A Metaheuristic Scheduler for Time Division Multiplexed Network-on-Chip

Sørensen, Rasmus Bo; Sparsø, Jens; Pedersen, Mark Ruvald; Højgaard, Jaspur

Publication date:
2014

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

A Metaheuristic Scheduler for Time Division Multiplexed Network-on-Chip*

Rasmus Bo Sørensen[1], Jens Sparsø[1], Mark Ruvald Pedersen[1], and Jaspur Højgaard[1]

[1]Department of Applied Mathematics and Computer Science, Technical University of Denmark, Email: {rboso@dtu.dk, jspa@dtu.dk, mark@radix63.dk, jaspurh@gmail.com}

February 17, 2014

DTU Compute Technical Report-2014-04

*This work was partially funded by the Danish Council for Independent Research | Technology and Production Sciences, Project no. 12-127600: Hard Real-Time Embedded Multiprocessor Platform (RTEMP) project, and by the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

Abstract

This report presents a metaheuristic scheduler for inter-processor communication in multi-core platforms using time division multiplexed (TDM) networks on chip (NOC). Input to the scheduler is a specification of the target multi-core platform and a specification of the application. Compared to previous works, the scheduler handles a broader and more general class of platforms.

Another contribution, which has significant practical implications, is the minimization of the TDM schedule period by over-provisioning bandwidth to connections with the smallest bandwidth requirements. Our results show that this is possible with only negligible impact on the schedule period.

We evaluate the scheduler with seven different applications from the MCSL NOC benchmark suite. We observe that the metaheuristics perform better than the greedy solution. In the special case of all-to-all communication with equal bandwidths on all communication channels, we obtain schedules with a shorter period than reported in previous work.

Chapter 1

Introduction

Today the hardware platforms used in general-purpose computing as well as in embedded systems are moving in the direction of many cores connected by a packet-switched network on chip (NOC) [1, 2]. Hard real-time systems are implemented in a conservative way. But these are also moving in the direction of multi-core platforms. The availability of many processing cores brings interesting perspectives related to time-predictability, for example isolation of applications executing on different processing cores. However, it also brings a number of challenges. One of these challenges is the NOC, which is a global and shared resource through which the cores communicate.

Most research in the field of NOC has addressed general-purpose computing or embedded systems that do not require hard real-time guarantees. In order to be useful in hard real-time systems, the NOC must provide (virtual) end-to-end circuits in order to avoid interference from traffic flows [3]. There are two fundamental ways of implementing virtual circuits: (a) by time division multiplexing (TDM) the hardware resources (routers and links) in the NOC, or (b) by using non-blocking routers in combination with rate control in the cores [3, 4].

Previous research has shown that TDM-based NOCs are far simpler to implement, because they avoid buffering and arbitration in the routers. In [3, 5] the difference in area between the two approaches is reported to be 1:10 – a strong argument in favor of TDM. A criticism often raised is that TDM-based networks are not work-conserving; this is generally of less interest in hard real-time systems, where the worst-case execution time is of importance.

In this report we address the scheduling of data-traffic in a TDM-based NOC that supports message passing across virtual end-to-end circuits in a multi-processor platform. There are two rather different variations of this problem of mapping applications to hardware platforms. The first variation maps the application onto a given hard real-time general-purpose platform.

The second variation synthesizes a specific NOC platform for the given application, including a TDM schedule.

The increasing cost of designing and fabricating integrated circuits and the limited fabrication volume of most hard real-time systems speaks strongly in favor of using platforms with some generality that can be used to implement a range of different applications. This is what we address in this report. The context for our work is the T-CREST project [6] in which we are developing a general-purpose multi-processor platform designed for hard real-time systems, and for which the scheduler described in this report is being used. Details of the hardware implementation may be found in [7, 8].

The mapping of an application onto a general-purpose NOC-based multi-core platform is generally understood to include the steps illustrated in Figure 1.1 [9, 10]. The application is modeled as a *task graph* where nodes represent tasks and edges represent communication channels (end-to-end circuits). Edges and nodes can be annotated with different information – for edges, required bandwidth is typically specified. From this starting point, the first step is to assign tasks to processors and as part of this to decide which tasks will share a processor. The result of this is a *core communication graph* where the nodes represent processing cores and the edges represent communication flows between the cores. Again (required) bandwidth is typically annotated to the edges. The binding of processors to cores in the platform influences the traffic in the NOC. The binding is driven by minimizing the number of router-to-router hops, and the total bandwidth of the communication flows that share a link in the NOC. These steps are not specific for TDM-based NOCs and they are well studied in the literature. Early works include [9].

