Downloaded from orbit.dtu.dk on: Oct 13, 2024

DTU Library

Modelling, Synthesis, and Configuration of Networks-on-Chips

Stuart, Matthias Bo

Publication date:
2010

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

Link back to DTU Orbit

Citation (APA):
Stuart, M. B. (2010). *Modelling, Synthesis, and Configuration of Networks-on-Chips*. Technical University of Denmark. IMM-PHD-2010-230

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Modelling, Synthesis, and Configuration of Networks-on-Chips

Matthias Bo Stuart

Kongens Lyngby 2010
IMM-PHD-2010-230

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

This thesis presents three contributions in two different areas of network-on-chip and system-on-chip research: Application modelling and identifying and solving different optimization problems related to two specific network-on-chip architectures. The contribution related to application modelling is an analytical method for deriving the worst-case traffic pattern caused by an application and the cache-coherence protocol in a cache-coherent shared-memory system. The contributions related to network-on-chip optimization problems consist of two parts: The development and evaluation of six heuristics for solving the network synthesis problem in the MANGO network-on-chip, and the identification and formalization of the ReNoC configuration problem together with three heuristics for solving it.

Resumé

Denne afhandling præsenterer tre bidrag til to forskningsområder inden for network-on-chip og system-on-chip områderne: Modellering af applikationer og identifikation og løsning af forskellige optimeringsproblemer vedrørende to forskellige network-on-chip arkitekturer. Bidraget til applikationsmodellering er en analytisk metode til at udlede worst-case trafikmønsteret forårsaget af en applikation og cachekohærens protokollen i et cachekohærent shared-memory system. Bidragene til network-on-chip optimeringsproblemer består af to dele: Udvikling og evaluering af seks heuristikker til at løse netværkssynteseproblemet i MANGO arkitekturen og identifikation og formalisering af ReNoC konfigureringsproblemet samt tre heuristikker til at løse det.

Preface

This thesis was prepared at DTU Informatics, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

The work has been supervised by Professor Jens Sparsø and co-supervised by Associate Professor Alberto Nannarelli.

Kgs. Lyngby, Februar 2010

Matthias Bo Stuart

Acknowledgements

Many people have helped me arriving at this point. I am grateful to you all.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Over the past decades, scaling of integrated circuit manufacturing technologies has allowed ever-increasing complex circuits and systems to be built. Thus, the amount of area occupied by a single microcontroller in the past is currently and in the future occupied by multi- and many-core systems featuring multiple microprocessors and on-chip peripherals. Such a whole system integrated on a single chip is commonly called a System-on-Chip (SoC). One requirement for such systems to be feasible is the existence of a scalable interconnect. Almost a decade ago, network-on-chip (NoC) was suggested for this purpose [5, 25]. Since then, it has evolved into a research area of its own with many different implementations of networks, models, algorithms for solving optimization problems, etc.

The next section gives a brief introduction to networks-on-chips. The reader is assumed to be mostly familiar with the general topic. If not, several textbooks on the area of interconnection networks [23, 28] can be recommended together with a survey of research and practices of network-on-chip [8]. Section 1.2 presents some of the research challenges in network-on-chip, while section 1.3 outlines the contributions of the work presented in this thesis to the research area of network-on-chip. Section 1.4 gives an overview of the thesis' structure.

1.1 Network-on-Chip

As mentioned above, increasingly complex systems can be built on a single chip. However, this complexity comes with a price in increased design effort, time-to-

market, and production costs (particularly non-recurring engineering costs) of these systems. These factors have led to the prediction that application-specific integrated circuits (ASICs) are becoming an infeasible way of implementation for many applications [45].

One way of handling these challenges is to improve the reusability of designs by providing processors, memories, on-chip peripherals, etc. with standardized interfaces to the interconnect. This allows system designers to use off-the-shelves components from multiple independent vendors to construct their systems. For this reason, the processors, memories, peripherals, etc. are commonly denoted intellectual property (IP) cores.

Such IP-based systems require an interconnect with similarly standardized interfaces. Although busses can accommodate these interfaces, they do not scale well with the increasing number of IP cores: As every transaction on a bus is broadcast to every device attached to the bus, each IP core's share of bandwidth is inversely proportional to the number of IP cores connected to the bus. Furthermore, connecting many IP cores to a single bus increases the bus' capacitance, leading to higher power consumption for driving the bus. Point-to-point links scale poorly to larger systems and also complicate the reuse of IP cores, as a given core may need two links in one system, but only one link in another system.

NoCs provide a scalable interconnect without any of these issues at an increased cost in the complexity of the interconnect. However, it is the only known generic type of interconnect that scales well with the number of IP cores, which makes it highly interesting for on-chip, system-level interconnects.

NoCs can be used as the interconnect in both ASICs and in generic platforms on which many different applications may execute. For ASICs, the NoC can be adapted to the specific application, providing a highly efficient interconnect, while for generic platforms, a similarly generic NoC can be used to provide a flexible interconnect, servicing multiple different applications' communication requirements. The decision to use either a customized or a generic implementation is a trade-off of cost versus efficiency: ASICs provide high efficiency at high cost, while platform chips provide low efficiency (compared to ASICs) at low cost. The ReNoC architecture [71] – which is considered in chapter 7 – aims at providing an interconnect with high efficiency at low cost by allowing the interconnect in a generic platform to be adapted to a specific application's requirements at run-time of the application.

The design and implementation of links, router, and network interface architectures is one aspect of NoCs. Another aspect is the tools and design methodologies that are required to make the use of NoCs feasible. Some of the individual

tools and algorithms are described in the related work in chapter 4, while some of the proposed overall design flows are discussed here.

Many researchers have suggested design flows for application-specific NoCs [14, 30, 55]. While these design flows have been developed for different NoCs, they share certain characteristics that are common to most design flows: The flow takes a number of inputs including a model of the application, power, area, and performance models of the NoC components, and different constraints that the output must satisfy. These inputs are given to one or more tools that perform various tasks such as mapping the application to IP cores, mapping IP cores to network interfaces, synthesizing a network topology, finding deadlock-free routes in this topology, performing floorplanning of the synthesized system, and solving optimization problems that are specific to the considered NoC. The output of each individual tool is evaluated against the given constraints, which may trigger the restart of one of the earlier steps in the flow, thereby causing multiple iterations of the design to be made. Once an acceptable solution is found, the design flow provides synthesis, simulation, and verification models as needed. Use of these design flows is claimed to produce high-quality solutions in a matter of hours instead of weeks [55].

The research presented in this thesis is in the context of two specific NoC architectures: Message passing, Asynchronous Network-on-chip providing Guaranteed services over OCP interfaces (MANGO) [7] and the Reconfigurable NoC (ReNoC) architecture [71]. MANGO is an asynchronous or clockless network-on-chip architecture that provides absolute guarantees on the worst-case latency experienced by individual flits. ReNoC is an architectural extension rather than a specific NoC implementation and allows the network topology in which routers are connected to be configured after the NoC has been manufactured. More details about these NoCs are provided in chapters 6 and 7 respectively.

1.2 Research Challenges

This section provides a brief overview of some of the research challenges related to NoCs in connection to the work presented in this thesis. A detailed overview of NoC research problems can be found in [50].

Communication characterization refers to the problem of determining the traffic pattern between IP cores a given application is expected to cause. Depending on the available information about the application, the difficulty of determining the traffic pattern varies. The overall flow of the

application may be extracted from various sources, such as design documents or through simulation or instrumentation [77, 42]. In chapter 5, research into analytically determining the traffic pattern in a cache-coherent, shared-memory system from an abstract model of the application is presented.

Application mapping refers both to the problem of mapping applications to IP cores and to that of mapping IP cores to network interfaces (in [50] the former is referred to as application scheduling). Multiple approaches to solving these problems are discussed in the related work. Although these problems are not considered in this thesis, they are included for the sake of completeness.

NoC synthesis refers to the synthesis of either application-specific or generic, irregular or conventional network topologies. Several methods for solving this problem are discussed in the related work. Chapter 6 presents research into synthesizing application-specific topologies in the MANGO network-on-chip, while topology synthesis is also part of the ReNoC configuration problem, which is the topic of chapter 7.

Routing concerns the paths packets take through the network. The challenges in routing include avoiding congestion, minimizing energy consumption, and avoiding (or recovering from) deadlocks. Finding deadlock-free routing algorithms for irregular, application-specific network topologies is also part of the ReNoC configuration problem presented in chapter 7.

Topology mapping is a problem specific to the ReNoC architecture. As described above, ReNoC aims at providing highly efficient, low cost interconnects in generic platform chips. The topology mapping problem consists of mapping an application-specific network topology onto a generic network topology in a power-efficient manner. Topology mapping is also part of the ReNoC configuration problem presented in chapter 7.

1.3 Contributions

This thesis presents three contributions: (1) Communication characterization in cache-coherent shared-memory systems, (2) development and evaluation of heuristics for synthesizing application-specific topologies in MANGO, and (3) identification and formalization of the ReNoC configuration problem and development and evaluation of heuristics for solving this problem.

The first contribution (chapter 5) constitutes research into analytically deriving the worst-case traffic pattern in a cache-coherent shared-memory system from an abstract model (known as a task graph) of an application. Commonly, a simple one-to-one mapping of communication edges in task graphs to communication in traffic patterns has been assumed, corresponding to message passing directly between processors. The work presented here relaxes this assumption and provides an analytical method for deciding the traffic pattern caused by task graphs when taking the cache-coherence protocol into consideration.

The second contribution (chapter 6) concerns the synthesis of application-specific topologies in the context of MANGO. Six different optimization algorithms are designed and evaluated. An untraditional approach is taken to the optimization objective: Rather than being given a specific goal, the optimization algorithms attempt to find the topology that allows the fastest possible execution of the application providing the designer – who knows the actual target execution time – with knowledge that can be used to decide between adding features to the application or slowing down the interconnect to save power.

The third contribution (chapter 7) is related to the ReNoC architecture [71]. The contributions of this thesis to ReNoC consist of the identification and formalization of the ReNoC configuration problem as well as the design, implementation, and evaluation of a number of optimization algorithms for solving it. The problem consists of three parts: Topology synthesis, topology mapping, and deadlock-free routing in highly irregular network topologies in which bidirectional links can not be assumed.

1.4 Thesis Overview

This thesis is structured in eight chapters of which this introduction is the first. The chapters can be divided in three categories: Background material and related work, contributions, and conclusions and future directions. The following presents an overview and a reader's guide to the remainder of this thesis.

Background Material and Related Work

Chapter 2: Presents an overview of the modelling approaches used in later chapters. Specific models (task graphs and bandwidth graphs) are defined. The reader who is familiar with the subject should pay attention to the definitions, as they are used in later chapters.

Chapter 3: Presents an overview of and introduction to optimization algorithms. The reader who is familiar with this subject may skip most of the chapter, but should skim section 3.6, where the basic technique used for comparing different heuristics in chapter 6 is presented.

Chapter 4: Presents the related work.

Contributions

Chapter 5: Presents the first contribution as outlined above.

Chapter 6: Presents the second contribution as outlined above.

Chapter 7: Presents the third contribution as outlined above.

Conclusions and Future Directions

Chapter 8: Presents the conclusions, puts the presented research into perspective, and discusses future directions.

Chapter 2

Modelling Approaches

This chapter defines some of the models used for representing applications and networks when doing design-space exploration. The chapter focuses on the models that are applied in later chapters. Some of the models are expanded upon in those chapters in order to make them better reflect the applications or networks being considered. The aim of the models is to sufficiently accurately reflect the characteristics of applications and networks-on-chip such that they may be used for performing design-space exploration and for analytically determining whether or not certain properties – such as absence of deadlocks – are fulfilled. Using models for this purpose obviously allows for more rapid evaluation of a larger number of points in the design space than having to actually manufacture each point in order to evaluate it. First, graph-based modelling is discussed in section 2.1 followed by simulation-based modelling in section 2.2.

2.1 Graph-Based Modelling

This section presents various graph-based models for applications and networks.

2.1.1 Applications

This section presents two graph-based models for applications: Task Graphs and Bandwidth Graphs. These models have both been used in NoC research previously, although various different names have been used to describe them. The models capture the behavior of an application at different levels: Task graphs

model dependencies and communication between the computational parts of applications, while bandwidth graphs model the actual communication between IP cores.

Task Graphs

Def. 2.1. A task graph $TG = (T, D)$ is a directed, acyclic graph, where a vertex $t \in T$ represents a task, and an edge $d_{i,j} \in D$ represents a dependency on t_i by t_j. Each vertex t_i represents computation in the application and has an attribute characterizing its computational demand on different IP cores, e.g., by associating an instruction count with each task and a clocks-per-instruction with each IP core [66]. Each edge $d_{i,j}$ represents communication in the application and has a weight $n_{data,i}$ indicating the amount of data transferred along the edge.

Task graphs are useful at modelling static applications, i.e., applications whose execution does not vary significantly due to variations in the application's environment, including inputs and ordering of memory accesses, especially accesses to synchronization variables. For example, safety critical applications with hard deadlines typically exhibit such static behavior, while applications in general may – but do not necessarily have to – do so. When using task graphs to model applications with such variations, care much be taken not to not fall into the following pitfall caused by making too broad assumptions about the correspondence between the model and the actual execution of the application.

Consider the example in Figure 2.1, where a producer puts two workloads (a large one, w_l, and a small one, w_s) on a queue, and two consumers each read the workloads off the queue and perform the associated computations. The task graph in Figure 2.1(e) models this situation. Assuming this task graph is used to map the application on a heterogeneous multiprocessor system with one fast and one slow processor for the consumers, w_l would be mapped to the faster processor. However, during execution, the allocation of workloads to consumer threads depends entirely on the order in which the consumer threads acquire the semaphore s and the order in which workloads are put on the queue. Thus, even if the task graph in Figure 2.1(e) is an accurate model of the application's flow, it is not a useful model for application mapping, as the application's behavior (here the allocation of workloads to consumers) is not static. This is one example, where the model – although correct – does not reflect the application's execution, where it is threads, not workloads, that are allocated to processors, and the allocation of workloads to threads is arbitrary.

Producer(Queue q, Semaphore s)
1: Generate workloads w_l and w_s
2: Pass s
3: Put w_l and w_s on q
4: Release s

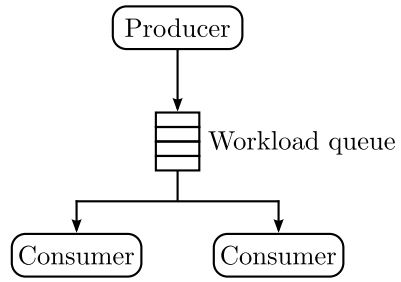
(a)

Consumer(Queue q, Semaphore s)
1: **while** true **do**
2: Pass s
3: Get workload w from q
4: Release s
5: Solve w

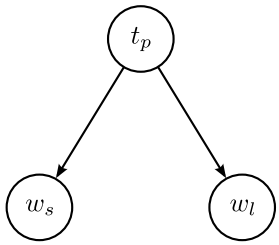
(b)

main()
1: Queue q
2: Semaphore s
3: Start thread **Producer**(q, s)
4: Start thread **Consumer**(q, s)
5: Start thread **Consumer**(q, s)

(c)



(d)



(e)

Figure 2.1: An example of an application modelled by a task graph: (a), (b), and (c) show the application in pseudo code, (d) shows a schematic of the application, and (e) shows a task graph model of the application.

One outstanding issue with respect to task graphs is determining the traffic pattern that mapping a given task graph on a given system results in. So far, it has been assumed that data is communicated directly between processors [66, 41]. However, this requires sufficiently large memories at each processing node, which is unrealistic for applications with data sets larger than a few megabytes in on-chip systems. Chapter 5 presents research into determining the traffic pattern (bandwidth graph) resulting from mapping a task graph on a cache-coherent shared-memory system.

A widely used tool for generating synthetic task graphs is Task Graphs For Free (TGFF) [26].

Bandwidth Graphs

Def. 2.2. A bandwidth graph is a directed graph $BG = (O, C)$, where each vertex $o_i \in O$ represents an IP core and each directed edge $c_{i,j} = (o_i, o_j) \in C$ represents a connection from o_i to o_j. Each edge $c_{i,j}$ has a weight $b_{i,j}$ that indicates the connection's bandwidth requirement.

Bandwidth graphs are used to model the communication between IP cores.[1] The graph can model parts or the entirety of an application's communication, e.g., a bandwidth graph could be used to model the main flows of a signal-processing application, while irregular traffic between IP cores, such as updating filter coefficients, is omitted. Care must be taken when using such partial bandwidth graphs for synthesizing topologies for example, as some synthesis algorithms may produce topologies that are not strongly connected, i.e., a pair of IP cores o_i and o_j may exist such that there is no path in the network topology from o_i to o_j. An example of a bandwidth graph is the VOPD application shown in Figure 2.2 and also in Figure 7.2(a).

Bandwidth graphs are widely used – sometimes with minor adaptations – in literature, although they are given different names in different papers, including Core Graphs [55], Application Task Graphs [78], and Application Characterization Graphs [38].

2.1.2 Networks-on-Chip

Various aspects of networks in general, and thereby also networks-on-chip, may also be modelled using graphs. In the following, the common graph models for

[1] Actually, the modelled communication is between network interfaces. However, the simplification that communication is between IP cores is used in general and in this thesis.

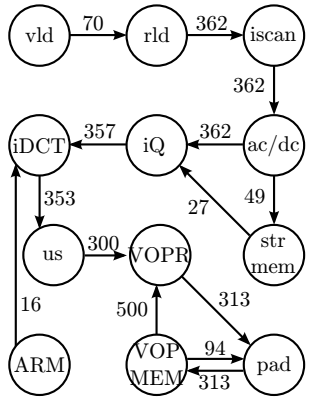


Figure 2.2: An example of a bandwidth graph. The vertices are IP cores, the edges are connections on which data flows, and the weights on edges indicate the bandwidth of the data flowing along the connection.

network topologies and dependencies in networks are described.

Network Topology

Def. 2.3. A network topology graph is a directed graph $NTG = (N, L)$, where each vertex $n \in N$ describes a network node consisting of a router, a network interface, and an IP core, and each edge $l_{i,j} = (n_i, n_j) \in L$ describes a link between the network nodes n_i and n_j. Links may be associated with a capacity, indicating either the peak or sustainable bandwidth that may be communicated along the link.

The model as described here specifies that network nodes consist of one router, one network interface, and one IP core, which is not always the case. For networks with multiple IP cores connected to each router, the model may be changed such that a network node includes all IP cores connected to the router or by letting a vertex describe either a router or an IP core.

The model as described in definition 2.3 is used in chapter 6 in connection with the MANGO network-on-chip, while a more detailed model for modelling ReNoC-based networks-on-chip is presented in chapter 7.

Routing Dependencies

Graphs can also be used to model routing dependencies in the network [20]. This can be used to analytically determine whether routing deadlocks may or may not occur.

Def. 2.4. A dependency graph is a directed graph $DG = (\mathcal{C}, \mathcal{D})$, where each vertex $c \in \mathcal{C}$ represents a channel in the network, and each edge $(c_i, c_j) = d_{i,j} \in \mathcal{D}$ represents a routing dependency by c_i on c_j, i.e., a flit travelling on c_i may continue on c_j.

A dependency graph describes all the possible paths flits may take through the network, and absence of cycles in this graph guarantees freedom from routing deadlocks. Assuming that flits are eventually extracted from the network at their destination, absence of cycles in the dependency graph guarantees absence of deadlocks in general in the network. This assumption is not always valid however: Consider for example a memory module with finite input and output buffers. If the input buffer is full, no further flits carrying requests to the memory can be extracted from the network until the memory has extracted a request from its input buffer. However, in order to do so, the memory needs to make a reply to the previous request in case of a read, which requires it to be allowed to insert a packet in the network. In this way, a dependency arises between two channels through the memory, although no routing dependency exists between the two channels. This type of dependencies is termed *message dependencies* and the deadlocks that may arise from them *message dependent deadlocks*. These have been discussed in the context of Networks-on-Chip by Hansson et al. [34]. These deadlocks are not considered further in this thesis, as the presented methods do not concern the network implementations, but rather assume that the considered networks are free from such deadlocks.

2.2 Simulation-Based Modelling

An alternative approach to determining the behavior of an application or a system is to make a model that may be used to simulate the application or system. Arguably, some of the models described above may be used in simulations, e.g., a bandwidth graph can be input to traffic generators that then produce the specified traffic pattern, but the bandwidth graph by itself can not be simulated. The main difference between graph and simulation-based

modelling is in the type of information that may be extracted from the models: Graph-based and other similar models provide qualitative information, e.g., that routing deadlocks are not possible, which can not be readily determined through simulation, while simulation provides quantitative information, such as a network's saturation bandwidth, which is difficult to determine from graph-based models. Simulation models of applications and networks are discussed in sections 2.2.1 and 2.2.2 respectively.

2.2.1 Applications

As described above, the graph-based models may be used as a basis for simulating the traffic patterns that an application gives rise to. This section concerns variations of executing the actual application in a simulator rather than simulating a model of an application. As the main interest of applications in the context of networks-on-chip is on the communication rather than the computational aspects, the focus is on the communication generated by the application that should be serviced by the network.

