benchmark applications, and they are all configured onto the three physical architectures mentioned earlier. In this way, the experiments evaluate both the algorithms and the ReNoC architecture.

No algorithm can be said to be the best in general, because their relative performance depends on both the application and the physical architecture. In general, the constructive algorithm produces the best results, when it does not encounter a deadlock. In case of deadlocks, the specializations can be used to find a guaranteed nondeadlocking solution. The combination of specializations B and A consistently produces good results, but not always the best ones, across the applications and physical architectures.

The SL architecture produces solutions with 36 % lower power consumption than what is possible in the conventional, static architecture, while the DL architecture reduces the power consumption by 58 % compared to the static architecture.

REFERENCES

AL FARUQUE, M. A., EBI, T., AND HENKEL, J. 2007. Run-Time adaptive on-chip communication scheme. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'07)*. IEEE Press, 26–31.

ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., AND YELICK, K. A. 2006. The landscape of parallel computing research: A view from Berkeley. Tech. rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

ASCIA, G., CATANIA, V., AND PALESI, M. 2004. Multi-Objective mapping for mesh-based NoC architectures. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 182–187.

BANERJEE, A., MULLINS, R., AND MOORE, S. 2007. A power and energy exploration of network-on-chip architectures. In *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS'07)*. IEEE Computer Society, Los Alamitos, CA, 163–172.

BENINI, L. AND DE MICHELI, G. 2002. Networks on chips: A new SoC paradigm. *Comput. 35,* 1, 70–78.

BJERREGAARD, T. AND SPARSØ, J. 2005a. 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)*. IEEE Computer Society, 1226–1231.

BJERREGAARD, T. AND SPARSØ, J. 2005b. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society, Los Alamitos, CA, 34–43.

CARLONI, L. P. AND SANGIOVANNI-VINCENTELLI, A. L. 2002. Coping with latency in soc design. *IEEE Micro 22,* 5, 24–35.

CARMONA, J., CORTADELLA, J., KISHINEVSKY, M., AND TAUBIN, A. 2009. Elastic circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 28,* 10, 1437–1455.

CHAN, J. AND PARAMESWARAN, S. 2008. Nocout: Noc topology generation with mixed packet-switched and point-to-point networks. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '08)*. IEEE Computer Society Press, Los Alamitos, CA, 265–270.

DALL'OSSO, M., BICCARI, G., GIOVANNINI, L., BERTOZZI, D., AND BENINI, L. 2003. Xpipes: A latency insensitive parameterized network-on-chip architecture for multiprocessor socs. In *Proceedings of the 21st International Conference on Computer Design*. 536–539.

DALLY, B. 2007. Enabling technology for on-chip interconnection networks. In *Proceedings of the 1st International Symposium on Networks-on-Chip*. 3–3.

DALLY, W. J. AND TOWLES, B. 2001. Route packets, not wires: On-Chip interconnection networks. In *Proceedings of Design Automation Conference*. IEEE Computer Society, Los Alamitos, CA, 684–689.

DALLY, W. J. AND TOWLES, B. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA.

DUATO, J. 1993. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Paral. Distrib. Syst. 4,* 12, 1320–1331.

DUATO, J. 2008. Managing heterogeneity in future NoCs. In *Proceedings of the 1st International Workshop on Network on Chip Architectures*. 2–4.

HAMMOND, L., NAYFEH, B. A., AND OLUKOTUN, K. 1997. A single-chip multiprocessor. *Comput. 30,* 9, 79–85.

HANSSON, A. AND GOOSSENS, K. 2007. Trade-Offs in the configuration of a network on chip for multiple use-cases. In *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS'07)*. IEEE Computer Society, Los Alamitos, CA, 233–242.

HANSSON, A., GOOSSENS, K., AND RĂDULESCU, 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.

HANSSON, A., COENEN, M., AND GOOSSENS, K. 2007. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*. 954–959.

HU, J. AND MARCULESCU, R. 2005. Energy- and performance-aware mapping for regular noc architectures. *Comput.-Aid. Des. Integr. Circ. Syst. 24,* 4, 551–562.

LEE, S.-J., LEE, K., AND YOO, H.-J. 2005. Analysis and implementation of practical, cost-effective networks on chips. *IEEE Des. Test Comput. 22,* 5, 422–433.

MAGARSHACK, P. AND PAULIN, P. G. 2003. System-on-chip beyond the nanometer wall. In *Proceedings of the 40th Annual Design Automation Conference (DAC'03)*. ACM, New York, 419–424.

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)*. IEEE Computer Society, Los Alamitos, CA, 20890.

MODARRESSI, M., SARBAZI-AZAD, H., AND TAVAKKOL, A. 2009. Performance and power efficient on-chip communication using adaptive virtual point-to-point connections. In *Proceedings of the International Symposium on Networks-on-Chip.* 203–212.

MURALI, S., BENINI, L., AND DE MICHELI, G. 2005. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *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.

OGRAS, U. AND MARCULESCU, R. 2006. "It's a small world after all": Noc performance optimization via long-range link insertion. *IEEE Trans. VLSI Syst. 14,* 7, 693–706.

OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. 1996. The case for a single-chip multiprocessor. *SIGPLAN Not. 31,* 9, 2–11.

PALESI, M., HOLSMARK, R., KUMAR, S., AND CATANIA, V. 2006. A methodology for design of application specific deadlock-free routing algorithms for noc systems. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'06)*. ACM, New York, 142–147.

RIJPKEMA, E., GOOSSENS, K., AND WIELAGE, P. 2001. A router architecture for networks on silicon. In *Proceedings of the 2nd Workshop on Embedded Systems (Progress'01)*.

SALMINEN, E., GRECU, C., HÄMÄLÄINEN, T. D., AND IVANOV, A. 2008. Network-on-Chip benchmark specification part 1: Application modelling and hardware description version 1.0. Tech. rep., OCP (http://www.ocpip.org).

SPARSØ, J. AND FURBER, S., Eds. 2001. *Principles of Asynchronous Circuit Design – A Systems Perspective*. Kluwer Academic Publishers.

STAROBINSKI, D., KARPOVSKY, M., AND ZAKREVSKI, L. 2003. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Netw. 11,* 3, 411–421.

STENSGAARD, M. B. AND SPARSØ, J. 2008. Renoc: A network-on-chip architecture with reconfigurable topology. In *Proceedings of the 2nd ACM/IEEE International Symposium on Networks-on-Chip (NOCS'08)*. IEEE Computer Society, Los Alamitos, CA, 55–64.

STUART, M. B. AND SPARSØ, J. 2007. Custom topology generation for network-on-chip. In *Proceedings of the Norchip Conference*. 1–4.

STUART, M. B., STENSGAARD, M. B., AND SPARSØ, J. 2009. Synthesis of topology configurations and deadlock free routing algorithms for renoc-based systems-on-chip. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'09)*. ACM, New York, 481–490.

WOLKOTTE, P. T., SMIT, G. J. M., RAUWERDA, G. K., AND SMIT, L. T. 2005. An energy-efficient reconfigurable circuit switched network-on-chip. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - 12th Reconfigurable Architecture Workshop (RAW'05)*. IEEE Computer Society, 155.

Received June 2009; revised December 2009; accepted January 2010