The final step, and the topic of this report is to generate the TDM schedule for the NOC. An important aspect of this problem is to allow a sufficiently generic and parameterized specification of the NOC-based platform, in order to allow the scheduler to target a large class of different multi-core platforms using TDM-based NOCs. These parameters include the topology of the NOC (regular as well as irregular topologies), the packet length, and the pipeline depth of the routers and links. The scheduling problem can be modeled as a fixed-flow, minimum-time integer multi-commodity flow problem that is known to be NP-complete [11].
The report makes contributions in two areas:

- Our scheduler is more general and produces better results than previously published schedulers – more general in the sense that it uses a highly parameterized specification of the target platform, and better in the sense that it produces shorter schedules. Furthermore, the

scheduler is available as open source.

- This report presents and explores a novel idea that aims at minimizing the number of slots in a period of the generated TDM schedule. The goal is to reduce the size of the schedule-tables (that can be very large) such that they fit into the memory structures that are implemented in the hardware platform (that we assume is given). Our results show that it is possible to compress the schedules to around 100 slots for actual benchmarks, with only a negligible increase of the frequency of the TDM clock.

The report is organized as follows. In Chapter 2 we review related work. In Chapter 3 we discuss and identify the details of the scheduling problem. Following this, Chapter 4 presents the design and implementation of the scheduler. A description of the benchmarks used in our experiments is given in Chapter 5. Results from a range of benchmarks are prechapterin Chapter 6 and discussed in Chapter 7. Finally, Chapter 7.1 concludes the report.