Full Simulations

Given an application, the most straight-forward way of determining its traffic pattern is to simulate the execution of the application on the intended target system. In this way, the expected traffic pattern for one or more given use-cases can be determined. However, as discussed for task graphs previously, even small variations in the system may lead to completely different traffic patterns: Using the same example as in Figure 2.1, assuming that the data sets for w_s and w_l are located in different on-chip memories or accessed through different memory controllers, a change in the order in which the semaphore is acquired changes the traffic pattern significantly, as the processors executing the consumer threads access different memories depending on the order of acquiring the semaphore. Therefore, in some cases, a single run of the simulator does not produce one traffic pattern that is representative for all possible executions of the application. However, if all relevant aspects of the application and the system on which it is executing are deterministic, a simulation yields the exact traffic pattern produced by the application.

The disadvantage of full system simulations is the fact that such simulations are very time consuming. Making use of simulations for design-space exploration leads to longer iterations than is the case when using models. One method for speeding up the simulation is to perform the computational parts of the appli-

cation natively on the computer running the simulation, while only simulating the interconnect. Such an approach is possible using SystemC [40]. Another method is to perform the full simulation once and record the communication in a manner that allows it to be played back, as is described in the following.

Trace-Based Simulations

An alternative to doing a full system simulation in every iteration of a design-space exploration, is to do the full system simulation once, and – in the case of shared-memory architectures – record every memory access made by every processor, thereby creating a memory access trace that may be played back essentially recreating the application's traffic pattern. The advantage of this approach is the omission of actual computation during the simulation: Only the parts of the application that are relevant to the interconnect are simulated, potentially speeding up the simulation significantly.

One implementation of such a trace-based simulation is the Reactive IP Emulator (RIPE) [1, 46, 48, 47]. The reactive part of RIPE comes from the fact that the trace contains information about both semaphores and the timing of requests. The timing information allows the emulator to be reactive to variations in the interconnect's latency and the latency of processing memory requests, e.g., the emulator waits for a specific number of clock cycles after receiving the response to a read before issuing the next memory request emulating the IP core being stalled while waiting for the response, then performing the required computation, and then issuing the next request. Changes in the interconnect's latency and the utilization of the memory are thus reflected in the generated traffic pattern.

Semaphores are handled by emulating the application's behavior upon encountering a semaphore instead of playing back the recorded trace. For example, the recorded trace might indicate that the semaphore is acquired on the 200th attempt at doing so, and playing back this recording would only be correct, if exactly 200 attempts are actually required to pass it. In truth, changes in the system during design-space exploration may cause the correct number of attempts to change significantly, thereby making the trace-based simulation inaccurate. Therefore, the emulator emulates the behavior of these parts rather than simply plays back the recorded trace. Although this produces the correct behavior of accessing semaphores, it does not capture the consequences of changing the order in which different threads or processes acquire the semaphore: Again using the example in Figure 2.1, the processor that executed w_l when the trace was recorded emulates executing w_l in all subsequent playbacks of the trace

irrespectively of the order in which the semaphore is acquired by the different processors.

2.2.2 Networks-on-Chip

While the previous section discussed simulating the application in a full system simulation, this section concerns simulating the interconnect. Simulation models of a network-on-chip – like any other hardware component – range from register-transfer level (RTL) models over behavioral models to high level models [72, 24]. The trade-offs of these different modelling levels – higher accuracy and slower simulation at lower levels of modelling and lower accuracy and faster simulation at higher levels of modelling – are considered well-known, and are not discussed further here.

Chapter 3

Optimization Algorithms

This chapter gives a brief introduction to the optimization or search algorithms that have been used in the related work and that form the basis of the optimization algorithms described in chapters 6 and 7. A reader who is familiar with this subject may skip most of this chapter, but should pay attention to section 3.6, where it is presented how to compare different heuristics.

Many of the challenges or problems in NoC research listed in section 1.2 require a vast problem space to be searched for a solution that both is optimal with respect to a certain optimization criteria (e.g., power consumption, latency, throughput) and satisfies one or more constraints (e.g., freedom of deadlocks in routing, upper bound on router sizes, maximum power consumption, minimum throughput). Common for these problems is that the search space typically is too large to be searched exhaustively for the optimal solution, and that no method has yet been found that is guaranteed to find the optimal solution with less than an exhaustive search. In the following, the main focus is on solving optimization problems using heuristics. A few papers in relation to NoC have used linear programming methods to solve some of the related optimization problems [68, 69]. However, these methods fall outside the purview of this thesis.

This chapter first presents the general terminology used in describing optimization problems and heuristics for solving them. Then, four different meta-heuristics (greedy algorithms, simulated annealing, tabu search, and genetic algorithms) are described, and finally, the methodology used for comparing heuristics is presented. A textbook on the subject of meta-heuristics is [15], which contains most of the meta-heuristics described here. The only exception

is greedy algorithms, which are discussed in many textbooks, including [19].

3.1 General Terminology

This section describes the general terminology used in optimization or search algorithms or heuristics.

Optimization objective: The optimization objective is the measure that is being optimized, e.g., power consumption. Some optimization problems have multiple optimization objectives.

Constraints: An optimization problem can specify a set of constraints that a solution must satisfy.

Solution: A solution to an optimization problem is any solution that satisfies the given constraints. The solution that has the optimal value for the optimization objective is denoted the optimal solution. The *best* solution is typically used to denote the best found solution during the search, i.e., the solution for which the value of the optimization objective is closest to that of the optimal solution. In most cases, the best solution is not the optimal one, while if it is the case that the optimal solution has been found, it is often very difficult to prove.

Evaluation function: A function that calculates the value of the optimization objective for a given solution.

Search space: The search space is the set of solutions that is searched in order to find the best solution. Due to the size of the search space, this set is normally not physically maintained in the implementation of a heuristic. Rather, a method is provided for producing a solution, either from scratch or based on one or more given solutions. Procedurally generating solutions may lead to invalid solutions, i.e., solutions that violate the constraints specified in the optimization problem, and that therefore are outside of the search space. How such invalid solutions are handled depends on the specific heuristic, but the common options are to ignore them or to change them in such a way that they become valid.

Initial solution: Most heuristics start out with one or more initial solutions. This may be any valid solution and is typically either a randomly generated

solution or generated by a method that is expected to yield a good solution. The second approach may however lead to initial solutions close to a local optimum rather than the global optimum, artificially restricting the search. A compromise between these extremes is to add non-determinism to the method used to generate what is expected to be a good solution, e.g., GRASP [29]. However, this approach may also lead to initial solutions that are close to local optima, but far from the global optimum.

Multi-objective optimization: Some optimization problems require multiple objectives to be optimized, e.g., synthesizing a NoC topology with the best combination of high bandwidth and low power consumption. With a single objective, deciding if one solution is better than another one is easy: The lower the power consumption, the better. With multiple objectives making this decision becomes more difficult: Is a topology that can support a uniform traffic pattern at 50 Mb/s using 10 mW better than one that can support the same traffic pattern at 100 Mb/s using 20 mW? In some cases, the multiple values can be combined in one using addition or multiplication, thereby allowing the use of a heuristic that optimizes a single objective. However, if the individual values of objectives are of interest, the notion of a single best solution disappears. Instead, a set of *non-dominated* solutions is found, where one solution is said to dominate another solution if and only if it is no worse in all objectives than the other solution, and it is strictly better in at least one objective than the other solution [15]. The optimization algorithm thus provides the user with a number of solutions from which the user must decide the best one.

Parameters: Most meta-heuristics have one or more parameters for which good values need to be found for each optimization problem the heuristic is applied to. These parameters are described together with the heuristic in the following sections, while the method for evaluating the quality of a given set of parameter values is described in section 3.6.

Move: Some heuristics (simulated annealing, tabu search, etc.) make use of walks through the solution space. A move is a change in a solution according to a well-defined method, e.g., changing which IP core a task is mapped to in the application mapping problem.

Neighborhood: A solution's neighborhood consists of those solutions that may be reached in a single move from the given solution. In general, an n-bit neighborhood is the set of reachable solutions in a single move,

where a move is defined as changing n "bits" in the solution, e.g., changing which IP cores n tasks are mapped to in the application mapping problem.

Given a solution, choosing which move to make can be done in different ways, depending on the heuristic. Some heuristics choose a random move, others make an exhaustive search of the neighborhood and choose the best move, while yet others start an exhaustive search of the neighborhood but terminate the search when the first move producing a better solution than the current one has been found. The preferred method depends on both the heuristic and the type and size of the problem.

The meta-heuristics described here may also be used to iteratively construct a single solution. In this case, a move adds (or possibly subtracts) from the solution being created.

3.2 Greedy Algorithms

Greedy algorithms are the conceptually simplest of the optimization algorithms. First, an initial solution is found using one of the methods described above. Then, in each iteration, the move that leads to the best neighboring solution is made, i.e., the move for which the evaluation of the resulting solution is the best. Depending on the implementation of the algorithm, termination may occur in one of two cases:

- With the above description of greedy algorithms, it is possible for the algorithm to take one move away from a local optimum, as the neighborhood in general does not include the current solution. However, in most cases, the algorithm will return to the local optimum in the next iteration. Thus, the termination criterion should be that the same solution is encountered for the second time.

- Alternatively, the search can be terminated when no move exists that improves the current solution.

The disadvantage of greedy algorithms is that no mechanism exists for escaping local optima. Even in the first case above, where it is possible for the algorithm to take one move away from the local optimum, it will return in the very next iteration. This is caused by a lack of memory, which is addressed by the tabu search meta-heuristic described in section 3.4.

3.3 Simulated Annealing

Simulated annealing provides a mechanism for escaping local optima in the form of randomly accepting a move which results in a worse solution than the current one. This is done by making use of an exponentially decreasing temperature

$$\tau_i = {\tau_{start}}^{\alpha^i} \tag{3.1}$$

where τ_i is the temperature in iteration i, τ_{start} is the starting temperature, and α is the rate at which the temperature decreases. Given the current solution, s_c, a move resulting in a new solution, s_n, an evaluation function, $eval(s)$, and a random number between zero and one, r, the new solution is accepted – becomes the current solution for the next iteration – if either of two conditions are true:

$$eval(s_n) < eval(s_c) \tag{3.2}$$

$$r < \exp\left(-\frac{(eval(s_n) - eval(s_c))}{\tau_i}\right) \tag{3.3}$$

The condition in equation 3.2 states that the new solution will be accepted, if it is better with respect to the optimization criteria than the current solution. This is similar to the behavior of greedy algorithms. The condition in equation 3.3 gives a probabilistic acceptance of a worse solution with respect to the optimization criteria. The probability of accepting a worse solution depends on the iteration number (through the decrease of the temperature) and on the difference between the quality – measured by the evaluation function – of the current and new solutions: As the temperature decreases, the probability also decreases, while as the difference decreases, the probability increases, i.e., a new solution that is not significantly worse than the current one has a higher probability of being accepted than a new solution that is much worse than the current one. Through the condition in equation 3.3, simulated annealing is able to escape a local optimum.

Simulated annealing has two parameters: The starting temperature τ_{start} and the rate of cooling α. The intervals for these parameters' values are $]0; \infty[$ for τ_{start} and $]0; 1[$ for α, both belonging to the set of real numbers.

3.4 Tabu Search

Tabu search is another meta-heuristic with the capability of escaping local optima. The basic functionality is similar to that of greedy algorithms, where

the entire neighborhood is searched for the best move, but memory is added to the algorithm to avoid the situation where one move is taken out of a local optimum, only to take the same (or inverse) move back to the local optimum in the very next iteration.

This added memory is a list of the most recently encountered solutions or moves made, which are disallowed as long as they reside in the list – they are said to be tabu. A move (or solution) is added to the list when it is taken and is removed again after a certain number of iterations. In essence, the list works as a fixed capacity FIFO buffer, where one move is inserted at the front of the buffer, and another move falls off the end of the buffer in every iteration. The number of iterations that a move is tabu is called the length of the tabu list. This is tabu search's only parameter and may take any natural number as its value.

The tabu list represents a case of short term memory: The memory only goes back as many iterations as the length of the tabu list. Depending on the nature of the optimization problem, longer term memory than what can reasonably be provided by tabu lists may be required to escape local optima. The means for acquiring this long term memory is highly dependent on the optimization problem, but one example is a counter that keeps track of the number of iterations without finding a new best solution. This long term memory may for example be used to detect that the search is in an uninteresting part of the search space.

When the long term memory indicates a lack of progress, a change in the search can be forced. This is typically done as either an *intensification* or as a *diversification*. Intensifying the search means that the search focuses on the area around a known, good solution to see if an even better solution may be found in that area. Diversifying the search means that a change is made to the current solution such that the search is forced to a different part of the search space. As with deciding on the means for acquiring long term memory in the tabu search, deciding on what to do, when that long term memory indicates a lack of progress is highly dependent on the optimization problem.

3.5 Genetic Algorithms

Genetic algorithms are a different type of heuristic compared to the other heuristics described here. The name and functionality of the meta-heuristic are inspired by biology. The meta-heuristic brings Darwinian evolution to optimization problems: Instead of a single current solution, a set of current solutions – called the population – is maintained. In each iteration, a number of solu-

Task (array index)	0	1	2	3	4	5	6	7
IP core	3	1	2	2	0	3	1	2

Figure 3.1: A representation of a solution to the task mapping problem. In genetic algorithms, each task's mapping is considered a gene in the solution.

tions are paired to produce offspring (new solutions) consisting of parts of both parent solutions, and a number of solutions are eliminated from the population. Additionally, a mutation may be applied to the new solutions in the population. The aim of the heuristic is to have good solutions evolve into better ones, while bad solutions are pruned from the population.

Each solution is considered as made up of a number of genes, each gene describing part of the solution. As an example, consider the application mapping problem: If the solution is represented by an array describing which IP core each task is mapped to as illustrated in Figure 3.1, each element in the array is considered a gene. When pairing two solutions, a crossover operator is used to determine which genes are taken from each of the parent solutions and put into the child solutions, each child getting a complementary set of genes from each parent. Considering the example in Figure 3.2, the parent solutions are shown in (a) and (b) with a crossover point between genes 3 and 4. The child in (c) receives all the genes on one side of the crossover point – indicated with gray – from one parent and all the genes on the other side of the crossover point from the other parent, while the child in (d) receives the complementary set of genes from each parent. While the crossover point is typically decided randomly for each pair of parents, the number of crossover points is a parameter of the meta-heuristic. When multiple crossover points are used, the children receive the set of genes from the first gene to the first crossover point from one parent, the set of genes between the first and second crossover points from the other parent, the set of genes between the second and third crossover points from the first parent and so on alternating between the parents.

Multiple factors influence on how solutions are chosen to be parents: Both the speed with which the population should be renewed and the weight that should be put on good solutions in the population can be adjusted. Two types of evolutions are *bacteria* and *elephant* evolutions, which are named for their respectively fast and slow renewing of the population: In bacteria evolution, all solutions in the population are randomly paired, thereby changing the entire population in a single iteration, whereas in elephant evolution, two solutions are pruned from and another two solutions added to the population in each

Task	0	1	2	3	4	5
IP core	0	1	2	3	0	1

(a) Parent 1

Task	0	1	2	3	4	5
IP core	3	2	1	0	3	2

(b) Parent 2

Task	0	1	2	3	4	5
IP core	0	1	2	3	3	2

(c) Child 1

Task	0	1	2	3	4	5
IP core	3	2	1	0	0	1

(d) Child 2

Figure 3.2: An example of a crossover operation. The parents are shown in (a) and (b), while the children are shown in (c) and (d).

iteration. Choosing which two solutions to pair and which to remove in elephant evolution can be done randomly with a uniform probability, randomly with the probability weighted by the solutions' quality, or in any number of other ways. Going into more detail on these points is beyond the scope of this introduction to the subject.

Mutations are very similar to moves in simulated annealing and tabu search: One or more genes are changed to a random value. The number of genes to change is a parameter of the meta-heuristic and corresponds to the number of bits in the neighborhood.

This section has given a very generic introduction to genetic algorithms. For each of the operators (crossover and mutation), it is also possible to define problem specific versions. Additionally, problem specific operators may also be designed together with evolutions that fall between the extremes of elephant and bacteria evolution. The parameters for the generic genetic algorithms presented here include the population size, the number of crossover points, the number of mutations to make in each iteration, the type of evolution, and how to select which solutions to pair. For a genetic algorithm that has been adapted to a specific problem, the number of parameters may increase further. In summary, genetic algorithms have many aspects to customize in order to create a heuristic for solving a specific optimization problem.

3.6 Heuristic Comparison

Before a meaningful comparison of multiple heuristics for solving a given optimization problem can be made, good values need to be found for the heuristics' parameters. The process for finding these parameter values is called *parameter tuning*, and the steps outlined for tabu search in [15] are generally applicable to other heuristics.

The aim of parameter tuning is to find those parameter values that consistently produce good results across a set of representative benchmark applications. Finding these values is accomplished by experimenting with different values for each parameter. Either parameter values are varied individually, in groups, or all at the same time. In the first case, experiments are first done to determine which parameters have only a small influence on the heuristic's performance, and values are locked for these parameters. Then, the remaining parameter values are decided by experimenting with one parameter value at a time. In case two or more parameters influence on each other, these parameters should be varied as a group, i.e., a good value should not be found for one parameter at a time while assuming certain values for the others. Instead, all of the interdependent parameters should be varied together. Alternatively, all the parameters may be varied as one large group, i.e., a range of values are decided for each parameter, and all combinations of these values are then evaluated. This type of tuning may be very time consuming though.

Independently of the tuning method, it should be ensured that the best found parameter values are not on the edge of the explored values. If this is the case, the search should be extended to include the neighboring values iteratively until the best parameter values are no longer on the edge of the explored area. Parameter tuning is an optimization problem in itself, and the process outlined above leads to a local – possibly the global – optimum for the parameter values.

After determining the parameter values that yield the best results, the heuristics are applied to a second set of benchmarks using these parameter values. This process is called *parameter testing* and uses a second set of benchmarks to avoid situations, where the heuristics have been specialized to such a degree that they produce very good solutions to a limited set of benchmark problems but produce bad solutions to all other problem instances. A comparison of two or more heuristics is based on the solutions they produce during parameter testing. In order to provide a fair comparison of the heuristics, they should be given equal conditions in their execution. Typically this means a fixed execution time for all heuristics.

As many optimization problems have a high degree of variation in their

objective function over different problem instances – consider for example the power consumption in a 16 versus a 1 024 node NoC – a measure of the quality of the solutions produced by the heuristics that is independent of the problem size is required. For this purpose, the *percentage gap* is introduced. This measure describes the percentage difference between the result of a single execution of the heuristic, z, and the optimal solution, z'. It is defined for problems where the objective is to minimize or maximize z respectively as:

$$e^{\min} = 100\frac{z - z'}{z'} \tag{3.4}$$

$$e^{\max} = 100\frac{z' - z}{z'} \tag{3.5}$$

Due to the nature of most optimization problems, the optimal solution is typically not known. The best known solution for the given problem can then be substituted for the optimal solution in this calculation.

Another issue to take into consideration when comparing heuristics and when performing the parameter tuning is the fact that most heuristics include a certain degree of non-determinism, e.g., randomly finding initial solutions and randomly choosing a move to take. In order to prevent a single exceptionally good or bad random choice in the execution of the heuristic from leading to false conclusions, multiple samples of the heuristic's performance should be taken. Calling the number of samples N, the *average percentage gap* is found by averaging the samples' percentage gaps:

$$E = \frac{\sum_{i=1}^{N} e_i}{N} \tag{3.6}$$

As the percentage gap is a statistical measure of a heuristic's ability to find good solutions, not only the average, but also the standard deviation should be used in the comparison:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(e_i - E)^2}{N - 1}} \tag{3.7}$$

Determining the best parameter set during tuning or the best heuristic during testing thus requires considering both E and σ: Lower values of E signify that the solutions are closer to optimality on average, while lower values of σ signify consistency in the quality of the solutions. Obviously, if both of these values are low, a good parameter set has been found, but more commonly, a trade-off needs to be made between average performance and consistency.

Chapter 4

Related Work

This chapter gives an overview of the related work. First the literature on application mapping is described in section 4.1, then the related research into synthesizing network-on-chips is discussed in section 4.2 followed by a brief overview of application-specific routing in on-chip systems in section 4.3. In section 4.4, the NoC architectures in which the optimization algorithms presented in chapters 6 and 7 operate are briefly introduced. More thorough descriptions of these architectures are given in the respective chapters. Finally, section 4.5 presents related work on deriving a traffic pattern from an application model.

4.1 Application Mapping

This section presents the related work on application mapping. Although no research in methods for automating application mapping is presented in this thesis, this description of the related work is included for the sake of completeness. The term "application mapping" covers multiple problems: It can be used both to describe the mapping of a task graph (definition 2.1) on a set of IP cores and to describe the mapping of a set of IP cores on interconnect interfaces. These two cases are treated separately in the following.

4.1.1 Task Graph on IP Cores

This specification of the application mapping problem can be formulated as follows.

Def. 4.1. Given a task graph $TG = (T, D)$ as defined in definition 2.1 and a set of IP cores O, find a function $M : T \rightarrow O$ that maps each task $t \in T$ to an IP core $o \in O$. Typically, some objective is specified that the mapping should optimize.

One example of this type of application mapping is proposed by Lei and Kumar [41]. An optimization algorithm in two steps is presented, where the first step finds a mapping of tasks to IP cores such that the overall execution time of the application modelled by the task graph is minimized. In the second step, the mapping of IP cores on interconnect interfaces described in the next section is solved with the same optimization objective. The difference between the two steps is the accuracy of the underlying model of the system, where a coarse model is used in the first step, and a more fine-grained model is used in the second step.