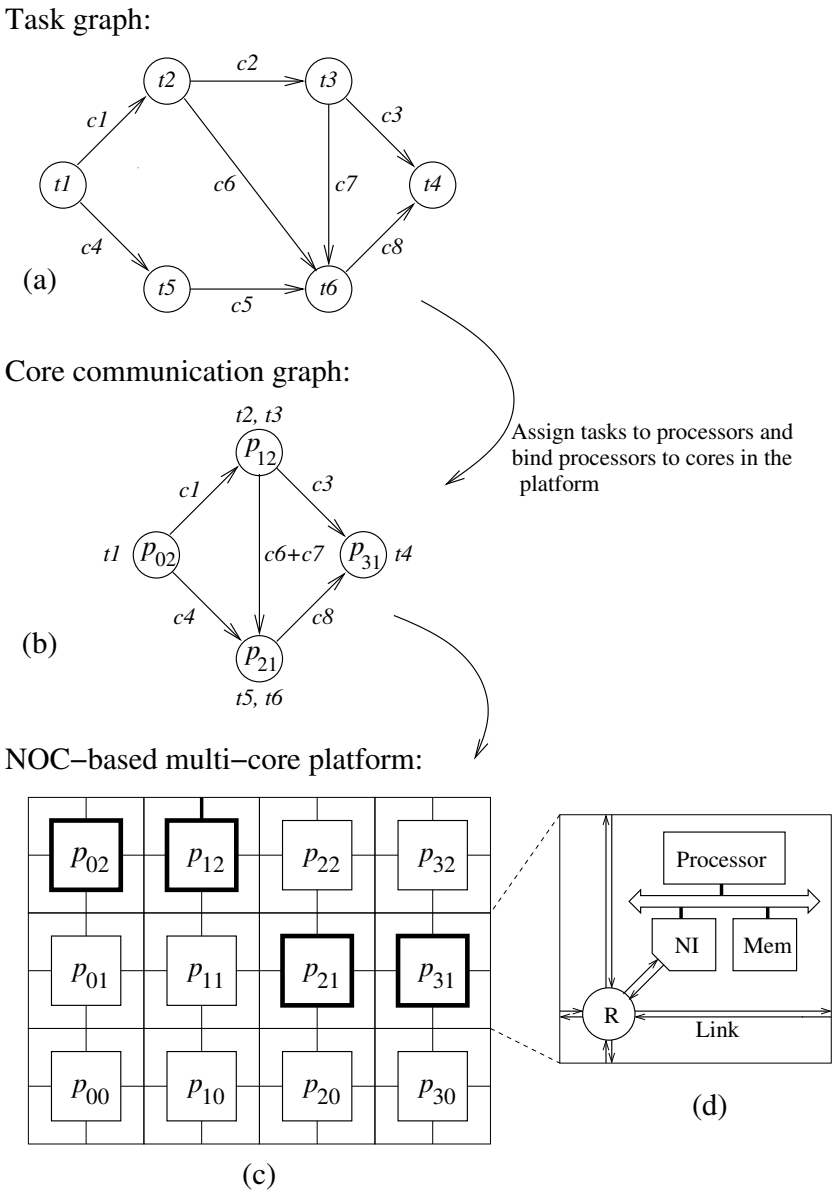

Figure 1.1: Mapping of an application onto a multi-core platform: (a) Task graph for application, (b) core communication graph, (c) multi-core platform, and (d) details of a node in the platform (router (R), links, network interface (NI), processor and local memory).

Chapter 2

Related work

The UMARS scheduler described in [12] can generate TDM schedules for the Æthereal platform. An application-specific platform is generated along with the schedule. This allows the scheduler to modify the topology of the NOC as well as the pipeline depth of the links in the NOC in order to obtain feasible schedules that satisfy the bandwidth requirements. The UMARS scheduler operates in two phases: path allocation and TDM-slot allocation. The path allocation phase is an all-pair shortest-path search, identifying feasible paths on which each communication channel can be routed. The TDM-slot allocation is a mapping of the allocated paths to TDM slots, in such a way that collisions are avoided. The UMARS scheduler works at the level of 3-word flits/packets and pipelining of routers and links is done in multiples of three. Our work is different in several respects: (a) it addresses the communication mapping of an application onto a *given* platform, (b) it works at the level of 1-word phits, and (c) it supports any degree of pipelining in links and routers.

Nostrum is another NOC that is based on TDM scheduling [13]. The scheduler supports regular topologies (mesh, torus, etc.) and it operates using the same two phases (path selection and slot allocation) that are used in the UMARS scheduler. Compared to a conventional packet-switched NOC, Nostrum is a bit more elaborated. It uses temporal-disjoint networks and looped containers (like multiple slotted rings on top of an underlying mesh-style topology). The number of virtual circuits through a router is limited to the number of temporally disjoint networks.

The mapping and scheduling of fully connected core communication graphs that offer virtual circuits with identical bandwidth between all pairs of processor nodes onto platforms with regular NOC topologies (mesh, torus, tree, etc.) is studied in [14]. The paper provides a number of theoretical lower bounds on the schedule length and it presents an ILP-based scheduler that

produces optimal schedules, in terms of schedule length. A very high run-time limits the scheduler to platforms with a small or modest number of nodes – for a platform with 25 nodes the run-time is reported to be 2 weeks.

The same problem – scheduling of all-to-all communication in platforms using NOCs with regular and squared topologies (mesh, torus and bitorus) – is studied in [15]. In order to be able to handle larger problems, a heuristic scheduler is used. A unique aspect of this work is that the routing is identical in all routers. This may allow sharing of routing tables in routers in technologies where this is possible (e.g. FPGAs).

The provisioning of all-to-all communication means that the same platform and the same schedule can be used by different applications, but it also results in rather large schedule periods. In a platform with m^2 nodes the schedule length grows by at least $\mathcal{O}(m^3)$; indicated by the bisection bound stated in [14]. In addition, many applications have sparsely connected core communication graphs. For these reasons it is desirable to be able to handle arbitrary core communication graphs as well as platforms with a range of NOC topologies. This is what we address in this report.

The work presented in this report differs from the above in several respects. Our scheduler is not limited to symmetric all-to-all core communication graphs, it targets a given platform, and it accepts a highly parameterized specification of the TDM-based NOC used in the platform (NOC-topology, pipeline depth of routers and links, different packet lengths on different channels in the core communication graph, etc.). The current version of the scheduler supports regular topologies like 2D-mesh, bi-torus, etc., but it is straightforward to specify other topologies including irregular ones. Furthermore the scheduler considers the above-mentioned parameters. All in all this makes the scheduler far more generic than previously published work. Last but not least, our work reveals some interesting insight into the period of the TDM schedules and ways to reduce the period.

Chapter 3

The Scheduling Problem

The task of the scheduler is to schedule and route the communication channels of the core communication graph onto the TDM-based NOC in a multiprocessor platform. The nodes in a core communication graph are processor cores and the edges are communication channels annotated with bandwidth requirements. The NOC must be configured to implement the communication channels.

Definition 1 *An application has the directed core communication graph $A(P, C)$, where P is the set of processors and C is the set of communication channels.*

Definition 2 *A communication channel $c \in C$ is the triple (p_{src}, p_{dest}, b), where $p_{src} \in P$ is the sending processor, $p_{dest} \in P$ is the receiving processor and $b \in \mathbb{R}$ is the required bandwidth in MB/s.*

The aim of the scheduling is to avoid situations where multiple packets compete for the same resource, i.e. a link in the NOC or an output port of the router. This requires a common time reference (e.g. a clock signal) that defines the time slots that are the basis for the scheduling. In the following we call this the TDM clock. A packet consists of a sequence of data-words that is sent in a corresponding sequence of TDM-clock cycles. The routers and links in the NOC are typically pipelined, which has to be considered when scheduling the traffic.

The TDM clock should be seen as variable parameter that can be set for a given application. In most situations, the bandwidth required by the application does not need the NOC to run at its maximum clock frequency. This allows the use of a TDM clock with a lower frequency.

Input to the scheduler is a specification of the application and a specification of the platform. The application is specified by a core communication graph as explained in Chapter 1 and Figure 1.1.

A platform specification describes the platform on which to route the communication from the core communication graph. The platform specification describes routes and links, as explained in Chapter 1, the routers and links have multiple parameters. In the following definitions we assume that pipeline depths for routers and links are 1 and 0, respectively, in order to keep the definitions simple. It is also possible to specify the packet size for each individual channel in the core communication graph, but in most cases the entire NOC will use only one packet size.

Definition 3 *A platform specification is the directed graph $T(N, L)$, where N is the set of nodes in the platform and L is the set of directed physical links connecting two unique nodes. A node $n \in N$ consists of a router connected to a local processor. A directed physical link $l \in L$ is the tuple (n_{src}, n_{dest}), where $src, dest \in \mathbb{N}^2$, i.e., two-dimensional Cartesian coordinates.*

Given a core communication graph and a platform specification, the task of the scheduler is to determine the routing and scheduling of the data traffic in the NOC. Figure 3.1 shows a close-up of Figure 1.1(c) and illustrates the routing and scheduling of traffic out of processor core p_{02}, i.e., channels c1 and c4 in the core communication graph shown in Figure 1.1(b). The scheduler allows a channel in the core communication graph to be provided by multiple communication paths through the NOC. To ensure that packets arrive in order, the scheduler allows only shortest-path communication channels. For channel c1 there is only one option, as processor cores p_{02} and p_{12} are direct neighbors (connected by one link between routers in tiles 02 and 12). For channel c4, three different paths marked c4', c4' and c4"' may be used. This assumes that the total pipeline depth from p_{02} to p_{12} is the same along all three paths (to ensure in-order delivery of packets). The ports connecting p_{02} and p_{21} to their respective router must be able to carry all the data traffic corresponding to channels c1 and c4.

A communication path is the route of one packet traversing the NOC from its source to its destination. The communication path consists of a sequence of neighboring scheduled links in consecutive time slots. A scheduled link is a physical link together with a time slot.

Definition 4 *A scheduled link $s \in S$ is the pair (l, t), where $l \in L$ is the physical link considered at time slot $t \in \mathbb{N}_0$.*

Definition 5 *A communication path is the vector $\vec{\phi} = \langle s_0, s_1, \ldots, s_{\text{last}} \rangle \in \Phi$ of scheduled links $s \in S$ with $\vec{\phi}_{[i]}.t + 1 = \vec{\phi}_{[i+1]}.t$ and $\vec{\phi}_{[i]}.l.p_{dest} = \vec{\phi}_{[i+1]}.l.p_{src}$. Φ is the set of all possible communication paths.*