Another approach is taken by Madsen et al. [44], who use a genetic algorithm to find both the mapping of tasks to IP cores and of communication to links with the constraint that deadlines are met while optimizing multiple objectives: Power consumption, memory size, buffer sizes in the interconnect, and component cost. The presented algorithm can be used both to map tasks to a static architecture (set of IP cores) and to synthesize the architecture and map the tasks on it simultaneously.

Manolache et al. [49] address the problem of mapping communication to NoC links in a scenario where faults may trigger packages to be dropped. The aim of the mapping is to minimize power consumption, while satisfying certain message-arrival probabilities and task deadlines.

Mapping applications onto the ReNoC and MANGO NoCs considered in later chapters is an interesting problem to consider in the future. However, the methods presented in literature are not directly applicable, as they do not consider the special characteristics of these two NoC architectures.

4.1.2 IP Cores on Interconnect Interfaces

The second specification of the application mapping problem can be stated as follows:

Def. 4.2. Given a bandwidth graph (definition 2.2) $BG = (O, C)$ and a network topology graph (definition 2.3) $NTG = (N, L)$, find a function $M : O \rightarrow N$ that maps the IP cores in O on the network nodes in N.

Multiple papers have addressed this problem in the context of different network-on-chip architectures and network topologies.

Hu and Marculescu [38, 37, 39] have considered mapping of IP cores and routing in a 2D mesh topology with the objective of minimizing power consumption constrained by performance (bandwidth) requirements. A branch-and-bound algorithm is used to solve the problem. A similar use of constraints for performance requirements while optimizing power consumption is used in the context of ReNoC in chapter 7.

Ascia et al. [2, 3] have considered mapping IP cores to tiles in a 2D mesh topology with multiple optimization objectives. A genetic algorithm is used to optimize both performance (measured as execution time) and energy consumption.

Murali and De Micheli [59, 60] map the IP cores on multiple different topologies and then select the best topology and mapping with the objective of minimizing the sum of products of bandwidth requirements and distance between IP cores. The optimization is performed in context of the ×pipes NoC [21, 6]. This topology selection has also been extended with floorplanning of the IP cores and network routers [56].

Lu et al. [43] extend a simulated annealing heuristic with clustering of IP cores to speed up the search compared to a basic simulated annealing heuristic. The speed-up is achieved without worsening the results produced by the heuristic.

4.2 Network-on-Chip Synthesis

In this section the related work on synthesizing application-specific network-on-chip topologies is presented. Various approaches have been used in various network architectures.

Bolotin et al. [14] optimize a mesh topology by removing unused links and router ports in the context of QNoC [12]. Routing in such a degenerated mesh is described in [13].

Ogras and Marculescu [62] take the opposite approach and customize mesh topologies by adding application-specific long-range links in order to increase the network's saturation bandwidth.

Murali et al. [61] have presented a method for synthesizing an application-specific topology, while taking the synthesized system's floorplan into consideration. The method also provides deadlock-free routing algorithms while improving both performance and power consumption over regular network topologies.

Chan and Parameswaran [18] show further improvements by extending Murali's method [61]. Point-to-point links are inserted directly between frequently

communicating IP cores, while taking into consideration the cost of adding additional communication interfaces to the IP cores.

Hansson et al. [35, 33] present a combination of mapping groups of IP cores to interfaces, synthesizing a network topology, finding routes through the network, and allocating time-slots to connections in context of the networks-on-chip with time division multiple access mechanisms, such as Æthereal [65] and Nostrum [52, 51]. Instead of mapping individual IP cores to interfaces, groups of IP cores share a single interface using time-slots to arbitrate access to the network and to provide guaranteed service. The unified mapping, routing, and time-slot allocation problem is solved through an extension of path selection. The approach taken for optimizing in ReNoC in chapter 7 also combines topology synthesis and routing in a single problem.

Hansson's approach solves the problem for a single bandwidth graph. Murali et al. extend the approach for multiple bandwidth graphs or multiple use-cases by finding the worst-case use-case (combining all the bandwidth graphs into one) [57] and by restricting the possible combinations of bandwidth graphs [58]. This approach is further extended by Hansson [31] to allow for some applications to be persistent across switches between different use-cases. Hansson et al. also present a methodology for making these use-case switches in a safe manner [32].

While many approaches to synthesizing application-specific networks-on-chips certainly exist, they are not directly applicable to the network architectures presented in later chapters: MANGO is sufficiently different from other network architectures to make porting an optimization algorithm very difficult, while in ReNoC, the application-specific topology is instantiated at run-time rather than at design-time: Using one of the above approaches most likely results in topologies that are infeasible to map on ReNoC. Instead, the topology synthesis and mapping problems are combined in one problem in chapter 7.

4.3 Application-Specific Routing

Synthesizing application-specific network topologies necessitates application-specific routing in order to actually use the synthesized networks, although application-specific routing can also be applied to regular topologies. Duato [27] has developed the method of using channel-dependency graphs to design deadlock-free, adaptive routing algorithms. Palesi et al. [63] has extended this work by taking an application's traffic pattern into consideration in order to design application-specific, deadlock-free, adaptive routing algorithms. A similar approach is used in chapter 7 for developing deterministic, application-specific,

deadlock-free routing algorithms in ReNoC.

4.4 Network-on-Chip Architectures

Chapters 6 and 7 present algorithms for optimizing in two different NoC architectures: MANGO [7] and ReNoC [71]. In-depth descriptions of these architectures are given in the respective chapters.

Modaressi et al. [53, 54] have presented a NoC router architecture with characteristics resembling a reduced version of ReNoC implemented using prioritized, virtual circuits as in MANGO. In this way, individual connections are given high-priority communication through the network, bypassing many of the arbitration mechanisms. Only one connection can be prioritized in each router port in Modaressi's approach, while MANGO allows multiple connections with each connection having a unique priority to share each router port. Additionally, MANGO decouples the usual connection between bandwidth and latency, which prevents starvation of connections with lower priorities. In ReNoC, multiple connections share the physically circuit-switched long links, while only one connection can make use of the virtually circuit-switched long links in Modaressi's approach.

The router presented by Modaressi represents an interesting point in the design space and may in fact (as any other router) be combined with the ReNoC architecture.

4.5 Derivation of Bandwidth Graphs from Task Graphs

Chapter 5 deals with analytically determining the traffic pattern (bandwidth graph) caused by a task graph mapped to a cache-coherent shared-memory system. The derivation of a bandwidth graph from a task graph has not been explicitly considered in literature. Rather, multiple papers [41, 66, 44, 49] have assumed that message-passing semantics for inter-task communication translate to message-passing communication between the IP cores these tasks are mapped to, resulting in a one-to-one mapping of edges in a task graph to edges in a bandwidth graph.

Chapter 5

Analytical Derivation of Bandwidth Graphs

Chapter 2 presented two graph-based application models: Task graphs (definition 2.1) and bandwidth graphs (definition 2.2). This chapter considers how to analytically derive a worst-case bandwidth graph from a task graph assuming a cache-coherent shared-memory architecture. Considering the NoC design flows discussed in section 1.1, this derivation is useful for determining the traffic pattern after the task graph has been mapped to IP cores. Using an analytical approach allows determining the bandwidth graph much faster than through simulation. The chapter is based on the work presented in [74].

5.1 (Distributed) Shared Memory

In a shared-memory organization, all memory in the system is shared between all processors. From a programming point of view, all communication between different processes and threads go through variables stored in memory. Synchronization is done on special synchronization variables that are also stored in memory and accessed using special instructions such as atomic test-and-set operations or load-linked and store-conditional [36].

One of the main issues in shared-memory systems is ensuring memory consistency between caches belonging to different processors [36]. In systems using a network structure as the interconnect, broadcast-based methods such as snooping on other processors' memory accesses would quickly saturate the in-

terconnect, making such approaches impractical. Here, a cache-coherent non-uniform memory access (CC-NUMA) architecture using a directory to ensure cache-coherence is therefore assumed. This system is sketched in Figure 5.1.

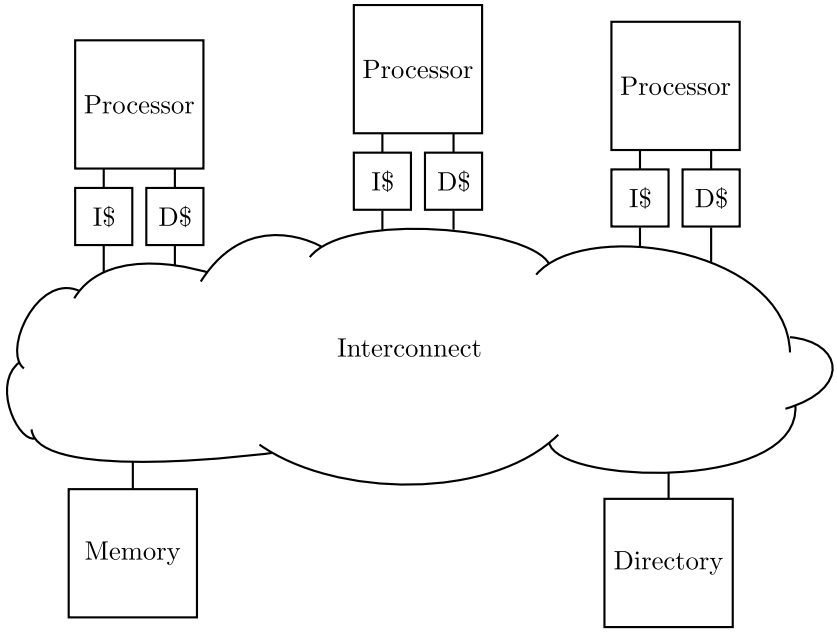


Figure 5.1: A CC-NUMA system consisting of processors with caches, memories, and one or more directories is assumed.

For maintaining memory coherence, a simplified protocol similar to the MSI protocol [20] is used: As the focus here is on the traffic generated in the system, i.e., the data and the protocol messages, the intricacies of the protocol are not modelled. Thus, only the overall flow of protocol messages and data is included in the model. In the considered protocol, a cache-line may be in one of the following three states: Modified, shared, or invalid. In the following, the protocol messages caused by respectively load and store instructions are described.

Load: When a processor issues a load instruction, first its local cache(s) are checked for the requested data. In case of a cache hit, no contribution is made to the traffic pattern. In case of a cache miss on the other hand, the

cache sends a message to the directory requesting a copy of the cache-line containing the address. At the directory, two different cases may occur depending on the state of the requested cache-line:

- If the cache-line is not present in the system, the request is forwarded to the memory, which retrieves the cache-line and sends it directly to the requesting processor.
- If the cache-line is present in the system, i.e., cached in one or more other processors' caches, the request is forwarded to one of these processors, which sends a copy of the cache-line in question to the requesting processor.

Finally, at a later point in the execution of the application, the cache-line may be evicted from the cache, which requires the processor to send a message to the directory, informing it of this such that the list of caches containing the cache-line can be updated.

Store: Similar to loads, the first action on a store instruction is to check whether the local cache(s) contain the requested address. As for loads, a miss may occur, indicating that the cache-line is not present in the local cache, triggering the same events and messages as for a load to bring the cache-line in to the local cache. However, even if the cache-line is present in the cache, the processor is not necessarily allowed to modify it, as other caches may also contain the cache-line. These other copies need to be invalidated before the processor is allowed to modify its local copy. Thus, a request is sent to the directory, which sends out messages to all other caches containing a copy of the cache-line in question, informing them to invalidate the cache-line. Additionally, whenever a modified line changes state, either through eviction or because another processor requests a copy of the line, the directory sends a message to the cache containing the modified cache-line instructing it to send the updated data to main memory.

5.2 Analytical Derivation of Bandwidth Graphs

Above, the shared-memory architecture including the cache protocol was described. The following discusses how to analytically derive a variation of a bandwidth graph from a task graph in such a system. The variation consists in changing the unit on the bandwidth graph's edges from a bandwidth to an

amount of data. The average bandwidth on each edge can be found by dividing this amount of data by the application's execution time. The contributions to the bandwidth graph come from both data being transmitted between different caches and memory and from the cache-coherence protocol messages. The analysis discerns between these two types of messages, and in the following, "protocol message" refers to a packet not carrying any data and "cache-line message" refers to a packet carrying a cache-line.

In order to use a task graph for the purpose outlined above, it is necessary to augment the task graph model with information about the memory locations used for inter-task communication. For each edge in the task graph, a property α is added that represents the set of addresses on which data is exchanged between the two tasks connected by the edge. It is then assumed that the edge's source writes its output to these addresses, while the edge's target reads its input from the addresses. It is assumed that the modelled application does not contain any race conditions. This requirement can be stated as follows:

Given a task graph TG, an address a, and a set E of all edges in TG for which $a \in \alpha$, for any pair of edges e_0, e_1 in E, a path exists either from the target of e_0 to the source of e_1 or from the target of e_1 to the source of e_0, or a single task is the source of both e_0 and e_1. Using $\mathrm{src}(e)$ and $\mathrm{tgt}(e)$ for the source and target of an edge respectively, this can be formalized as

$$\forall e_o, e_1 \in E : \exists \langle \mathrm{tgt}(e_0) \dots \mathrm{src}(e_1) \rangle \vee \exists \langle \mathrm{tgt}(e_1) \dots \mathrm{src}(e_0) \rangle \vee \mathrm{src}(e_o) = \mathrm{src}(e_1). \tag{5.1}$$

This formulation ensures that all tasks reading the value written to a by their respective predecessors have actually done so before another task writes a new value to a, which means that no race condition can exist.

For the analysis presented here, the following assumptions are made:

- For each edge, α constitutes a contiguous range of addresses.
- If multiple edges have a given address $a \in \alpha$, all the addresses found in α on each edge are found in α for all these edges, e.g., if one edge e_x has $\alpha_x = [0; 63]$, any other edge e_y with any address between 0 and 63 in α_y must also have $\alpha_y = [0; 63]$.
- Each edge's address range is aligned to cache-lines. Together with the above item, this prevents false sharing in the application.
- Each task reads each of its input- and writes each of its output-addresses sequentially, and these are the only memory accesses made by the tasks.

- No race conditions exist in the application.
- The cache organization is fully associative with a least-recently-used replacement policy.

Finally, a map of tasks to IP cores, $M : T \rightarrow O$, is required.

The assumption that memory accesses are performed sequentially has the effect that each task has at most one miss per cache-line it either reads or writes. If a task both reads from and writes to a given cache-line, two misses may be produced by the cache-line, one for each access. Given the above assumptions and requirements, each task's contribution to the bandwidth graph can be derived as follows.

The analysis is made by looping over all tasks and for each task looping over all edges incident on the task with a separate analysis being made for input- and output-edges corresponding to reads and writes respectively.

For an input edge e_i of a task t, the contiguous address range α is divided by the cache-line size, cls, and rounded up to get the potential number of misses, which equals the number of cache-lines cl that α is spread over. As outlined above in the description of the cache-coherence protocol, each of these misses produces one protocol message from the IP core to the directory, one protocol message from the directory to a cache containing the requested cache-line or to main memory, and one cache-line message from main memory or another cache to $M(t)$'s cache (the cache of the IP core t is mapped to) in order to send the actual cache-line.

Using t_p to describe the task that is the source of e_i, i.e., that of t's predecessors that writes α, the list of caches that potentially contain the requested cache-line can be determined analytically: Given that t reads the cache-line, t_p must have written it, thus it may be found in $M(t_p)$ cache. Additionally, if t_p has multiple output edges with the address range α, each of the tasks that are the destinations of these edges (denoted by T_s) may have already read or be in the process of reading α. Thus, the cache-line may also be found in t's siblings' IP cores' caches. As t and the tasks in T_s may be reading α concurrently, it is possible for $M(T_s)$'s caches to be instructed to send the full data set to $M(t)$'s cache, even if the data set is larger than the cache. However, t_p has finished accessing α, when t is reading it. The largest amount of data that can be transmitted from $M(t_p)$'s cache to $M(t)$'s cache is therefore the size of $M(t_p)$'s cache.

Based on the above, the set $\mathcal{C}$ of caches that may contain the cache-lines in α can be determined as well as whether the full data set or at most as many

cache-lines as the cache holds may be found in each of these caches. The set of IP cores these caches are attached to is given by

$$\mathcal{C} = (\{M(t_p)\} \cup \{M(t_s) : t_s \in T_s\}) \setminus \{M(t)\}. \tag{5.2}$$

The caches described by the second term (t's siblings) may – due to concurrent execution – hold every cache-line of α when it is requested by t. The full data set may thus be sent from these caches to $M(t)$'s cache, while the amount of data sent from $M(t_p)$'s cache is bounded by the cache size. If $M(t_p)$ is included in $\{M(t_s) : t_s \in T_s\}$, i.e., one of t's siblings is mapped to the same IP core as t_p, the full data set may be sent from this IP core's cache to $M(t)$'s cache. If t is mapped to the same IP core as one of its siblings or its predecessor, no contribution is made to the bandwidth graph for this IP core.

Using cs for the cache size and ds for the size of the data set α addresses (α times the word size), the following contributions are added to the bandwidth graph for a read: cl protocol messages from $M(t)$ to the directory, either cl or $\lceil \frac{\min(ds,cs)}{cls} \rceil$ protocol messages from the directory to the unique elements of $\mathcal{C}$ as described above, cl cache-line messages from main memory to $M(t)$, and either cl or $\lceil \frac{\min(cs,ds)}{cls} \rceil$ cache-line messages from the unique elements of $\mathcal{C}$ to $M(t)$.

Finally, the read cache-lines may be evicted from the cache to make room for other cache-lines. This eviction results in cl protocol messages from $M(t)$ to the directory. The contributions to the bandwidth graph from each input edge are summarized in Table 5.1.

From	→	To	Protocol messages	cache-line messages
$M(t)$	→	Directory	$2cl$	-
Directory	→	Memory	cl	-
Directory	→	$M(t_p)$	$\lceil \frac{\min(ds,cs)}{cls} \rceil$	-
Directory	→	$M(t_s)$	cl	-
Memory	→	$M(t)$	-	cl
$M(t_p)$	→	$M(t)$	-	$\lceil \frac{\min(ds,cs)}{cls} \rceil$
$M(t_s)$	→	$M(t)$	-	cl

Table 5.1: The contributions to a bandwidth graph from an input edge. The contributions involving $M(t_s)$ are added for each IP core on which one or more of t's siblings reading α is mapped as long as t is not mapped to that IP core as well. The contributions involving $M(t_p)$ are conditional on none of t's siblings reading α or t being mapped to $M(t_p)$.

For each output edge e_o of t, the address range α is also divided by the cache-line size to get the potential number of misses, also labeled cl. As these misses require the cache-line to be retrieved, they contribute to the bandwidth graph in a similar manner to input edges: A protocol message is sent from $M(t)$ to the directory, which sends a protocol message to those locations that may contain the cache-line that is then sent from this location to $M(t)$. The difference to input edges is which locations may contain the cache-line. Due to the requirement that no race conditions exist, the tasks that may have accessed the currently valid version of the cache-line are the latest task (if any) of t's predecessors that has written α and those of its children that have read α. Denoting the latest writing predecessor t_w and those of its children that have read the address range T_r, only the caches belonging to the IP cores to which these tasks are mapped may hold a copy of the cache-line. It is a requirement that all tasks in T_r are predecessors to t, as otherwise a race condition would exist, as described by equation 5.1. As no other tasks can access α while t is writing, the amount of data that may be fetched from $M(t_w)$ and $M(t_r) : T_r \in M(t_r)$ is bounded by the cache size. As for input edges, the set $\mathcal{C}$ of unique IP cores whose caches may contain the cache-line and to which $M(t)$ is not mapped is determined.

$$\mathcal{C} = (\{M(t_w)\} \cup \{M(t_r) : t_r \in T_r\})\{M(t)\} \tag{5.3}$$

Additionally, the cache-line may be found in main memory.

In the analytical model, fetching a cache-line as part of an output edge thus contributes cl protocol messages from $M(t)$ to the directory, $\lceil \frac{\min(ds,cs)}{cls} \rceil$ protocol messages from the directory to each unique element in $\mathcal{C}$, cl protocol messages from the directory to the memory, cl cache-line messages from the memory to $M(t)$, and $\lceil \frac{\min(ds,cs)}{cls} \rceil$ cache-line messages from each unique element in $\mathcal{C}$ to $M(t)$.

When an IP core requests write access to a cache-line, the directory also needs to instruct those caches containing a copy of the cache-line to invalidate it. Thus, for the address range α, additional $\lceil \frac{\min(ds,cs)}{cls} \rceil$ protocol messages are sent from the directory to each unique element in $\mathcal{C}$.

The cache-lines containing α may also be evicted from the cache, which requires sending a protocol message to the directory for each cache-line, resulting in cl protocol messages from $M(t)$ to the directory. The written cache-lines eventually have to be written back to main memory. This write-back is triggered by the directory, which sends cl protocol messages from the directory to $M(t)$. For the actual write-back, cl cache-line messages are sent from $M(t)$ to main

memory.

The contributions to the bandwidth graph from an output edge are summarized in Table 5.2.

From	→	To	Protocol messages	cache-line messages
$M(t)$	→	Directory	$2cl$	-
$M(t)$	→	Memory	-	cl
Directory	→	$M(t)$	cl	-
Directory	→	Memory	cl	-
Directory	→	$M(t_w)$	$\lceil \frac{\min(ds,cs)}{cls} \rceil$	-
Directory	→	$M(t_r)$	$\lceil \frac{\min(ds,cs)}{cls} \rceil$	-
Memory	→	$M(t)$	-	cl
$M(t_w)$	→	$M(t)$	-	$\lceil \frac{\min(ds,cs)}{cls} \rceil$
$M(t_r)$	→	$M(t)$	-	$\lceil \frac{\min(ds,cs)}{cls} \rceil$

Table 5.2: The contributions to a bandwidth graph from an output edge. The contributions involving $M(t_r)$ are added for each IP core on which one or more of the tasks in T_r are mapped as long as t is not mapped to that IP core as well. Likewise, the contributions involving $M(t_W)$ are not added if $M(t) = M(t_w)$.

5.3 Simulator

This section describes the simulator used to find the expected bandwidth graph for a task graph mapped on a shared-memory system. This is done by simulating both the task graph's execution as well as the caches' behavior.

The simulator does not use a fixed time scale, but rather operates with memory accesses – whether cache hits or misses – as discrete time units: In each time unit or simulation cycle, each IP core is able to make one memory access including the handling of possible cache misses.

Each task is mapped to an IP core using the same mapping function as the analytical derivation, $M : T \to O$. Additionally, each task is given an execution time measured in simulation cycles. The execution semantics for the task graph have been chosen such that they more closely resemble execution in a shared-memory system. Specifically, communication is part of the tasks execution in the simulator: When a task starts executing, it reads its inputs sequentially, while at the end of its execution, it writes its outputs sequentially. Therefore,

a task's execution time has a lower bound given by the number of memory accesses it makes. The execution time for a task measured in simulation cycles is a static property of the task in the task graph.

Besides the mapping of tasks to IP cores, a schedule describing each task's start of execution is required. Here, as-soon-as-possible (ASAP) scheduling is used.

During simulation, the simulator keeps track of the contents of the caches. Each memory access thus either generates a hit or a miss, where misses trigger simulation of the cache-coherence protocol. By counting the number of messages and the amount of data communicated in the system during simulation, a bandwidth graph can be constructed.

5.4 Experiments

This section presents the task graphs and system configurations used for comparing the analytical derivation of the bandwidth graphs to the bandwidth graphs produced through simulation.

Synthetic task graphs have been used for the evaluation. Sixteen task graphs have been generated, four each with 16, 32, 128, and 1024 tasks. A system configuration consists of two parts: The number of IP cores used for executing tasks – i.e., excluding the directory and main memory – and the number of cache-lines in a cache, where the cache-line size is assumed constant at four 32-bit words, or 16 bytes. A protocol message takes up 8 bytes. Whenever a cache-line is transmitted, a protocol message is included, bringing the size of a message containing a cache-line to 24 bytes. Four different system configurations are generated by varying these two parameters over two values each: 16 and 64 IP cores, and cache sizes of 1 and 256 cache-lines. For each of the two system sizes (number of IP cores), five random mappings are made of each task graph to the IP cores. As stated previously, the same mapping is used for both the simulation and the analytical derivation of the bandwidth graph.

The synthetic task graphs represent an average case for evaluating by how much the analytical method overestimates the communication bandwidth. The worst case is represented by a task graph, where one task writes an address range that is read by many tasks. In this case, the analytical method is unable to determine in which order the sibling tasks read the given address range. Therefore, it adds bandwidth to the worst-case bandwidth graph corresponding to each task receiving the data from all of its siblings. This worst case is evaluated using task graphs with 4, 8, 16, and 32 tasks mapped to separate

IP cores, where one task writes an address range that is read by the remaining tasks.

Conversely, the best case is represented by a task graph with no two edges having the same values for α, corresponding to the situation where no two tasks have the same input data and no address range is used for multiple data exchanges, and where all tasks execute on a single IP core. However, this case is uninteresting as a uniprocessor system has no need for shared memory and cache-coherence. Therefore, the approximately best case is evaluated using four synthetically generated task graphs with 16, 32, 128, and 1024 tasks with a random mapping of tasks to both 16 and 64 IP cores with only a single cache-line in each cache.

Each execution of both the simulator and the analytical derivation is timed. The time is measured from just after the input files have been read to just before the output files are written in both cases.

5.5 Results

This section presents a comparison of the results of the analytical derivation of bandwidth graphs and simulation. As previously stated, the analytical derivation produces a worse-than-worst case bandwidth graph. The presented results show by how much the analytical method overestimates the total bandwidth compared to the bandwidth observed in the simulation.

The results from the synthetic task graphs show that different mappings have negligible influence on the accuracy of the analytical transformation. Two mappings result in different bandwidth graphs only when two tasks sharing data are mapped to the same IP core in one mapping but not in the other. In the case where the two tasks are mapped to the same IP core, the protocol messages from the directory to the IP core and the cache-line messages between the two IP cores are not transmitted. The interesting variables are therefore the task graph size, the number of IP cores, and the cache size.

Figure 5.2 shows the percentage difference in total bandwidth averaged over all four task graphs of each size between the analytical method and the simulation for systems with large caches (256 cache-lines). Formally, the reported result can be expressed using the notation for the bandwidth from definition 2.2 as

$$\frac{\sum_{TGs} \sum_{i \in O} \sum_{j \in O} (b_{i,j}^{ana} - b_{i,j}^{sim})}{|TG|} \tag{5.4}$$

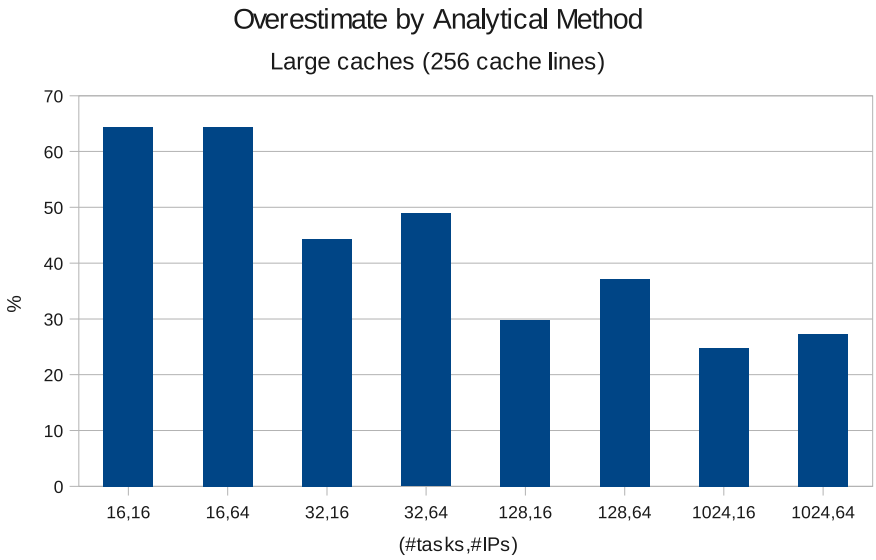


Figure 5.2: Overestimate by the analytical method for systems with large caches.

where superscript *ana* indicates the analytical method, and superscript *sim* indicates simulation.

The first observation is the decreasing difference between the analytical method and the simulation as the task graph size increases, i.e., the analytical method is more accurate with larger task graphs. This tendency is explained by the fact that smaller task graphs result in less evictions from the caches in simulation, as a larger portion of the tasks are the last ones to execute on their respective IP cores. For example, when 16 tasks execute on 64 IP cores, even with a completely random mapping, each task is likely to have its own IP core. Therefore, all evictions during simulation are caused by the tasks' data sets being larger than the cache, whereas with 1024 tasks, evictions are also caused by one task executing on an IP core on which another task previously executed. Therefore, the cache contains the previous task's data set, which needs to be evicted to allow the following task to load its data set.

The second observation is the difference caused by changing the number of IP cores while keeping the number of tasks constant. In this case, the analytical derivation is more accurate with fewer IP cores with the difference being more

pronounced with bigger task graphs. Again, the main contribution is a difference in the number of evictions: With fewer IP cores, more of the analytically predicted evictions actually occur, while with more IP cores, the larger amount of cache memory in the system results in a larger portion of main memory being kept in caches leading to fewer replacements and thereby evictions.

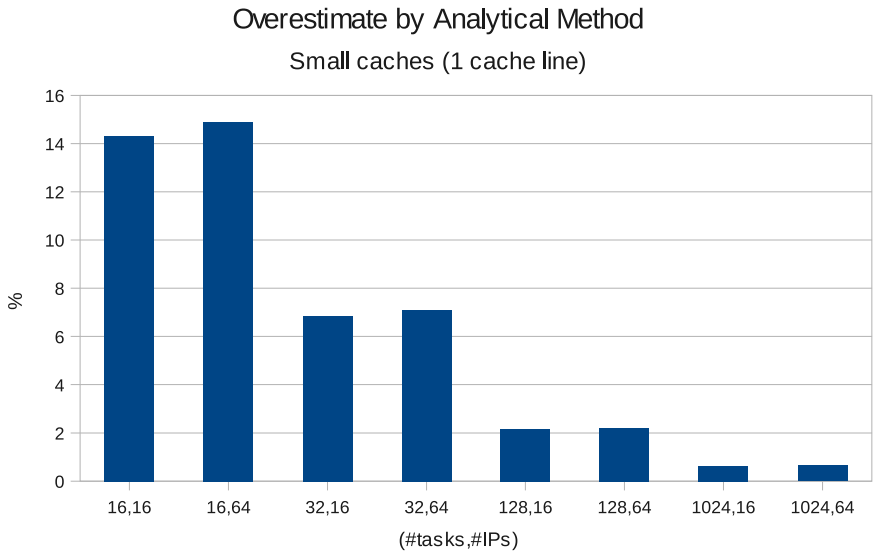


Figure 5.3: Overestimate by the analytical method for systems with small caches.

Figure 5.3 shows the same percentage difference in the bandwidth determined through simulation and by the analytical method for smaller caches. This graph shows the same tendencies as the one for larger caches, also caused by the analytical method overestimating the number of evictions. The difference is in the results ranging from approximately 1% to 15% compared to 25% to 65% for larger caches. This difference is explained by the inherent unpredictability of systems with caches: As memory contents are stored in multiple places, it is almost impossible to predict from which of these places the memory contents actually are fetched during execution. The analytical method therefore adds bandwidth corresponding to the data being fetched from *all* these places in order to find the worst-case bandwidth communicated between individual pairs

of IP cores, while in reality, it is only fetched from one of them. Thus, reducing the cache size has the effect of also reducing the uncertainty in the predictions at the price of significantly reduced performance.

The above results concerned synthetic task graphs that represent problems somewhere between the best and worst cases for the analytical method. The following concerns approximations of these two extremes.

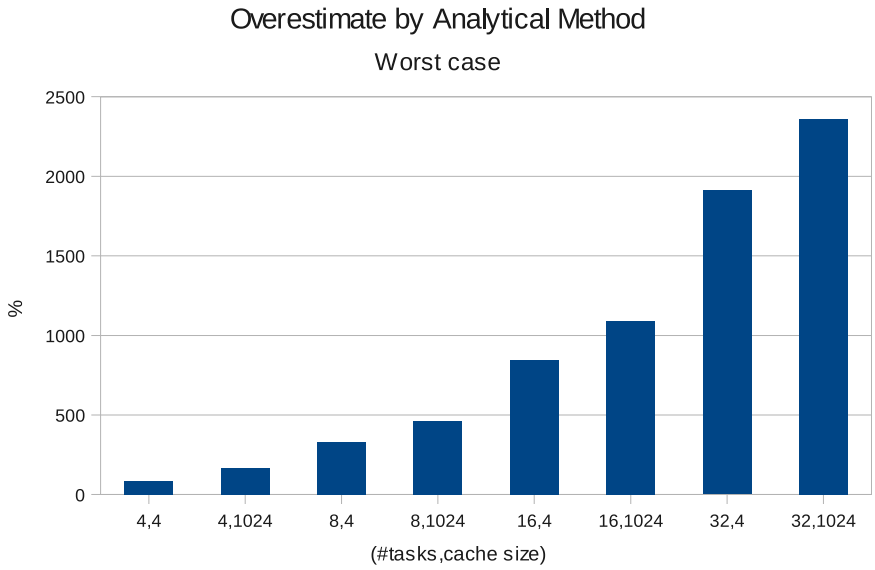


Figure 5.4: Worst-case overestimate by the analytical method.

The worst case for the analytical method is many tasks that are mapped to different IP cores and read the same input with no dependencies between them, as it is impossible to analytically determine the sequence in which the tasks read the input. The analytical method overestimates the total bandwidth in this case by up to 2500%, as shown in Figure 5.4. The degree of the overestimate increases with the number of siblings reading the same address range in the task graph. This is as expected, as the analytical method is unable to determine the actual flow of data between caches, and therefore assumes data is copied from every cache to every other cache in the system.

The best case for the analytical method has no two tasks reading from or

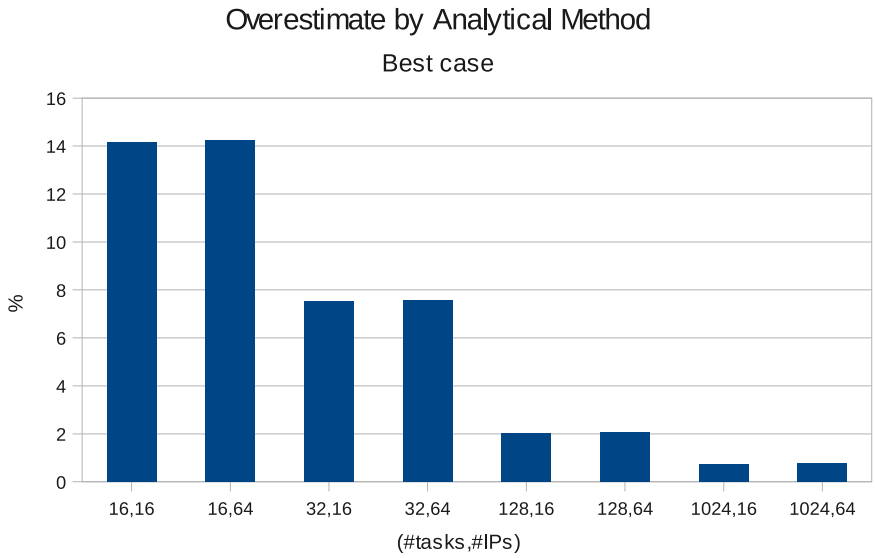


Figure 5.5: Best-case overestimate by the analytical method.

writing to the same addresses. Figure 5.5 shows the overestimate in this case. Comparing to Figure 5.3, very little difference is seen between the best case and the average case with small caches. This indicates that the impact of a few tasks reading from or writing to the same address range has very little impact.

It is impossible to determine the analytical method's accuracy for real applications through the results presented here. Depending on the application's pattern of sharing data, the overestimate may be anything between the best case of approximately 1% and the worst case, which depends on the number of tasks sharing the same input data.

The average execution times of the analytical method and the simulation for the different problem sizes (number of IP cores and number of tasks) are shown in Figure 5.6. In all cases, the execution time of the analytical method is less than 5% of that of the simulation, and in most cases, it is closer to 1%. Increasing the number of tasks and the number of IP cores both lead to an increase in execution time.

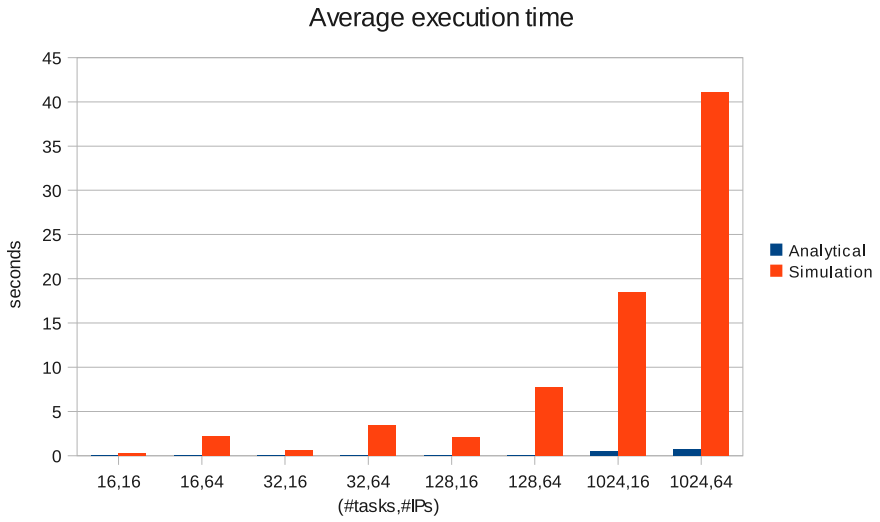


Figure 5.6: Comparison of the execution time of the analytical method and simulation.

5.6 Summary

This chapter presented an approach to analytically derive a bandwidth graph from a task graph in a shared-memory system by considering the communication caused by the cache-coherence protocol.

The analytical method was evaluated by comparing the bandwidth graphs it produced to those generated through simulations of the application and the shared-memory system including the cache-coherence protocol. Varying the number of tasks, the number of IP cores, and the cache sizes all impacted the analytical method's accuracy for average-case task graphs: The accuracy increases with the number of tasks, while it decreases as the number of IP cores is increased, both primarily due to fewer evictions in the simulations. Having a system with very small caches significantly improves the accuracy of the analytical method, as most of the communication occurs through main memory, while larger caches lead to overestimates of the total bandwidth of 25% to 65%.

For approximate best- and worst-case task graphs, the analytical method

overestimates the total bandwidth by up to 15% and 2500% respectively for the investigated task graph sizes. The worst case overestimate depends on the number of tasks in the task graph. The overestimate for a real application will be somewhere between these extremes.

Chapter 6

Topology Synthesis in MANGO

This chapter presents methods for synthesizing topologies in the MANGO NoC. The objective of the synthesis is to find a topology that minimizes the execution period of a periodic application. First, in section 6.1 an introduction to MANGO is given, in section 6.2 the problem formulation is presented, and the modelling of MANGO is discussed in section 6.3. Then, in section 6.4, the optimization approaches are presented, and in section 6.5 the problems used to evaluate the heuristics are described. Finally, the results are presented in section 6.6, and conclusions are given in section 6.7. The MANGO NoC was developed by Tobias Bjerregaard [7]. This chapter is based on the work presented in [73].

6.1 The MANGO Network-on-Chip

MANGO [7] is a NoC that is designed and implemented using clockless or asynchronous design principles [67]. The MANGO router architecture [10] provides both connection-oriented guaranteed service and connection-less best effort communication. For guaranteed service connections, an absolute guarantee is given on the worst-case end-to-end latency of individual flits, provided the network adapters adhere to a maximum rate for injecting flits. Virtual circuit-switching is used to form guaranteed service connections, while best effort communication uses packet-switching with source routing and credit-based flow control. For connecting IP cores to the NoC, network adapters with OCP [64] interfaces are

provided [9]. Figure 6.1 shows the main components of the MANGO router.

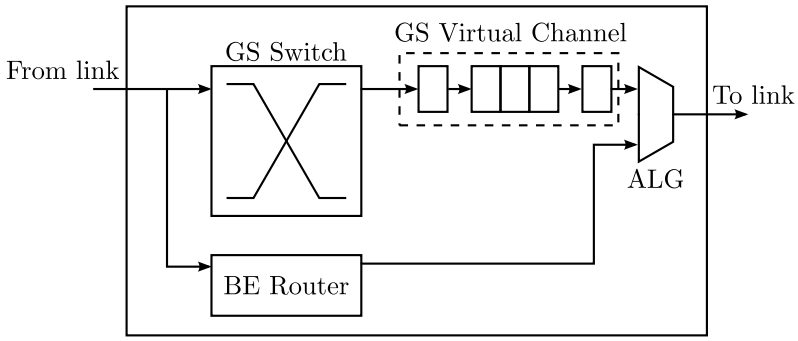


Figure 6.1: The main components of the MANGO router. Only one input and one output port is shown, as well as only one guaranteed service (GS) virtual channel in the output port.

Providing absolute latency guarantees in an environment with no global timing reference is accomplished by utilizing the virtual channel flow control and a special scheduling discipline called Asynchronous Latency Guarantees (ALG) [11]. This scheduling discipline is used to arbitrate individual virtual channels' flits' access to each link. The components involved are sketched in Figure 6.2.

Flow control is provided for each virtual channel using a sharebox and an unsharebox, which together ensure that at most one flit is in flight between two succeeding virtual channel buffers on a connection. Furthermore, the unsharebox includes a latch which guarantees that a flit will not block the switching structure in the router ensuring that the only dependencies between flits on different connections are those that occur in arbitration for access to links. When a flit, f_i, leaves a virtual channel buffer, it passes through the sharebox which prevents the succeeding flit, f_{i+1}, from leaving the buffer. Only when f_i has passed through the unsharebox in front of the succeeding virtual channel buffer, the sharebox is unlocked allowing f_{i+1} to leave. This mechanism prevents flits on two connections sharing one or more links from blocking each other, thereby providing the basis for giving latency guarantees on individual connections.