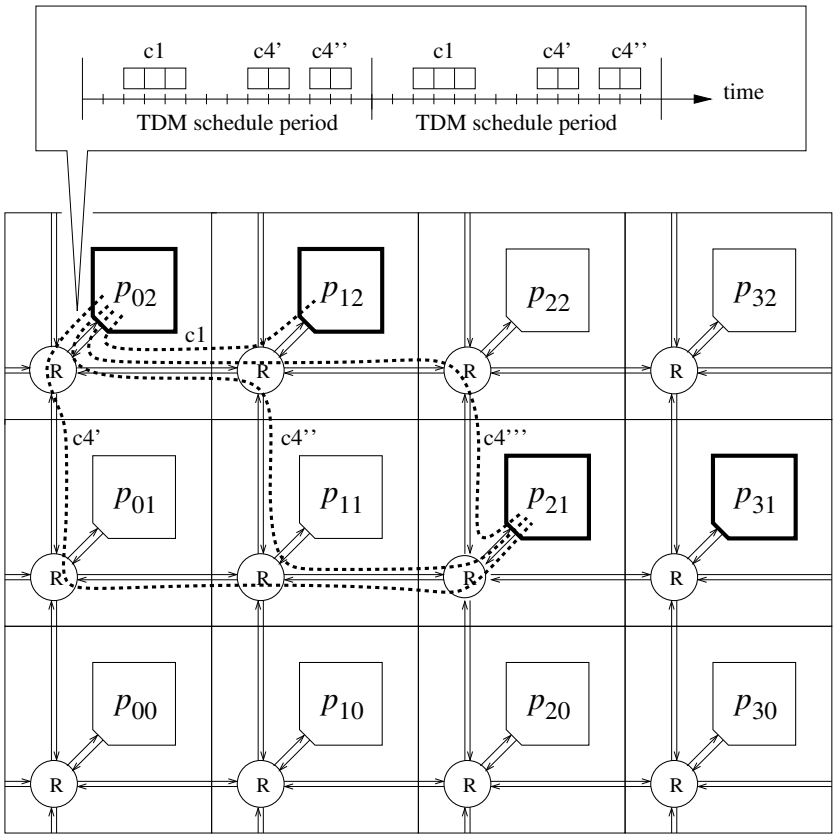

Figure 3.1: Possible routing and scheduling of traffic out of processor core p_{02} (channels c1 and c4 from figure 1.1(b). Several or all of the paths marked c4', c4" and c4"' may be used to implement channel c4. In the schedule shown, only paths c4' and c4" are used.

When the scheduler schedules the communication channels, it needs to know how many packets it needs to schedule in one TDM period. The bandwidth of the communication channels is given in MB/s, this bandwidth needs to be normalized to a number of packets per period. Because we want to minimize the schedule period, we cannot directly convert between MB/s and the number of packets per TDM period. Therefore, we normalize the bandwidth of each communication channel to the bandwidth of the communication channel with the smallest bandwidth requirement. This bandwidth is rounded up to the nearest integer; this (dimensionless) bandwidth is then the required number of packets in one TDM period.

Definition 6 *The normalization of the bandwidth of a communication channel is given by the* norm *function.*

$$\text{norm} : b \mapsto \left\lceil \frac{b}{\min\limits_{c_m \in C} c_m.b} \right\rceil$$

A valid schedule is a specific set of communication paths that satisfy the normalized bandwidth requirements of the core communication graph. In a valid schedule, no link must be part of two different communication paths in the same time slot. The objective of the scheduler is to minimize the TDM period. This minimization may be exploited to decrease latency (and increase bandwidth), or to lower the frequency of the TDM clock (while preserving the bandwidth).

Definition 7 *A schedule is a specific set of paths* $\Phi^* \subset \mathcal{P}(\Phi)$*, where* $\mathcal{P}(\cdot)$ *is the powerset. The schedule period of a schedule is the function* $\text{period} : \Phi^* \mapsto \max\limits_{\vec{\phi} \in \Phi^*} \{\vec{\phi}_{[\text{last}]}.t\}$.

Definition 8 *The scheduler* $\text{sched} : (A, T) \rightarrow \Phi^*$ *maps the channels of* A*, given the platform* T*, to a specific set of communication paths* Φ^**, i.e., a valid schedule.*
Constraint: No overlapping paths $\vec{\phi}_a \neq \vec{\phi}_b : \forall\limits_{i,j} \vec{\phi}_{a[i]} \neq \vec{\phi}_{b[j]}$
Objective: Minimize $\text{period}(\Phi^*)$*, where* Φ^* *is the produced schedule from specific* A^*, T^*.