The link arbitration consists of two parts: A static priority arbiter (SPQ) and an admission control in front of the arbiter. The arbiter assigns each virtual channel a static priority for accessing the link, while the admission control is

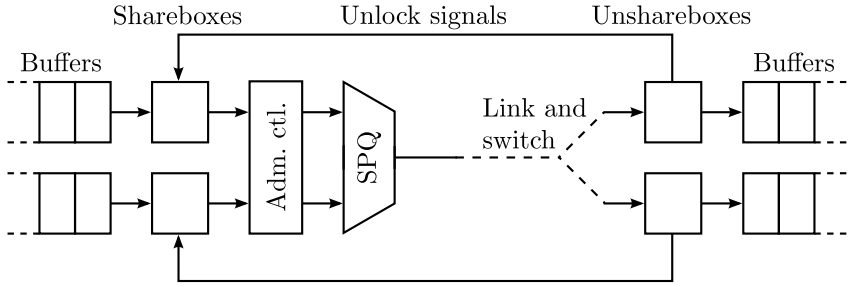


Figure 6.2: Implementation of guaranteed services in MANGO. The flow control mechanism consisting of shareboxes and unshareboxes prevents a virtual channel from sending a flit that would block the link and switching structures due to insufficient buffer capacity.

used to prevent starvation of virtual channels with low priorities. This is done by allowing a flit on a high priority virtual channel to only preempt a single flit on each of the lower prioritized virtual channels, e.g., if virtual channels 0, 2, and 5 have flits ready for transmission, the highest prioritized (0) is granted access to the link, but the next flit on virtual channel 0 is blocked by the admission control until one flit from each of virtual channels 2 and 5 has been granted access to the link. The best effort virtual channels are also connected to the link through the arbiter, thereby avoiding starvation of the best effort traffic.

Combining arbitration, admission control, and flow control allows maximum latency guarantees to be given, provided the users of the NoC (the network adapters) adhere to some constraints. Specifically, the user of a high priority connection is guaranteed fast transmission of flits, as long as the temporal distance between succeeding flits is sufficient to guarantee that the admission control does not block the later flits.

The combination of arbitration and admission control guarantees an equal share of bandwidth to all virtual channels in all cases. However, if the high priority connections seldom send flits, these flits are allowed to overtake flits on lower prioritized connections in each arbiter along their route. Alternatively, a connection may make full use of its bandwidth guarantee, exchanging the guarantee on individual flits' latency for bandwidth. However, a maximum latency can also be determined in this case, as the admission control at most delays a flit until the time at which it would have arrived had it adhered to the given constraint.

6.2 Problem Formulation

This chapter considers synthesizing application-specific topologies in MANGO. Applications are assumed to have a periodically recurring communication pattern, which is modelled using a variation of a bandwidth graph. Instead of associating the weights on the bandwidth graph's edges with an amount of data per second, they are taken as indicating the amount of data per execution period of the application. The optimization objective is to minimize this period, i.e., finding the topology that handles a given communication load the fastest.

Considering the edge weights as the number of flits per periodic execution of an application allows a designer to use the approach presented here to determine the obtainable period and compare that one to the required period. The difference between the actual and required periods may then be utilized to either reduce the speed of the system, saving power, or adding features to the application, if possible.

A formal definition of the optimization problem is as follows:

Def. 6.1. A periodic bandwidth graph $PBG = (O, C)$ is a directed graph modelling the communication in a periodically executing application. Each vertex $o \in O$ represents an IP core, and each directed edge $c_{i,j} \in C$ represents a connection from o_i to o_j. Each connection $c_{i,j} \in C$ has a weight $b_{i,j}$ which indicates the number of flits transmitted on $c_{i,j}$ in each periodic execution of the application.

Def. 6.2. A topology graph $TG = (V, L)$ is an undirected graph, where each vertex $v \in V$ represents a network node consisting of an IP core and a router, and each edge $l_{i,j} \in L$ represents a bidirectional link between v_i and v_j.

Def. 6.3. A mapping $M : O \rightarrow N$ maps each vertex in O to a vertex in V. M is required to be bijective, i.e., a given vertex in O maps to one and only one vertex in V, and each vertex in V has a vertex in O mapped to it.

The optimization problem can now be defined as: Given PBG, V, and M, synthesize L such that the period of the communication in the application specified by PBG is minimized, subject to

$$\forall x, y \in V \exists \langle v_x \dots v_y \rangle \tag{6.1}$$

$$\nexists v \in V : \mathit{degree}(v) > 4 \tag{6.2}$$

The constraint in equation 6.1 states that the topology must be connected, i.e., a path must exist between any two network nodes.

The constraint in equation 6.2 specifies that the degree of any network node may not be greater than four, i.e., the largest routers allowed are five-port routers including the port used by the IP core. As the optimization objective is simply to minimize latency, the optimal topology will is a fully connected graph, i.e., the topology will have point-to-point connections between all pairs of network nodes. Such a topology is undesirable when factors such as area, power consumption, and feasibility of layout are taken into consideration, and constraining the maximum degree of network nodes prevents such topologies from occurring.

6.3 Modelling Latency in MANGO

Given the above optimization problem, a model of MANGO is required to capture the latency of the execution of one period of the modelled application when using guaranteed service connections. For this purpose, the latency guarantee that can be given for individual flits is not interesting, as the interesting measure is the latency of all flits on each connection, i.e., it is the bandwidth rather than the latency guarantees provided by MANGO that is considered here. The model should thus describe the latency of transmitting a number of flits on a connection under the worst-case assumption that flits are blocked for the maximum possible time in the admission control. In this case, the admission control and arbitration essentially have the functionality of a round-robin arbiter which is therefore used in the following analysis.

In order to determine the latency of an application modelled as a periodic bandwidth graph, this graph, a topology graph, the mapping of IP cores to network nodes, and routes for each connection in the periodic bandwidth graph are required.

For a connection c, the link along the route servicing c that is shared by the highest number of other routes is found. The number of routes sharing this link is denoted by $n_{bottleneck,c}$ as this link constitutes the bottleneck for the bandwidth guarantee given to the route. Further, given an injection rate, ρ_{inj}, which is the maximum rate at which network adapters inject flits in the network, and a link rate, ρ_{link}, which is the maximum rate at which flits may be transmitted on a link, the rate at which flits are actually injected on c in a steady state of the network is determined as

$$\rho_{max,c} = \min\left(\rho_{inj}, \frac{\rho_{link}}{n_{bottleneck,c}}\right) \tag{6.3}$$

i.e., as the minimum of the rate with which the network adapter attempts to inject flits with and the rate with which flits are transmitted through the network. It is assumed that flits may be ejected from the network at a greater rate than that with which they are injected.

The worst-case latency for a flit to be transmitted through the network has two contributions: The best-case latency (i.e., the latency assuming no stalls in the arbitration or admission control) and the amount of time stalled in the arbitration and admission control. The first contribution is determined by the route's length or hop count, $n_{hops,c}$, multiplied by the latency of a hop, t_{hop}, which includes the fall-through latency of the admission control, arbiter, link, switching structures, the flow control mechanism, and the virtual channel buffer. Note that one or more of these components may be pipelined, such that

$$t_{hop} \geq \frac{1}{\rho_{link}}$$

The second contribution is determined by the number of other routes with which each link along the route is shared and the rate with which flits are transmitted on each link. The number of routes with which a link is shared determines the worst-case number of times a flit is not granted access to the link in arbitration, while ρ_{link} determines the delay incurred by each stall in arbitration. Using $n_{shared,i}$ to denote the number of routes sharing link i and $n_{shared,c} = \sum_i (n_{shared,i} - 1)$ (the subtraction by one comes from the fact that a route does not share a link with itself, i.e., it does not compete with itself in arbitration), the latency of an individual flit on the connection c can thus be described as

$$t_{flit,c} = n_{hops,c} \times t_{hop} + \frac{n_{shared,c}}{\rho_{link}} \tag{6.4}$$

assuming no stalls due to flow control. While this assumption does not reflect the actual behavior of MANGO as implemented in [7] under certain circumstances, a simple change to credit-based flow control for the virtual channels used for guaranteed service will bring the model and implementation in agreement with each other. Specifically, if $\rho_{max,c} < \frac{1}{t_{hop}}$, a connection in the model has a higher rate of transmitting flits than is achievable in reality. However, as the proposed change is simple to implement and improves the best-case bandwidth of individual guaranteed service connections, i.e., the bandwidth a connection can achieve when it is the only connection using a given link, this is what has been chosen for the model. Alternatively, the model can be made to reflect the

current implementation by modifying equation 6.3 to

$$\rho_{max,c} = \min\left(\rho_{inj}, \frac{\rho_{link}}{n_{bottleneck,c}}, \frac{1}{t_{hop}}\right)$$

although this still assumes that stalls due to insufficient buffer capacity causing the flow control to block flits that would otherwise win arbitration for the link do not occur.

So far, this section has described how to find the latency of individual flits. The objective of interest for the optimization problem is the total latency of transmitting all flits in one period of the application. First, the latency of transmitting all flits, $n_{flits,c}$, on a given connection, c, is

$$t_c = \frac{n_{flits,c}}{\rho_{max,c}} + t_{flit,c} \tag{6.5}$$

This equation reflects the pipelined nature of transmitting flits: A flit is injected in the network every $\frac{1}{\rho_{max,c}}$ second until all flits have been injected, at which point it takes $t_{flit,c}$ seconds for the last flit to reach its destination. The second term in equation 6.5 may be omitted if the application periods are allowed to overlap. However, in most cases, the first term will dominate the second term.

Given the latency of individual connections, the maximum period of the application can be determined as

$$T_{app} = \max_{c \in C}(t_c) \tag{6.6}$$

which is the measure of interest. The connection with the highest t_c, i.e., the connection that determines the period of the application, is referred to as the critical connection, $c_{critical}$, due to its resemblance with the critical path in combinatorial, digital logic.

In most cases, the model described above finds a worse than worst-case period of the application described by the *PBG*: In general, different connections have a different amount of flits, i.e., $b_{x,y} \neq b_{u,v}$, which is not considered when calculating t_c for each of the connections $c_{x,y}$ and $c_{u,v}$. The model assumes that all flits incur the maximum waiting time in the admission control and arbitration even if one of the connections sharing a link only transmits a single flit in each period of the application. In this extreme case, at most one flit on each of the other connections sharing a link with this single-flit connection would experience the maximum delay in the admission control. For the remaining flits, $n_{shared,c}$ in equation 6.4 should be decreased by one in order to reflect the actual latencies

of individual flits, while $\rho_{max,c}$ in equation 6.3 could potentially be increased due to a decrease in $n_{bottleneck,c}$. However, capturing all these interdependencies between different connections is left for future work.

One aspect that is necessary to evaluate the quality of a topology, but which is not included in the optimization at this time, is routing. The route for a connection $c_{i,j}$ is found using a breadth-first search in the topology graph with $M(i)$ as the source vertex. This type of search guarantees routes with minimal hop counts to be found. However, in some cases, non-minimal routes may lead to a lower latency for the application: By rerouting certain connections on longer routes, $n_{bottleneck}$ of the critical connection, $c_{critical}$, may be lowered, thereby shortening $t_{c_{critical}}$ and T_{app}. However, which connection is the critical one may also be changed by this rerouting. Conducting experiments involving routing beyond breadth-first search is left for future work.

The values used in the experiments for the different model parameters are as follows.

$$\rho_{inj} = \frac{1}{5}\frac{\text{flits}}{\text{ns}} \tag{6.7}$$

$$\rho_{link} = \frac{1}{3}\frac{\text{flits}}{\text{ns}} \tag{6.8}$$

$$t_{hop} = 6\text{ns} \tag{6.9}$$

6.4 Optimization Approaches

This section describes the heuristics that have been implemented to solve the optimization problem defined above. The heuristics are based on the simulated annealing and tabu search meta-heuristics described in chapter 3. Both of these meta-heuristics require an initial solution, a means of handling invalid solutions (solutions that violate the constraints in equations 6.1 and 6.2), and a definition of moves and neighborhoods.

For the initial solution, a random, valid solution is used to avoid artificially restricting the search, as discussed in chapter 3.

A solution is invalid, if it violates either or both of the constraints in equations 6.1 and 6.2. A breadth-first search is used to determine whether a solution is connected: Starting the search at a given node, v_{start}, all nodes that are reachable from v_{start} are found. If all nodes are reachable, the solution satisfies the constraint. If unreachable nodes exist, two sets are formed: One with the reachable nodes and one with the unreachable nodes. A random node is selected

from each set, and a link is added between these two nodes. This procedure is repeated until the solution is connected.

If the constraint on the maximum degree of a node is violated, random links are removed from the node until the constraint is satisfied. Removing links may lead to the solution becoming unconnected, violating the first constraint, while making the solution connected may increase the degree of a node beyond the maximum allowed, violating the second constraint. Thus, it is possible that making a solution satisfy these two constraints leads to infinite cycling. However, due to the non-deterministic nature of the corrections made to the solutions, the number of cycles is limited in practice.

The n-bit neighborhood of a solution is defined by those solutions that differ in the presence of n links in the topology. The exact number of links is considered a parameter of the heuristics. A move is thus defined as toggling the presence of n links, although making a solution satisfy the given constraints may lead to a different number of links actually being toggled.

6.4.1 Simulated Annealing

One of the considered heuristics is simulated annealing. The heuristic itself is implemented as described in section 3.3, using the above choices for initial solution, neighborhood, and moves, and the model described in section 6.3 for the evaluation function. The heuristic's parameters are thus the starting temperature, τ_{start}, the cooling rate, α, and the number of links to toggle in each move, n_{nhood}.

6.4.2 Tabu Search

The other considered meta-heuristic is tabu search. Five variations of the tabu search meta-heuristic have been considered: Plain tabu search without any long term memory and two variations of adding long term memory to the heuristic with two different diversifications being triggered by the long term memory. Each of these is described in the following sections.

Base Tabu Search

The initial solution, move, and neighborhood are defined as above. As opposed to the simulated annealing, the neighborhood is fixed at 1-bit, i.e., the presence of only a single link is toggled in a move. This is done to keep the size of the

neighborhood reasonable for doing an exhaustive search. In a problem with N IP cores, there are

$$\frac{N(N-1)}{2}$$

links whose presence may be toggled. With a n_{nhood}-bit neighborhood, the number of neighbors to a solution is described by the binomial coefficient

$$\binom{\frac{N(N-1)}{2}}{n_{nhood}}$$

Given a system with 256 IP cores, a 1-bit neighborhood thus results in any given solution having 32 640 neighbors, while a 2-bit neighborhood results in 532 668 480 neighbors. Although some (even many) of these neighbors most likely are invalid solutions, performing an exhaustive search of that many solutions is prohibitively slow assuming a requirement of at least a few iterations of the tabu search per second. Additionally, the neighborhood search is terminated after finding the first solution with a lower latency than the current solution.

The base tabu search has a single parameter: The length of the tabu list, $l_{tabulist}$.

Long Term Memory

As mentioned above, two variations of adding long term memory to the heuristic have been investigated. Note that the description "long term" is used even though the results presented later show that, in some cases, this added memory produces better solutions if the long term memory triggers a diversification earlier rather than later.

Both variations are built over the same theme: A counter that keeps track of the number of iterations without improvement in the best found solution. The limit of the counter (the number of iterations without improvement that triggers a diversification) is designated by n_{noimp}.

The first variation (V1) triggers a diversification after n_{noimp} iterations without improvement since the last diversification (or the start of the search), regardless of whether any improvement has been made since then, i.e., the counter is incremented in each iteration without improvement in the best found solution and is reset only when a diversification is triggered (when the counter equals n_{noimp}).

The second variation (V2) is very similar to the first one, but differs in the criteria for resetting the counter: The counter is both reset when it reaches

n_{noimp} (and triggers a diversification), and when the search encounters a solution with lower latency than the currently best found solution, i.e., when the best found solution is updated.

In the first variation, the long term memory is associated with the particular segment of the search space that the latest diversification moved the search to. This is also true in the second variation, but resetting the counter upon finding a new best solution extends the search of the particular search space segment, trading off the number of segments explored to the depth with which each segment is explored. Both variations add one parameter to the heuristics using them: The number of cycles without improvement that triggers a diversification, n_{noimp}.

Diversifications

Both of the considered diversifications make a change in the critical connection, $c_{critical}$, in an attempt to reduce its latency, $t_{c_{critical}}$, and thereby the period of the application, T_{app}.

The first diversification (D_{change}) forces a change in the route servicing the critical connection by removing all links along it and adding them to the tabu list. In this way, the search is moved to a different part of the search space and forced in a different direction, as the removed links are disallowed for as many iterations as the length of the tabu list.

The second diversification (D_{short}) changes the critical connection's route by making it a single hop, i.e., given $c_{critical} = c_{x,y}$, the link $l_{M(x),M(y)}$ is inserted. If adding this link causes either node's degree to violate the constraint in equation 6.2, the link that was previously part of the critical route is removed.

It can be argued whether or not these changes in a solution should be classified as diversifications: The changes affect only a small part of the solution, producing solutions that could be reached in only a handful of regular moves in most cases, which would normally not fall in the diversification category. However, an analysis of the characteristics of the evaluation function reveals why they are considered diversifications here, why they are necessary, and why they are expected to be efficient:

Given that the evaluation function reports the latency of the critical connection (which is dominated by n_{flits}/ρ_{max}), only some links' presence in the topology has an impact on the evaluation function. These links are:

1. The links that are used by the route servicing the critical connection. Removing one of these links may change $n_{bottleneck,c_{critical}}$ as the contributing routes are rerouted on different links.

2. The links that are included in the topology and are used by the routes sharing the critical connection's bottleneck link, i.e., the links that are used by routes that contribute to $n_{bottleneck,c_{critical}}$. By removing one or more of these links, the routes using them are rerouted, which most likely impacts $n_{bottleneck,c_{critical}}$.
3. The links that are *not* included in the topology, but when included would be used by either the critical connection or the routes contributing to $n_{bottleneck,c_{critical}}$.
4. The set of links, excluding the links in the three above sets, whose presence or absence, if toggled, would change which connection is the critical one.

From these four sets of links, the first three may either improve or worsen the critical connection's latency, while the links in the last set can only lead to a new critical connection with a higher latency than the existing one. Given the neighborhood exploration in tabu search, toggling the presence of a link from the first three sets can only be chosen as the best move to make, if doing so improves the critical connection's latency, or if no non-tabu links improves (or maintains) the critical connection's latency, and this is the link that worsens it the least. A link from the fourth set may also only be chosen in this second case.

However, there is a fifth set of links that contains all the remaining links. These links' presence can be toggled with no impact, positive or negative, on the critical connection's latency. Thus, all links in this fifth set maintain the result of the evaluation function, which has the effect that the tabu search prefers toggling a link from this set to a link that worsens the critical connection's latency. Therefore, considering the evaluation function as a landscape, local minima are not points in the landscape, but plateaus surrounded by crater-like slopes. Even if the tabu list pushes the search one step up the slope, the search will run along the contour lines around the plateau instead of moving further up the slope and over the edge onto the neighboring plateau or local minima. The diversifications however have the effect of moving the search from one plateau to another, as they consider the critical connection and force a change in its route.

6.5 Experiments

For evaluating the six heuristics (simulated annealing, base tabu search, and tabu search extended with all combinations of long term memory and diversifi-

cations), three synthetically generated periodic bandwidth graphs are used for tuning the heuristics' parameters, while six others are used for testing the found parameters, i.e., for comparing the heuristics.

The nine synthetic periodic bandwidth graphs are varied over three graph sizes (number of IP cores) and three traffic patterns. The graph sizes are 16, 64, and 256 IP cores, while the traffic patterns are:

Toroidal communication (TC): Assuming a bi-directional torus topology, the number of flits transmitted on the connection $c_{i,j}$ is

$$b_{i,j} = \begin{cases} r_{\alpha,\beta} & o_i \text{ and } o_j \text{ neighbors} \quad (6.10) \\ 0 & \text{otherwise} \quad (6.11) \end{cases}$$

where $r_{\alpha,\beta}$ is a uniformly distributed random variable in the interval $]\alpha;\beta[$.

Random (R): Given a probability p, the number of flits transmitted on the connection $c_{i,j}$ is

$$b_{i,j} = \begin{cases} r_{\alpha,\beta} & r_{0,1} < p \quad (6.12) \\ 0 & \text{otherwise} \quad (6.13) \end{cases}$$

This produces a completely random traffic pattern.

Hybrid (H): This traffic pattern is a random traffic pattern with a tendency towards communication between toroidal neighbors. Given a probability p, the number of flits is given by

$$b_{i,j} = \begin{cases} 2r_{\alpha,\beta} & r_{0,1} < \frac{3}{2}p \text{ and } o_i \text{ and } o_j \text{ neighbors} \quad (6.14) \\ r_{\alpha,\beta} & r_{0,1} < \frac{p}{2} \quad (6.15) \\ 0 & \text{otherwise} \quad (6.16) \end{cases}$$

These nine *PBG*s (three traffic patterns and three graph sizes) are generated once and used throughout all executions of the heuristics. Each *PBG* is referred to using its abbreviation (indicated in parenthesis in the above description) and the number of IP cores, e.g., H256 for the hybrid traffic pattern with 256 IP cores.

For both tuning and testing, fifteen repetitions are made of each run to minimize the possible interference from a particularly good or bad initial solution. Each run is allocated 60 seconds of CPU time (the heuristic terminates in the first iteration finishing after the 60 second mark) on a Sun Fire V440 with four UltraSPARC III CPUs running at 1062 MHz and with 1 MB L2 cache.