In the generated schedule, the channel with the smallest bandwidth requirement is scheduled to send exactly one packet and the scheduler aims at generating a schedule with the shortest possible period that satisfies the normalized bandwidth requirements. Based on this schedule period, the minimum frequency of the TDM clock that satisfies the absolute bandwidth requirements is determined.

The normalization of bandwidth requirements *may* result in very long schedules – the channel with the smallest bandwidth requirement is assigned one slot and all other channels are assigned a number of slots corresponding to their normalized bandwidth. As the TDM schedule table for each node has to be implemented in hardware in the NI, it is desirable to limit the number of table entries and therefore to derive schedules with a modest period. Based on previously published hardware designs, we consider 64-128 entries as realistic and 256 entries on the high side. By assigning more bandwidth to the channel with the smallest required bandwidth, the normalized bandwidth of all other channels is reduced, and this results in a reduced TDM-schedule period. In Chapter 6 we show that it is possible to compress the TDM schedules of typical benchmark applications to below 100 slots.

During a physical time window a channel of cause needs the same number of slots and the compressed TDM schedule is simply repeated more times. In many cases the TDM-schedule period is reduced by the same factor as the normalized bandwidths are reduced, and this means that the frequency of the TDM clock is the same. When aggressively compressing the TDM schedule, the NOC may start to saturate, and the schedule period is no longer reduced proportional to the normalized bandwidths. This may be compensated for by increasing the frequency of the TDM clock. This schedule compression is an important contribution of the report.

As the platform is given, there is no guarantee that the required absolute bandwidths (for example MB/s) can be supported by the platform – some links in the NOC may not have enough bandwidth even when the TDM clock is running with the maximum frequency. To accept a produced schedule, we need to verify that it provides enough bandwidth on the specified platform. If the inequality in equation 3.1 holds, the produced schedule provides more than the required bandwidth from the core communication graph.

$$B_{req} < \frac{B_{sched}}{\text{period}} \cdot D_p \cdot f_{max} \tag{3.1}$$

In equation 3.1 B_{req} is the maximum bandwidth of any channel in the core communication graph in MB/s, and B_{sched} is the number of time slots allocated to the communication channel with the largest bandwidth requirement in one TDM-schedule period. period is number of slots in a TDM period, f_{max} is the maximum operating frequency of the NOC, and D_p is the number of bytes that can be transferred in one time slot.

The inequality only needs to be verified for the communication channel with the largest required bandwidth, because the remaining communication channels are assigned more than the required bandwidth.

Chapter 4

The metaheuristic scheduler

Our metaheuristic scheduler produces a schedule that satisfies the given normalized bandwidth requirements while minimizing the TDM schedule period. It implements two different metaheuristic algorithms. In this section we give a brief outline of the metaheuristic algorithms that we have used and their implementation.

4.1 Metaheuristics

A metaheuristic is a high-level optimization strategy that can be used to explore large search spaces. The non-problem-specific metaheuristics guide the search process and the search process is usually non-deterministic [16]. Metaheuristics are used to find good, but not guaranteed optimal, solutions to NP-hard problems.

The two metaheuristics we have implemented are Greedy Randomized Adaptive Search Procedure (GRASP) [17] and Adaptive Large Neighborhood Search (ALNS) [18]. These metaheuristics are chosen because they are well suited for searching very large search spaces. GRASP and ALNS work well for problems with no clear sense of direction, as opposed to the metaheuristic TABU search [19, Chapter 6], which saves the path in the solution space that it has already searched to avoid going back to an already visited solution.

The GRASP pseudo code is shown in Algorithm 1. GRASP creates a greedy randomized initial solution and tries to improve it through a local search, until it finds a local optimum. This local search is performed by selecting an operator from the operator table. Each entry in the operator table is an operator and the probability of the given operator being selected; the sum of probabilities in the operator table is 1. The probabilities in the operator table are updated after each of the iterations, depending on the

Algorithm 1 GRASP(A, T) – Pseudo code for the GRASP metaheuristic.

```
Require:
    A: the normalized core communication graph
    T: the platform specification
 1: Best ← initialSolution(A, T)
 2: operator ← OperatorTable.select()
 3: Best.localSearch(operator)
 4: while Time left do
 5:    Solution ← initialSolution(A, T)
 6:    operator ← OperatorTable.select()
 7:    Solution.localSearch(operator)
 8:    if eval(Solution) better eval(Best) then
 9:       Best ← Solution
10:    OperatorTable.update()
11: return Best
```

results of the performed local search. This process of creating an initial solution and improving it is then repeated for a given amount of time. In each of the iterations, the best solution is updated if the current solution is better.