τ_{start}	α	n_{nhood}
$100, 200, \mathbf{300}, 400$	$0.97, \mathbf{0.98}, 0.99, 0.999$	$1, \mathbf{2}, 3$

Table 6.1: The parameter values explored during parameter tuning for simulated annealing.

Variation	$l_{tabulist}$	n_{noimp}
Base tabu search	$3, 5, 10, 15, \mathbf{20}, 25, 30$	-
V1, D_{change}	$3, 5, 10, 15, 20, 25, \mathbf{30}, 35$	$1, 3, 5, 10, 15, 20, \mathbf{25}, 30$
V1, D_{short}	$5, 10, 15, \mathbf{20}, 25, 30, 35$	$\mathbf{1}, 3, 5, 10, 15, 20$
V2, D_{change}	$5, \mathbf{10}, 15$	$10, \mathbf{15}, 20$
V2, D_{short}	$5, \mathbf{10}, 15$	$5, \mathbf{10}, 15, 20$

Table 6.2: The parameter values explored during parameter tuning for the tabu search variations. As the base tabu search has no long term memory, n_{noimp} is not a parameter for this variation.

6.6 Results

This section presents the results of the parameter tuning and testing of the heuristics. The three *PBG*s used for parameter tuning are TC256, R64, and H16, yielding a representative selection of both traffic patterns and problem sizes. The remaining *PBG*s are used for parameter testing.

6.6.1 Parameter Tuning

For the parameter tuning, a range of parameters has been selected for each heuristic, and the quality of the solutions produced by the heuristics with all combinations of parameter values has been determined. As described in chapter 3, the parameter values' range is iteratively extended until the parameter values yielding the best solutions are no longer on the edge of the range.

Tables 6.1 and 6.2 show the parameters and the ranges of their values for each heuristic with the best parameter values highlighted. It is interesting to note that for both (V1, D_{change}) and (V1, D_{short}), n_{noimp} is less than $l_{tabulist}$, i.e., the best option is to have the "long term" memory work over a smaller number of iterations than the tabu list, which is supposed to be an instance of "short term" memory. However, for (V2, D_{change}) and (V2, D_{short}), n_{noimp} is greater than or equal to $l_{tabulist}$. As V2 inherently has more iterations between diversifications

than V1 (due to resetting the counter more often), this difference in n_{noimp} can not be attributed to a desirable frequency with which diversifications should occur. Indeed, no explanation has been found for this difference: For now, it must be accepted that these are the best parameter values found by empirical means.

6.6.2 Parameter Testing

The results of the parameter testing are shown in Figures 6.3, 6.4, and 6.5. For each heuristic and problem, marks show the percentage gap of each of the fifteen repetitions, while a box shows the interval $[\mu-\sigma; \mu+\sigma]$. Including the individual repetitions makes it possible to determine whether any statistical outliers are present. SA indicates simulated annealing, BTS the base tabu search, V1C (V1, D_{change}), etc.

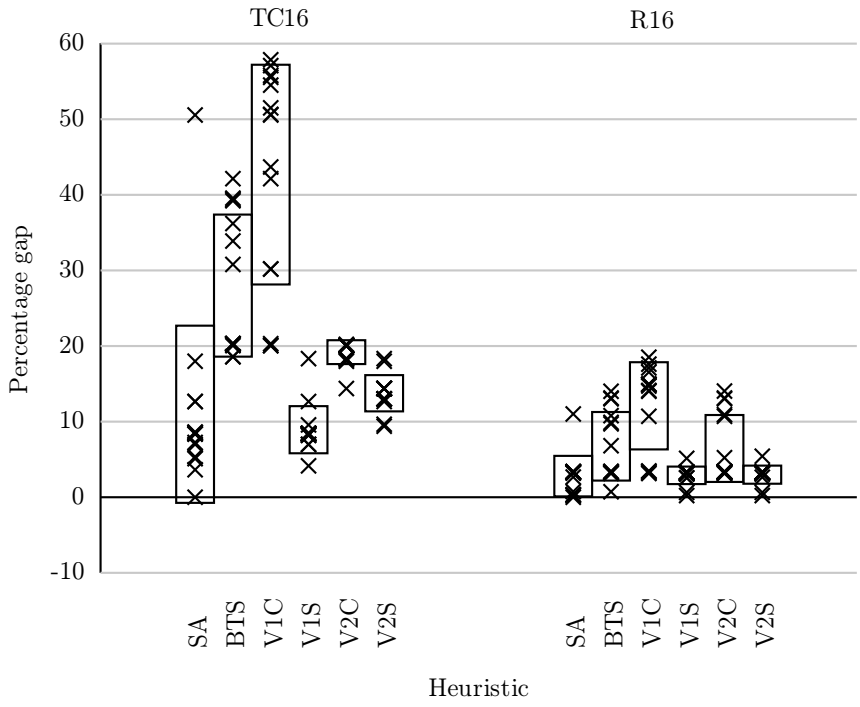


Figure 6.3: Results for *PBG*s with 16 IP cores.

For the smaller (16 IP cores) *PBG*s, the results are shown in Figure 6.3. In TC16, and to a certain degree in R16, simulated annealing has a single outlier, skewing the box indicating the mean and standard deviation. However, the remaining solutions produced by simulated annealing are all within 20% of the best found solution, which is also the case for V1S, V2C, and V2S. On the other hand, both BTS and V1C perform worse than the other heuristics with only a few solutions less than 20% from the best one for BTS and none for V1C. It is interesting to note that BTS performs better than V1C, i.e., that diversifying the search in this way is worse than performing the search without any diversification. Considering the diversifications, for both V1 and V2, D_{short} produces better results than D_{change}. This is as expected, as the changes made to the solution by D_{short} directly cause a reduction in the critical connection's latency, while the changes made by D_{change} force a change in the route servicing the critical connection, but not necessarily a change that reduces its latency. The relative performance of the heuristics on R16 is similar to that for TC16, except for V2C which produces solutions that are more spread out.

Overall, for the small problems, V1S produces the best solutions followed by V2S and SA that both are reasonably close to V1S, despite SA's outliers. After these, V2C, BTS, and V1C produce solutions that, while not exceptionally good, are not exceptionally bad either.

For the medium sized (64 IP cores) *PBG*s, the heuristics' relative performance shown in Figure 6.4 differs significantly from that for the smaller *PBG*s. First, BTS's performance is in line with, and in some cases better than, that of the other tabu search variations. Also, for V1, the performance of D_{change} is better than that of D_{short}, while for V2, the two diversifications' performances are similar to each other. As stated above, D_{short} is expected to produce better results than D_{change}, as the changes made are more likely to reduce the critical connection's latency. One possible explanation for this not being the case for these applications is the same explanation for why choosing an expectedly good initial solution for the search can lead to suboptimal performance of the heuristic: The good solution may be close to a locally optimal solution, but far from the globally optimal one.

Considering SA, it is seen that it performs better than the tabu search variations. The main reason for this is the difference in the amount of calculations needed in each iteration of the heuristics: The 1-bit neighborhood used in the tabu search variations contains 2016 solutions, each of which is checked whether or not it is a valid solution and, if so, is evaluated. On the available hardware, some iterations last more than a second, which severely limits the number of iterations of the search in the allocated time.

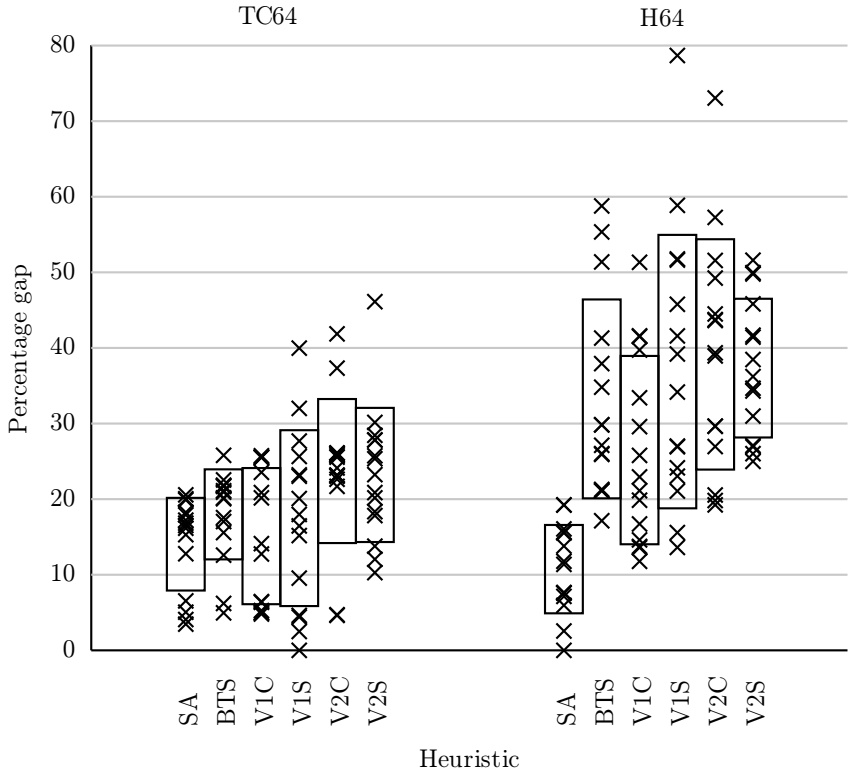


Figure 6.4: Results for *PBG*s with 64 IP cores.

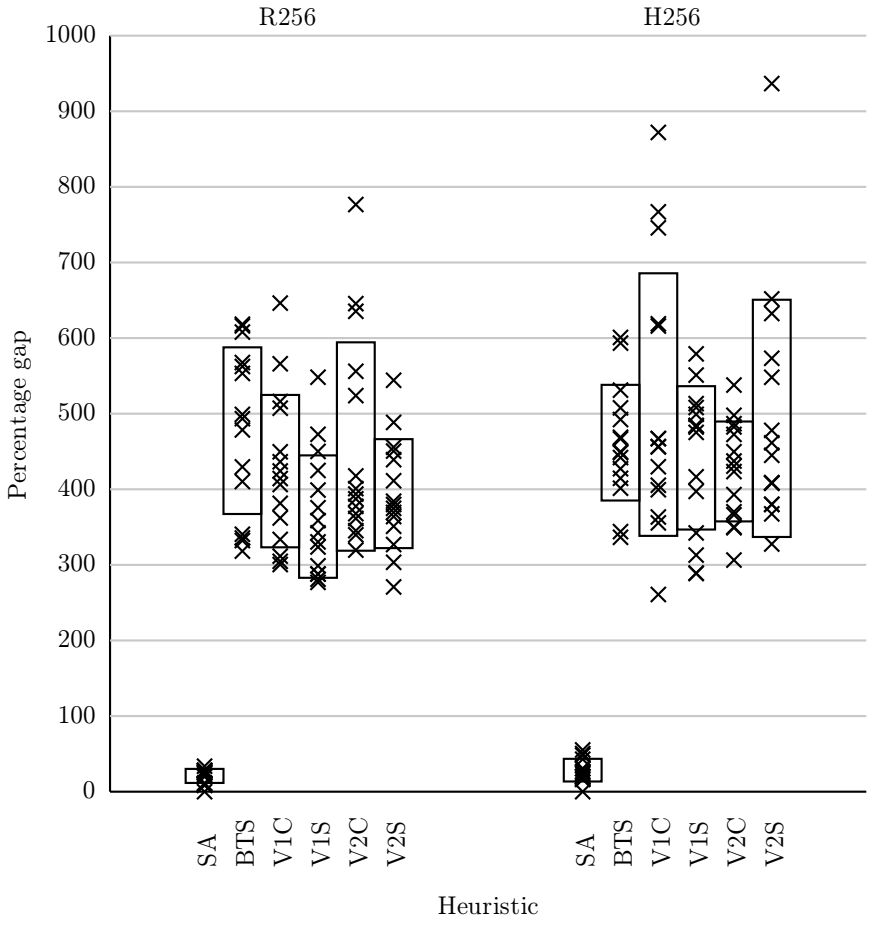


Figure 6.5: Results for *PBG*s with 256 IP cores.

The difference between SA and the tabu search variations is even more distinct for the *PBG*s with 256 IP cores, shown in Figure 6.5. Noticing the change of scale between Figures 6.3 and 6.4 and Figure 6.5, it is seen that none of the tabu search variations find solutions within 200% of those found by simulated annealing. Iteration times up to twenty seconds are observed for the tabu search variations for these sized problems. With the small number of iterations the tabu search variations can make in the allocated time, they can not be expected to perform much better than random sampling, as is indicated by the very large spread of the best found solutions. Comparing their individual performance therefore brings limited value to this discussion.

Overall, simulated annealing must be said to be the best heuristic of the ones investigated here, although V1S and V2S perform somewhat better for small problems.

6.7 Summary

In this chapter, the topology synthesis problem in context of the MANGO NoC has been investigated. An analytical model has been presented for finding the (worse than) worst case period for the communication of a periodic application that is modelled by a periodic bandwidth graph. Six different heuristics for solving the topology synthesis problem have been investigated. These are simulated annealing and five variations of tabu search: Basic tabu search both as an independent heuristic and augmented with two different criteria for performing two different diversifications.

Synthetic periodic bandwidth graphs have been used for tuning the heuristics' parameters and comparing the heuristics to each other. For small periodic bandwidth graphs with 16 IP cores, the heuristics are almost equal, but as the graph size increases to 64 and 256 IP cores, the computational intensity of the tabu searchs' exhaustive search of the neighborhood severely limits the number of iterations that may be carried out in the allocated time. Thus, with the given constraints, simulated annealing must be said to be the best overall heuristic from the ones investigated here for solving the topology synthesis problem in context of MANGO.

Chapter 7

The ReNoC Configuration Problem

This chapter presents methods for synthesizing topologies and mapping them on ReNoC-based platforms, while producing application-specific, deadlock-free routing algorithms. Collectively, these three problems are called the configuration problem. The chapter is organized as follows: First, in section 7.1, the overall ReNoC architecture and two specific instances of it are described. Then, in section 7.2, a model of ReNoC is described, while section 7.3 presents the heuristics for solving the aforementioned problems. In section 7.4, the experiments conducted for evaluating the heuristics and the specific instances of the ReNoC architecture are presented, and in section 7.5, the results of these experiments are discussed. Finally, section 7.6 gives conclusions on the heuristics and the ReNoC architecture. The ReNoC architecture has been developed by Mikkel Stensgaard [71]. The contributions presented here relate to the identification of the configuration problem and the development of heuristics to solve it, but not to the architecture itself. This chapter is based on the work presented in [75, 76].

7.1 The ReNoC Architecture

This section introduces the ReNoC architecture, first describing the generic architecture and its motivation, and then the specific instances that are used to evaluate the architecture and the algorithms for solving the configuration

problem.

As outlined in chapter 1, the increasing design effort, time-to-market, and non-recurring engineering costs of integrated circuits are expected to cause a shift from application-specific integrated circuits to more flexible platform systems for many applications.

The ReNoC architecture aims at providing a highly efficient interconnect in such platforms: Instead of providing a static interconnect that is dimensioned to be able to support *all* reasonably expectable traffic patterns at *any* time, provide a dynamic interconnect that can support *any* reasonably expectable traffic pattern *one* at a time, i.e., instead of an interconnect that is oversized for the task at hand and is thereby power inefficient, provide an interconnect that is adaptable to match the task at hand making it power efficient. ReNoC is adaptable in exactly this manner, as it allows an application to configure the topology of the NoC interconnect to best match its communication requirements.

Providing this flexibility is accomplished through the use of power efficient, physically circuit-switched topology switches (TS) as illustrated in Figure 7.1. Figure 7.1(a) shows a mesh of network nodes with one IP core attached to each node and double, bidirectional links between neighboring nodes. The ReNoC architectural extension (the topology switch) is shown in Figure 7.1(b) encircling a conventional packet-switched router. A close-up view of a conceptual ReNoC port is shown in Figure 7.1(c). In conventional NoCs, links are connected straight to the packet-switched routers' ports. The ReNoC topology switches also allow links to be connected directly to each other, bypassing the router. In this way, the packet-switched routers may be connected in a different topology than the underlying topology of network nodes. This underlying topology is denoted the *physical architecture*, while the topology of packet-switched routers after configuring the topology switches is denoted the *logical topology*.

ReNoC's flexibility makes it very well-suited for platform chips, which are meant to support a wide range of applications on a single hardware design, as each application can configure its own logical topology. This leads to a unique combination of flexibility and power efficiency of the interconnect in such a platform.

Figure 7.2(a) illustrates an application represented as a bandwidth graph, while Figure 7.2(b) shows a logical topology that supports the communication requirement of this application and that may be configured on the physical architecture in Figure 7.1(a). This logical topology demonstrates some of the possible connections that may be made using ReNoC: Pairs of IP cores are connected directly to each other, effectively forming point-to-point links using the physical circuit-switching provided by ReNoC. Other IP cores (e.g., ARM

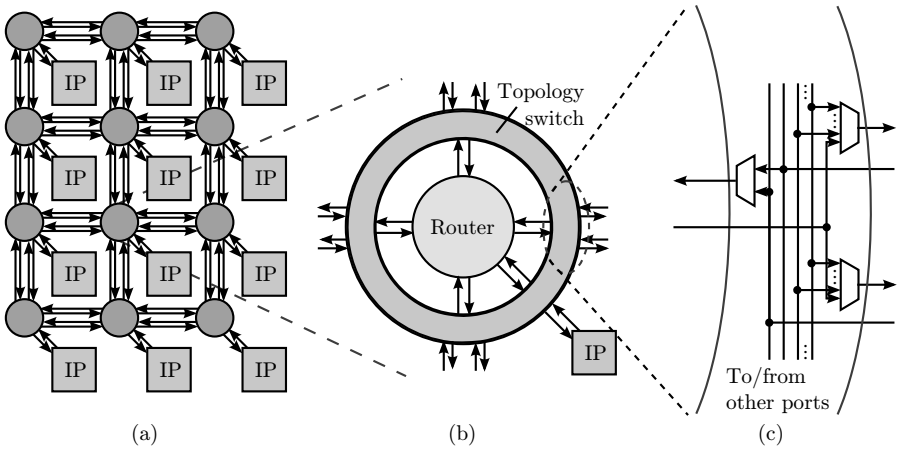


Figure 7.1: From left to right: A 3×4 2D mesh with double links, a detailed view of a network node with an IP core and a router encircled by a topology switch, and a close-up view of a TS port.

in the lower left corner) are connected directly to routers that are multiple hops away in the physical architecture, but only a single hop in the logical topology, while yet other IP cores and routers are connected as they would be in a conventional NoC. For this last case, the ReNoC topology switches constitute a slight overhead in power consumption compared to a static, conventional NoC. It is assumed that unused routers and IP cores are powered down.

It should be noted that although a specific packet-switched router is used in both previously presented work [71, 75] and in this thesis, ReNoC is not tied to this specific router. 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 [10] or ×pipes [21].

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 120 ps for a flit on a 1 mm link, thus – with a 100 MHz clock – very long, logical links can be formed with no need for pipelining [71]. 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 [16, 17] may be used to arbitrarily add pipeline

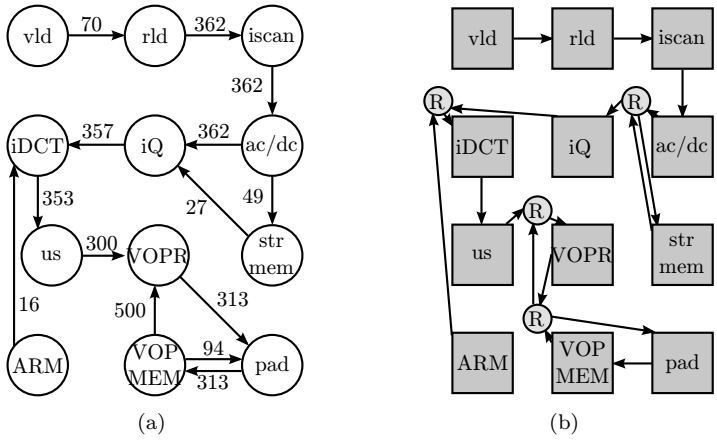


Figure 7.2: (a) The VOPD application from [71] and (b) a logical topology configured on a 3×4 physical architecture.

registers without changing the circuits' functionality. If the NoC is implemented using asynchronous techniques [67], such insensitivity to the addition of pipeline registers is typically already present.

The detailed view of a port on a TS in Figure 7.1(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 [22].
- 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, tri-state buffers or multiplexers as shown in Figure 7.1(c).

As shown in the logical topology in Figure 7.2(b), the ReNoC architecture does not impose any requirement on bidirectional connections through the NoC. For example, considering R0 in Figure 7.3, 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 [70] rely on bidirectional topologies and can thus not be applied to ReNoC in general. The approach to avoiding deadlocks is described together with the heuristics in section 7.3.

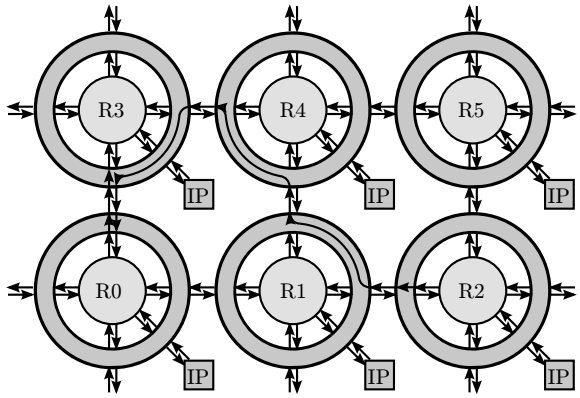


Figure 7.3: A logical topology does not necessarily have bidirectional links. Observe the north port of router R0, where the output goes to the south port of router R3, but the input comes from the west port of router R2.

The previous paragraphs concerned the generic ReNoC architecture. In the following, the specific implementation that is used in the evaluation is presented. Mesh-based physical architectures are used, although 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, a few restrictions are imposed 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, i.e., 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, i.e., 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, cf. item number two above. If an application does *not* use a given TS port, an enable-bit prevents the port from forwarding flits.

7.2 Modelling ReNoC

This section describes the models used 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. A fixed mapping of the application to IP cores is assumed. Using O for the set of IP cores:

Def. 7.1. 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. This is identical to definition 2.2.

Def. 7.2. 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.

A graph representation is also used for the physical architecture. Most graph representations of networks use vertices to represent routers and edges to represent links between routers as in definition 2.3. Here, it is necessary 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, vertices are used to represent *ports* on both routers, TSs, and IP cores. The sets of router, TS, and IP core ports are called U, S and O respectively. These are also indicated in Figure 7.4. It is 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 7.1), 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.

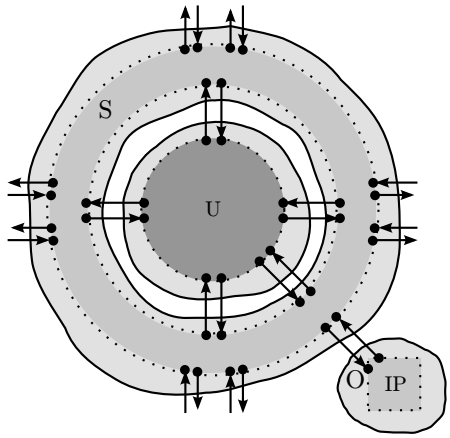


Figure 7.4: The vertices contributing to the sets O, S, and U from a single network node. The complete sets are found by taking the union of these subsets over all network nodes. Edges internal to the TS and the router are not shown.

Def. 7.3. 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 the term "link" is used to describe any edge in the NG – when a reference is made to links between network nodes, the term "NoC links" is used. For IP cores, input and output ports are differentiated from each other using subscripts, e.g., $o_{i,in}$. Two parameters are associated 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 NoC links but also the energy consumed internally in routers and TSs, i.e., 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 q, it is ensured that the saturation point is never reached. This approach is equivalent to the one used in [61], 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 $e = (u, v) \in E$, u is defined as the source and v as the destination of e, $\mathrm{src}(e) = u, \mathrm{dst}(e) = v$.

Routes are paths in the network between pairs of IP cores. In the context of these graph representations, a route contains *all ports* that a packet passes through, not only the ones where actual routing decisions are made. The routes are trimmed down to the necessary parts as a post-processing step. Note that the term "routing algorithm" is used in its most general sense of how to come from A to B, rather than an actual algorithmic description of the route taken.

Def. 7.4. A route $\mathcal{R}(o_i, o_j)$ between the two 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 $p_{n-1} = o_{j,in}$. The route servicing a connection c is defined as $\mathcal{R}(c) = \mathcal{R}(M(\mathrm{src}(c))_{out}, M(\mathrm{dst}(c))_{in}), c \in C$, i.e., 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. It is possible to determine if a deadlock is possible by analyzing if the set of all routes form a cycle in a graph. To do so, a dependency graph similar to the application-specific channel dependency graph in [63] is used.

Def. 7.5. 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 $\exists \mathcal{R} \in R \exists i . \mathcal{R}_i = p_i \wedge \mathcal{R}_{i+1} = p_j$, i.e., 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, a configuration graph is used that has the same vertices as the NG but whose edges are a subset of those in the NG. Specifically, internally to TSs only those edges that correspond to the connections between ports are included, i.e., an output port has multiple incoming edges but in the configuration graph only the one from the input port that is selected by the multiplexer shown in Figure 7.1(c) is included. Partial configurations are allowed, meaning that the configuration graph contains multiple edges for one TS port. This is useful when building solutions iteratively.

Def. 7.6. A configuration graph is a directed graph $CG = (P, A)$. The two edge properties are copied from the NG for each link $a \in A$.

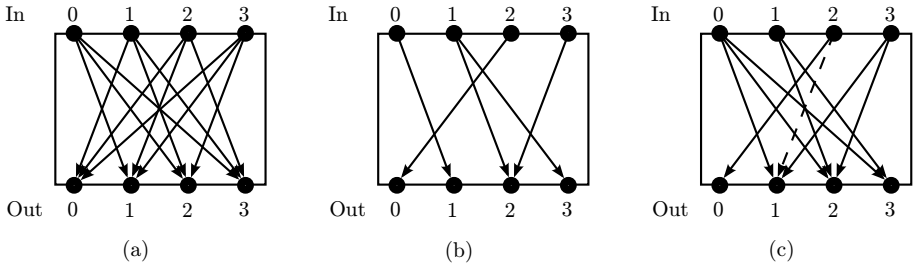


Figure 7.5: Care must be taken when unconfiguring links. In (c), the dashed link should *not* be added when unconfiguring $(0, 1)$, as this would inadvertently unconfigure $(2, 0)$.

An edge $a \in A$ in a TS is said to be *configured* by removing the other edges incident on $\mathrm{src}(a)$ or $\mathrm{dst}(a)$ 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 7.5, where a TS with four input and output ports is shown. In Figure 7.5(a), the unconfigured TS is shown, in Figure 7.5(b), the links $(0, 1)$ and $(2, 0)$ have been configured, while Figure 7.5(c) shows what happens 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

$$\forall c \in C : \mathcal{R}(c) \in R \tag{7.1}$$

$$\forall \mathcal{R} \in R \forall i < |\mathcal{R}| - 1 : (\mathcal{R}_i, \mathcal{R}_{i+1}) \in A \tag{7.2}$$

$$\forall p_i, p_j \sum_{\{c \in C | (p_i, p_j) \in \mathcal{R}(c)\}} b_c \leq q_{p_i, p_j} \tag{7.3}$$

$$\forall \langle \dots p_{i-1} p_i p_{i+1} \dots \rangle \in DG : \nexists i . p_i = p_j \wedge i \neq j \tag{7.4}$$

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, i.e., 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 makes. 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 constitute the set of all edges used in all routes, the absence of cycles guarantees freedom from routing deadlocks. A solution that fulfills all four requirements is said to be valid.

7.3 Optimization Approaches

In this section, three algorithms for solving the configuration problem are presented. 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 make modifications – *specializations* – to these.

In general, these algorithms are based on 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 non-minimal 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 one. Finding the optimal solution with minimal energy consumption would require an exhaustive search.

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

Figure 7.6: Pseudo code for the constructive algorithm.

7.3.1 Constructive Algorithm

The constructive algorithm is a greedy algorithm that starts from an unconfigured CG. The pseudo-code is given in Figure 7.6. 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 long link directly between IP cores, it is necessary to consider if this is allowed for the given connection – 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 for future work either backtracking or rerouting can be implemented 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 packet 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 is not exceeded.

As mentioned above, 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 an end-to-end circuit-switched link). If both are true, three cases can occur:

1. This is the only connection from $\mathrm{src}(c)$, but $\mathrm{dst}(c)$ is the destination of multiple connections. $M(\mathrm{dst}(c))_{in}$ 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 $\mathrm{src}(c)$, but this is the only connection to $\mathrm{dst}(c)$. $M(\mathrm{src}(c))_{out}$ is connected to the closest router, such that the traffic streams are split as close to the source IP core as possible.

3. $\mathrm{src}(c)$ is the source of multiple connections and $\mathrm{dst}(c)$ the destination of multiple connections. The sums of the bandwidth requirement of the

connections originating and terminating in src(c) and dst(c) respectively are found. 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 equation (7.1) as the loop is over all $c \in C$.

The routability requirement in equation (7.2) is also satisfied. To realize this, first assume that the requirement is violated, i.e., $\exists \mathcal{R}_i.(\mathcal{R}_i, \mathcal{R}_{i+1}) \notin A$. 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 time 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 equation (7.3) 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 equation (7.4) is satisfied, as the algorithm actively fails in case of a deadlock, i.e., 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 of the execution of the algorithm on a small problem is shown in Figure 7.7. 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

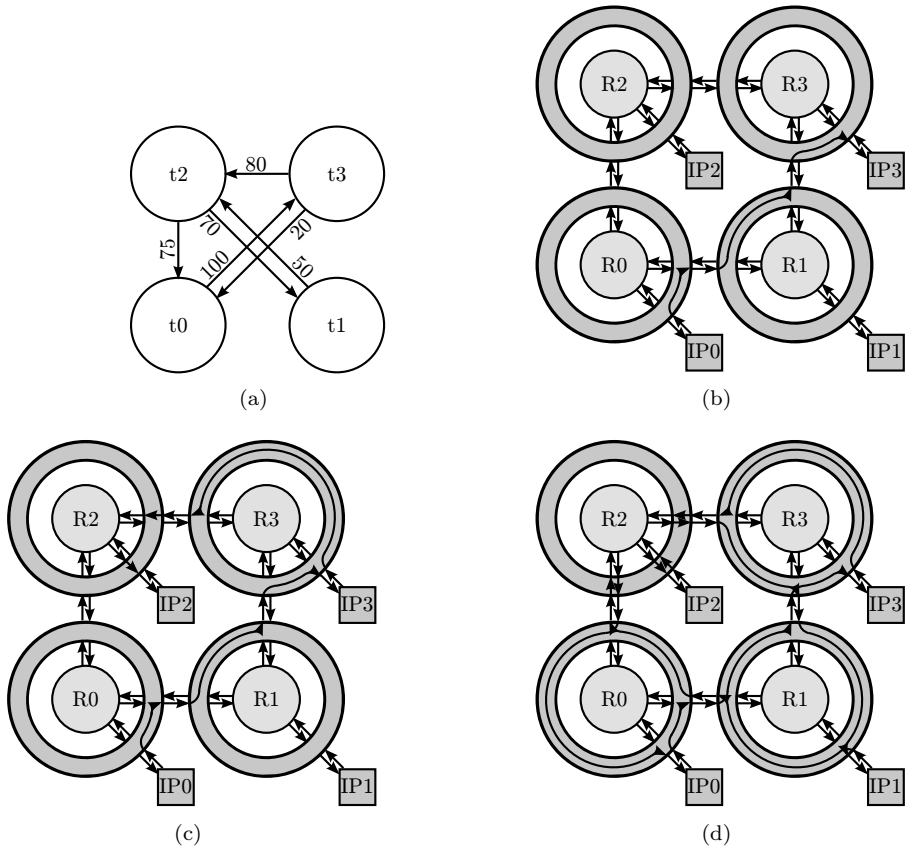


Figure 7.7: An example of the constructive algorithm's execution. The *BG* is shown in (a), intermediate steps in (b) and (c), and the solution in (d).

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).

A small variation of the algorithm has also been implemented 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.

7.3.2 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.

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. Here, two ways of acquiring the initial configuration are used: (1) letting the greedy algorithm generate the initial configuration and (2) generating an initial configuration in which the TSs are configured such that the logical topology matches the physical architecture, in this 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, two different routing functions are used: 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 the earlier comment about not being able to use deadlock-free routing algorithms from literature does not apply here, as the topology in this case in fact is a regular one with bidirectional links. A greedy algorithm as shown in Figure 7.8 is used to select the routes for all connections. In dimension ordered routing, only one route exists for each connection, whereas

in e.g., 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.

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

Figure 7.8: Pseudo code for generating an initial configuration that matches the physical architecture, here a mesh.

The algorithm in Figure 7.8 is quite similar to the constructive algorithm with the exception that it is 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 equations (7.1) and (7.3) 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 equation (7.2) is also satisfied, and the chosen routing functions are known from literature to be deadlock-free. With the initial configuration now in place, the specializations can be considered.

Specialization A: Bypass Router

The pseudo code for the first specialization is given in Figure 7.9 with an example shown in Figure 7.10. 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 originates in u_i. As DG describes which edges

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)

Figure 7.9: Pseudo code for specialization A: Bypass Router.

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, a bypass is inserted using the TS.

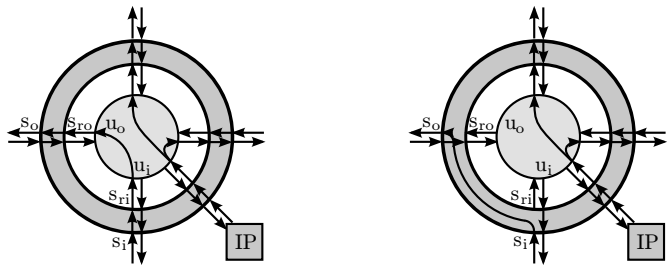


Figure 7.10: An example of specialization A: (a) before and (b) after inserting a bypass.

Referring to the example in Figure 7.10, 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 7.10(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 $\phi' = \langle s_i s_o \rangle$ the router has been bypassed as shown in Figure 7.10(b). Corresponding changes are made to D as well. No further bypasses can be made in the example in Figure 7.10 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 a valid solution is assumed *before* the specialization is applied. Thus, it is only necessary 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 ϕ is replaced with ϕ' in all routes where ϕ occurs and changes corresponding to this replacement are made to A, i.e., all routes previously using ϕ use ϕ' after the specialization has been applied and A has been updated to include ϕ' instead of ϕ.

For the capacity requirement, the fact that the capacity q of all links belonging to TSs is identical is utilized: 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 the capacity of (s_i, s_{ri}) is assumed to be 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 is applied, it is known that the out-degree of all vertices in ϕ is one (the very reason the bypass could be introduced). 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 also does not form any cycles in the DG. No cycles can possibly be introduced in the DG by this specialization.

Specialization B: Insert Long Links

This specialization takes a route and modifies it by trying to insert the longest link possible. The pseudo-code can be found in Figure 7.11.

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

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

i.e., 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

InsertLongLinks(BG, CG, NG, M, R)

```
1: Sort C according to b
2: for all c ∈ C do
3:    Find X_in, X_out and Y
4:    for all y ∈ 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 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 ∈ Y
9:       Configure path in CG; Update DG; Update R(c)
10:      for all R ∈ R_upd do
11:         Find new route for R
12:         if No route found then
13:            Undo changes to CG, DG and all R; Continue with next y ∈ Y
14:         Configure path in CG; Update DG; Update R
15:   if DG is cyclic then
16:      Undo all changes in this iteration
```

Figure 7.11: Pseudo code for inserting long links using specialization B.

$x_{out} \in X_{out}$ where $x_{in} = \mathcal{R}_i, x_{out} = \mathcal{R}_j, j > i$ and sorting them according to their distance $j - i$, an exhaustive search can be made for the possible long links to insert, terminating when the longest possible one is found. This sorted set of pairs is called 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, i.e., if inserting a long link for a connection requires another connection with a 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 a route

is never removed from R. The routability requirement is also satisfied, as whenever changes are made 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.

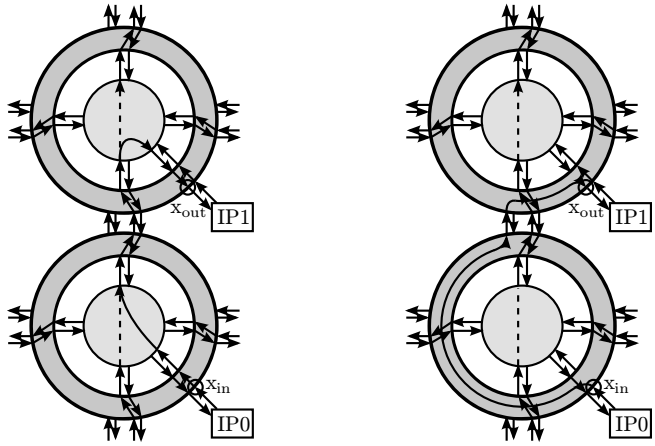


Figure 7.12: An example of specialization B. (a) shows two connections before the specialization is applied, while (b) shows the two connections after the specialization has been applied.

An example of the execution of this algorithm is shown in Figure 7.12, 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.

7.4 Experiments

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

As the focus is on the power savings achieved by using ReNoC to move traffic out of routers and on to long, logical links, it would be an unfair comparison to use a large router with many features and high power consumption. Therefore routers similar to those in [71] are used: 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.

For comparison purposes, three different physical architectures are considered, 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 provides a baseline to evaluate the overhead of the TSs. This physical architecture is referred to as a static mesh. For routing in this mesh, both dimension ordered (XY and YX) and north-, south-, east-, and west-first routing are used.
- The first ReNoC-based physical architecture is, as above, a standard mesh of network nodes, where network nodes here consist of a router, a TS, and an IP core. This physical architecture is referred to as the single-link architecture (SL). An example can be seen in Figure 7.3.
- The second ReNoC-based physical architecture is also a mesh of network nodes as above, 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 e.g., allows a long, logical link to skip unaffected through an area that is otherwise congested. This physical architecture is referred to as the double-link architecture (DL). This is the architecture shown in Figure 7.1.

Table 7.1 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 [71, 4]. All figures in the table are pre-layout, based on low-leakage cells from a commercial 90 nm standard cell library, using a 1 V 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

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

Table 7.1: Energy- and power-consumption of the components in a ReNoC-based NoC. The two energy/packet values are to a router input and to a NoC link output respectively.

TSs have previously been shown to add around 10% to the area of the interconnect [71].

A mixture of real and synthetic applications is used to evaluate the algorithms and physical architectures. The following applications can be found in literature:

- VOPD: A multimedia application [71].
- MMS: A multimedia system [39].
- Rotate and complement: Well known traffic patterns from computer networks [23] that have also been suggested for use in NoC micro benchmarks [66].

A larger synthetic application (S64) that incorporates some of the patterns suggested in [66] is also used. Characteristics of the different applications can be found in Table 7.2.

The mapping M has been manually generated 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.

The conducted experiments are (for all applications on SL and DL):

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

Table 7.2: Characteristics of the applications. $|T|$ is the number of task sets (IP cores), and $|C|$ is the number of connections between these.

- Configure a logical mesh and route using the algorithm in Figure 7.8. This allows us to evaluate the overhead of the TSs.
- Use the constructive algorithm (Figure 7.6) to configure each application on each platform.
- Generate an initial configuration using the algorithm in Figure 7.8 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, the power consumption is evaluated 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, the best of the results are selected, as the evaluation is not intended to decide whether e.g., XY routing is better than YX routing.

7.5 Results

In this section, the results of the experiments described above are presented and discussed. The power consumption in "links" shown in the graphs in this

section relates to "NoC links," i.e., links between network nodes.

In Figure 7.13, the overhead of the ReNoC architecture when the logical topology forms a mesh is shown. The power consumption of each application has been normalized to that of the static architecture. As can be seen, the TSs only add between two and five percent to the power consumption. This is the worst-case scenario as the TSs' ability to move traffic out of routers is not utilized.

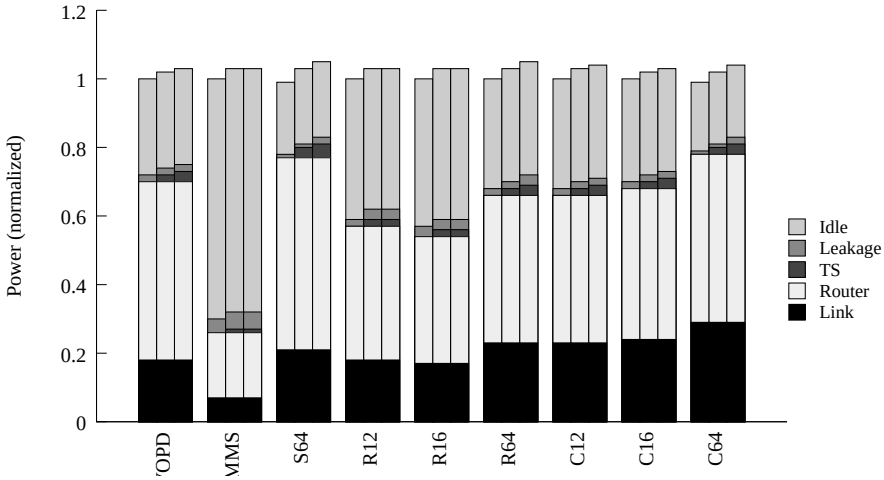


Figure 7.13: ReNoC power overhead. The bars represent the power consumption normalized to the static architecture. The order of the bars is: Static, SL, and DL.

Figure 7.14 shows a comparison of the different physical architectures. For each combination of physical architecture and application, the best solution from all the experiments has been selected 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% 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

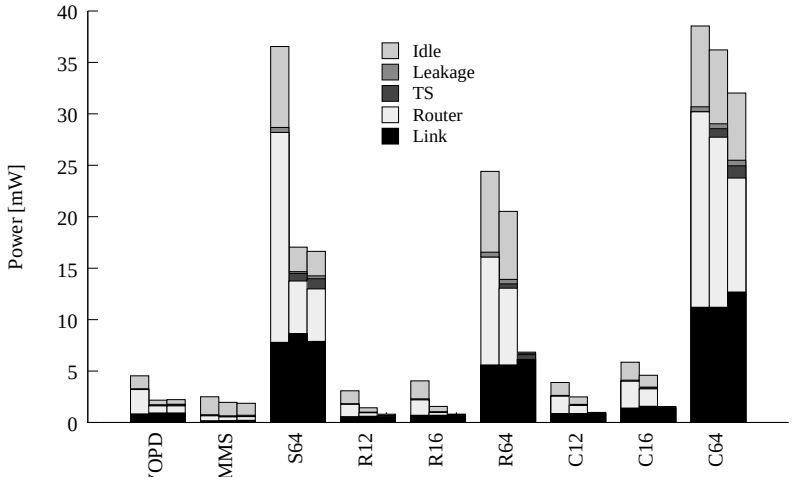


Figure 7.14: Comparison of the physical architectures. The order of the bars is: Static, SL, DL.

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 7.3 shows the number of routers that are powered on in the same solutions whose power consumption is shown in Figure 7.14. 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 on to 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 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 7.14, 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

		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

Table 7.3: The number of powered-on routers in the best configuration of all physical architectures and applications.

some routers where the streams in question are neither split, nor merged. In Figure 7.14, 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 on to a long link that passes right by the busy routers.

Figures 7.15(a) and 7.15(b) give a comparison of the algorithms on SL and DL respectively. The bars are normalized to the first non-zero 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

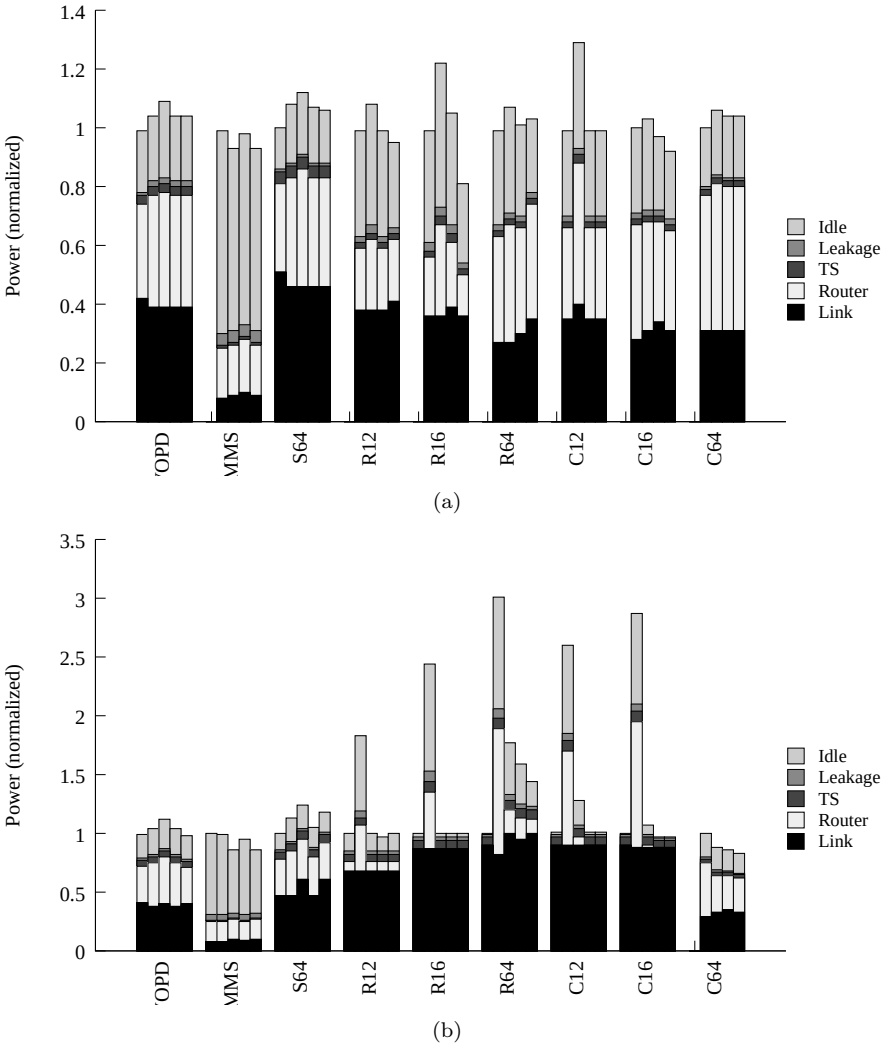


Figure 7.15: Comparison of the optimization algorithms for SL (a) and DL (b). The order of columns is constructive algorithm, specialization A, specialization B, AB, and BA. Bars are normalized to the first non-zero bar for each application.

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 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 on to 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 above, 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 without the preprocessing step or by 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 constructive algorithm produces the best results.

The constructive algorithm has also been used to create application-specific, deadlock-free routing algorithms in the static mesh. However, for the considered 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, it is possible on a single chip to switch between the best found configurations 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.

7.6 Summary

In this chapter, optimization algorithms in context of the ReNoC architecture have been presented. Two example architectures – both based on a 2D mesh topology – have been used for evaluating both the algorithms and the ReNoC architecture itself. In the first architecture, the number of links matches the number of router ports, while in the second architecture, the number of links has been doubled. These platforms are called SL (single link) and DL (double link) respectively, and their performance is compared to a conventional 2D mesh topology without topology switches (denoted "static").

The ReNoC configuration problem has been identified and formalized in section 7.2. It consists of three sub-problems: Topology synthesis, topology mapping, and routing. The optimization algorithms solve these three problems in a single pass with the objective of minimizing the power consumption in the interconnect.

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 non-deadlocking 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.

Chapter 8

Conclusions and Future Directions

This thesis has presented three contributions in two different areas of network-on-chip and system-on-chip research: Application modelling and optimizations of the network. The following provides conclusions about the research presented in this thesis and puts it into perspective and discusses future directions

8.1 Conclusions and Perspective

The presented research on application modelling takes an analytical approach to determining the traffic pattern caused by an application that is modelled with a task graph and executing on a cache-coherent shared-memory system using a directory to ensure memory consistency. The analysis produces a worse-than-worst case bandwidth graph of the communication caused by memory accesses and the cache-coherence protocol. The analytical method has been evaluated using worst-case, best-case, and synthetic task graphs. Determining the method's usefulness for real applications requires evaluation using such applications.

The ability to analytically determine a worst-case bandwidth graph is highly useful in system interconnect design flows. Most interconnect synthesis tools in literature require the application's traffic pattern to be described using a bandwidth graph. This is also true for the algorithms for solving the synthesis and configuration problems in MANGO and ReNoC presented in this thesis. Such an analytical method may also be used to extend existing design flows to

include the mapping of task graphs to IP cores. Using an analytical method allows the traffic pattern resulting from each mapping to be quickly evaluated, which in turn leads to faster iterations of the design-space exploration compared to determining the traffic pattern through simulation.

Previous approaches to analytically determine a bandwidth graph from a task graph have assumed communication between IP cores to be implemented by passing messages directly between communicating pairs of IP cores. However, many – if not most – multiprocessors systems are based on communication occurring through variables located in shared memory. Developing an analytical method for determining a bandwidth graph for an application executing on a shared-memory system is therefore highly important.

The bandwidth graphs produced by the analytical method can be used as input to many of the design flows presented in literature for synthesizing or otherwise optimizing interconnects. This includes the algorithms presented in this thesis for optimizing in the MANGO and ReNoC networks-on-chips.

The research concerning MANGO presented in this thesis developed and evaluated six heuristics for synthesizing an application-specific topology. One heuristic is based on simulated annealing, while the other five are based on tabu search with different implementations of long-term memory and diversifications of the search. The evaluation shows that for smaller sized problems – systems with fewer network nodes – several of the heuristics produce good results. However, when the problem sized is increased, the slow iteration time of tabu search severely limits the number of iterations carried out and thereby also the number of points in the search space that are evaluated.

Multiple tools for synthesizing application-specific network topologies have been presented in literature. However, these tools are often made with a specific network-on-chip architecture in mind and may be easily adapted to other architectures with similar characteristics. When a network-on-chip architecture differs sufficiently in these characteristics – for example architectural details or provided features such as guaranteed service – the heuristics used to solve the optimization problems need to be adapted to take these characteristics into consideration.

The advantage to adapting the heuristics or even developing new heuristics is the fact that doing so allows designers to make use of the network-on-chip architecture's features without needing to be experts on the architecture, similar to how most hardware designers today need not worry about layout, placement, routing (of wires), or synthesis: Tools exist for solving all of these problems. At the time of writing this, commercialization of network-on-chip design tools is not widespread, but with continuing increase in the number of devices that may

be integrated in a given unit of area, the need for network-on-chip interconnects is sure to arrive together with the need for tools to design and optimize these interconnects.

The continuing development in integration density however also leads to increasing cost of designing and manufacturing chips to such a high degree that doing so is set to become infeasible for most applications. This necessarily leads to the use of generic platform chips for these applications. These two ways – ASICs and platforms – of implementing an application represent different trade-offs of efficiency on one hand and cost on the other. The ReNoC architecture is designed to bridge the gap between these trade-offs by allowing efficient implementations at low cost.

The final contribution of this thesis is research into tools that ease the use of the ReNoC architecture. This constitutes both identifying and formalizing the optimization problems that need to be solved and developing heuristics for solving these problems. In this thesis, the ReNoC configuration problem has been identified and formalized, and three heuristics for solving it have been developed and evaluated.

To summarize, the three contributions of this thesis are (1) an analytical method of determining the traffic pattern caused by an application that is modelled as a task graph and executing on a shared-memory system, (2) development and evaluation of six heuristics for synthesizing application-specific topologies using the MANGO network-on-chip, and (3) identification and formalization of the ReNoC configuration problem and development and evaluation of three heuristics for solving it.

These individual research contributions may be combined such that a designer can start with an application modelled as a task graph and synthesize an application-specific shared-memory system-on-chip using MANGO as the interconnect or to configure a reconfigurable platform with a ReNoC-based interconnect such that the interconnect best matches the application's communication requirements.

8.2 Future Directions

The research areas addressed in this thesis may be explored further in the future. This section presents a discussion on the directions this future research may take.

The analytical derivation of bandwidth graphs from task graphs as described here makes a number of assumptions on the application's behavior. One direction for expanding on this research is to refine the analysis in order to reduce

the number of assumptions. One interesting approach for doing so is to associate a memory trace with each task, rather than associating an address range with each edge, and making a simulation of the local cache in the analysis. This simulation should not include the coherence protocol, but only be used to determine which memory accesses result in cache hits, and which ones result in cache misses. Evaluating the method with real applications will allow conclusions to be made about the feasibility of using an analytical approach to determining traffic patterns in shared-memory systems from task graphs.

The research on synthesizing application-specific topologies in MANGO is based on the assumption that applications are greedy in their communication patterns: Every IP core attempts to send a new flit as often as possible. Developing heuristics that take advantage of the latency guarantees provided by MANGO is one future direction for this research area.

Considering ReNoC, the configuration problem has been identified and formalized. Additional heuristics or improvements to the existing ones may be developed in the future to produce even better results. Another direction would be to develop heuristics that optimize the mapping of vertices in the bandwidth graph to IP cores in the system.

A third direction for research into the optimization problems surrounding ReNoC is to consider the problem of synthesizing a heterogeneous, ReNoC-based platform. Solving this problem would require a certain amount of knowledge about the application domain and common characteristics of applications' communication patterns in order to synthesize a reconfigurable platform that on one hand is not over dimensioned and on the other hand is sufficiently flexible.

Bibliography

[1] Federico Angiolini, Shankar Mahadevan, Jan Madsen, Luca Benini, and Jens Sparsø. Realistically rendering SoC traffic patterns with interrupt awareness. In *IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, sep 2005.

[2] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. Multi-objective mapping for mesh-based NoC architectures. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 182–187, New York, NY, USA, 2004. ACM.

[3] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. An evolutionary approach to network-on-chip mapping problem. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, pages 112–119, sept. 2005.

[4] Arnab Banerjee, Robert Mullins, and Simon Moore. A power and energy exploration of network-on-chip architectures. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 163–172, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[6] Davide Bertozzi and Luca Benini. Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4, 2004.

[7] Tobias Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building

321, DK-2800 Kgs. Lyngby, 2005. Supervised by Assoc. Prof. Jens Sparsø, IMM.

[8] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1):1, 2006.

[9] Tobias Bjerregaard, Shankar Mahadevan, Rasmus G. Olsen, and Jens Sparsø. An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 171–174, nov 2005.

[10] Tobias Bjerregaard and Jens Sparsø. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Tobias Bjerregaard and Jens Sparsø. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 34–43, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture, special issue on Network on Chip*, 50:105–128, February 2004.

[13] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Routing table minimization for irregular mesh NoCs. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 942–947, San Jose, CA, USA, 2007. EDA Consortium.

[14] Evgeny Bolotin, Arkadiy Morgenshtein, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Automatic hardware-efficient soc integration by QoS network on chip. In *11th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2004*, pages 479–482. Institute of Electrical and Electronics Engineers Computer Society, 2004.

[15] Edmund K. Burke and Graham Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, 2005.

[16] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, 2002.

[17] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. Elastic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1437–1455, 2009.

[18] Jeremy Chan and Sri Parameswaran. NoCOUT: NoC topology generation with mixed packet-switched and point-to-point networks. In *ASP-DAC '08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 265–270, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[20] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[21] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. xpipes: A latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 536, Washington, DC, USA, 2003. IEEE Computer Society.

[22] William Dally. Enabling technology for on-chip interconnection networks. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[23] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[24] William J. Dally. Network simulator from [23]. http://cva.stanford.edu.

[25] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 684–689, New York, NY, USA, 2001. ACM.

[26] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: Task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society.

[27] José Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, 1993.

[28] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[29] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[30] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago Gonzalez Pestana, Andrei Radulescu, and Edwin Rijpkema. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1182–1187, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 954–959, San Jose, CA, USA, 2007. EDA Consortium.

[32] Andreas Hansson and Kees Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

[33] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic. *VLSI Design*, 2007, 2007.

[34] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007:Article ID 95859, 10 pages, May 2007. Hindawi Publishing Corporation.

[35] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2005. ACM.

[36] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2007.

[37] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *ASP-DAC '03: Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 233–239, New York, NY, USA, 2003. ACM.

[38] Jingcao Hu and Radu Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10688, Washington, DC, USA, 2003. IEEE Computer Society.

[39] Jingcao Hu and Radu Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, April 2005.

[40] Open SystemC Initiative. http://www.systemc.org.

[41] Tang Lei and Shashi Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *DSD '03: Proceedings of the Euromicro Symposium on Digital Systems Design*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

[42] Ai-Hsin Liu and Robert P. Dick. Automatic run-time extraction of communication graphs from multithreaded applications. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 46–51, New York, NY, USA, 2006. ACM.

[43] Zhonghai Lu, Lei Xia, and Axel Jantsch. Cluster-based simulated annealing for mapping cores onto 2D mesh networks on chip. In *DDECS '08: Proceedings of the 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society.

[44] Jan Madsen, Thomas K. Stidsen, Peter Kjærulf, and Shankar Mahadevan. Multi-objective design space exploration of embedded system platfoms. In *IFIP Working Conference on Distributed and Parallel Embedded Systems*. IFIP, oct 2006. Invited paper.

[45] Philippe Magarshack and Pierre G. Paulin. System-on-chip beyond the nanometer wall. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 419–424, New York, NY, USA, 2003. ACM.

[46] Shankar Mahadevan. *Simulation-based Modeling Frameworks for Networked Multi-processor System-on-Chip*. PhD thesis, Technical University of Denmark, Informatics and Mathematical Modelling, Computer Science and Engineering, 2006.

[47] Shankar Mahadevan, Federico Angiolini, Jens Sparsø, Luca Benini, and Jan Madsen. A reactive and cycle-true IP emulator for MPSoC exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1):109–122, 2008.

[48] Shankar Mahadevan, Federico Angiolini, Michael Storgaard, Rasmus Grøndahl Olsen, Jens Sparsø, and Jan Madsen. A network traffic generator model for fast network-on-chip simulation. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 780–785, Washington, DC, USA, 2005. IEEE Computer Society.

[49] Sorin Manolache, Petru Eles, and Zebo Peng. Fault-aware communication mapping for NoCs with guaranteed latency. *International Journal of Parallel Programming*, 35(2):125–156, 2007.

[50] Radu Marculescu, Umit Y. Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(1):3–21, 2009.

[51] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20890, Washington, DC, USA, 2004. IEEE Computer Society.

[52] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar, and Axel Jantsch. The Nostrum backbone - a communication protocol stack for networks on chip. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 693, Washington, DC, USA, 2004. IEEE Computer Society.

[53] Mehdi Modarressi, Hamid Sarbazi-Azad, and Arash Tavakkol. Virtual point-to-point links in packet-switched NoCs. In *ISVLSI '08: Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, pages 433–436, Washington, DC, USA, 2008. IEEE Computer Society.

[54] Mehdi Modarressi, Hamid Sarbazi-Azad, and Arash Tavakkol. Performance and power efficient on-chip communication using adaptive virtual point-to-point connections. In *NOCS '09: Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 203–212, Washington, DC, USA, 2009. IEEE Computer Society.

[55] Srinivasan Murali. *Methodologies for Reliable and Efficient Design of Networks on Chips*. PhD thesis, Stanford University, March 2007.

[56] Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 27–32, New York, NY, USA, 2005. ACM.

[57] Srinivasan Murali, Martijn Coenen, Andrei Radulescu, Kees Goossens, and Giovanni De Micheli. Mapping and configuration methods for multi-use-case networks on chips. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 146–151, Piscataway, NJ, USA, 2006. IEEE Press.

[58] Srinivasan Murali, Martijn Coenen, Andrei Radulescu, Kees Goossens, and Giovanni De Micheli. A methodology for mapping multiple use-cases onto networks on chips. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 118–123, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[59] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto NoC architectures. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20896, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Srinivasan Murali and Giovanni De Micheli. Sunmap: A tool for automatic topology selection and generation for NoCs. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 914–919, New York, NY, USA, 2004. ACM.

[61] Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. Designing application-specific networks on chips with floorplan information. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 355–362, New York, NY, USA, 2006. ACM.

[62] Umit Y. Ogras and Radu Marculescu. "It's a small world after all": NoC performance optimization via long-range link insertion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(7):693–706, july 2006.

[63] Maurizio Palesi, Rickard Holsmark, Shashi Kumar, and Vincenzo Catania. A methodology for design of application specific deadlock-free routing algorithms for NoC systems. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 142–147, New York, NY, USA, 2006. ACM.

[64] Open Core Protocol International Partnership. http://www.ocpip.org.

[65] E. Rijpkema, K. Goossens, J. Dielissen, A. Rădulescu, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEE Proceedings: Computers and Digital Technique*, 150(5):294–302, sep 2003.

[66] Erno Salminen, Christian Grecu, Timo Hämäläinen, and André Ivanov. Network-on-chip benchmark specification part 1: Application modelling and hardware description version 1.0. Technical report, OCP (http://www.ocpip.org), 2008.

[67] J. Sparsø and S. Furber, editors. *Principles of Asynchronous Circuit Design – A Systems Perspective*. Kluwer Academic Publishers, 2001.

[68] Krishnan Srinivasan and Karam S. Chatha. A methodology for layout aware design and optimization of custom network-on-chip architectures. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 352–357, Washington, DC, USA, 2006. IEEE Computer Society.

[69] Krishnan Srinivasan, Karam S. Chatha, and Goran Konjevod. Linear-programming-based techniques for synthesis of network-on-chip architectures. *IEEE Transactions on Very Large Scale Integrated Systems*, 14(4):407–420, 2006.

[70] David Starobinski, Mark Karpovsky, and Lev A. Zakrevski. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Transactions on Networking*, 11(3):411–421, 2003.

[71] Mikkel Bystrup Stensgaard and Jens Sparsø. ReNoC: A network-on-chip architecture with reconfigurable topology. In *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 55–64, Washington, DC, USA, 2008. IEEE Computer Society.

[72] Matthias Bo Stuart. High-level modeling of network-on-chip. Master's thesis, Technical University of Denmark, Informatics and Mathematical Modelling, Computer Science and Engineering, 2006.

[73] Matthias Bo Stuart and Jens Sparsø. Custom topology generation for network-on-chip. In *25th Norchip Conference*, pages 81–84. IEEE, 2007.

[74] Matthias Bo Stuart and Jens Sparsø. Analytical derivation of traffic patterns in shared memory architectures from task graphs. In *27th Norchip Conference*, 2009.

[75] Matthias Bo Stuart, Mikkel Bystrup Stensgaard, and Jens Sparsø. Synthesis of topology configurations and deadlock free routing algorithms for ReNoC-based systems-on-chip. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 481–490, New York, NY, USA, 2009. ACM.

[76] Matthias Bo Stuart, Mikkel Bystrup Stensgaard, and Jens Sparsø. The ReNoC reconfigurable network-on-chip: Architecture, configuration algorithms, and evaluation. *ACM Transactions in Embedded Computing Systems*, 2010. Accepted.

[77] Keith S. Vallerio and Niraj K. Jha. Task graph extraction for embedded system synthesis. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, page 480, Washington, DC, USA, 2003. IEEE Computer Society.

[78] Chia-Ming Wu, Hsin-Chou Chi, and Ming-Chao Lee. Mapping of IP cores to network-on-chip architectures based on communication task graphs. In *2005 6th International Conference on ASIC*, pages 953–956, 2005.