The ALNS pseudo code is shown in Algorithm 2. ALNS creates an initial solution that satisfies the normalized core communication graph. Then part of the solution is destroyed and repaired and this process is repeated. Therefore, the initial solution used might heavily influence the result. During the design and programming of the scheduler, a number of methods for generating initial solutions were tested, and they are described in Section 4.2. In the destroy function, the operator is chosen probabilistically, and the probabilities are updated according to the improvements of the different operators in each of the iterations. The operators select which paths to destroy; after the paths are destroyed the same paths are repaired in a random greedy fashion. This is done for a given amount of time, and the globally best-seen solution is saved.

For both ALNS and GRASP, the operators are chosen adaptively to ensure that the best operator for the problem is used. The jspdensity?? density of the schedule might influence which operator is best. We calculate the ratio, r, between the period of the previous schedule and the new current schedule. To allow ALNS to move away from local minima, we want the probability of choosing a given operator to converge slowly, so we multiply by $\sqrt{r}$ rather than r. This slow convergence is similar to setting a higher temperature in simulated annealing [19, Chapter 7]. If $\sqrt{r} > 1$, the current solution is

Algorithm 2 ALNS(A, T) – Pseudo code for the ALNS metaheuristic.

Require:
A: the normalized core communication graph
T: the platform specification

```
1: Current ← initialSolution(A, T)
2: Best ← Current
3: while Time left do
4:    operator ← OperatorTable.select()
5:    Current.destroy(Current)
6:    Current.repair(Current)
7:    if eval(Current) better eval(Best) then
8:       Best ← Current
9:    OperatorTable.Update()
10: return Best
```

shorter than the previous, so we increase the probability of the same operator being selected in the next iteration. The operators are explained in more detail in Section 4.3.

The metaheuristic algorithms continuously try to minimize the number of slots in the TDM schedule; therefore a user should allow the scheduler to run for as long as can be afforded. Since the schedule is generated at compile time, it is affordable to let it run for several hours.

4.2 Generating initial solutions for GRASP and ALNS

The initial solutions are built to satisfy the bandwidth requirements of the normalized core communication graph; they are built using a greedy algorithm with an adjustable degree of randomness. We have applied randomization for the two types of routing decisions in the algorithm. The first is the decision of which output port to choose when a communication path is routed. The second is the order in which the communication paths are routed. In the deterministic case, the algorithm sorts the set of unscheduled paths by the length from its source to its destination. The sorted set of communication paths is then placed in the schedule one at a time, always picking the longest remaining channel. In any case, a communication path is routed in the earliest possible time slot. We have implemented the algorithm with the following combinations of randomization:

1. Deterministic, no randomization.

2. Randomization of choosing the next output port.

3. Randomization of both the next output port and order of paths.

4. (Only for GRASP) Takes parameter β, the percentage of paths to be swapped in the sorted set of unscheduled paths. If $\beta = 0$, the behavior is equal to the second combination of randomization. If $\beta = 1$ the behavior is equal to the third combination of randomization.

Good values for beta have been found by running the algorithm on many different problems of different sizes with a wide range of β values. For mesh topologies, we found 0.2 to be a good value and for bi-torus we found 0.02 to be a good value.

4.3 Operators

In this section we discuss which changes to a solution can lead to optimizations, and we select which operations to implement and use in the scheduler. In order to decrease the period of an existing solution, the end of the schedule should be moved backwards. Intuition says that an existing solution generated by the scheduler is more dense in the beginning than in the end, therefore there is more freedom to reroute paths in the end of the schedule. After each of the optimization iterations, all communication paths are routed completely. The quality of a solution is measured by the period of the TDM schedule.

Definition 9 *The dominating paths are the set*
$\mathrm{D} = \{\vec{\phi} \in \Phi^* : \vec{\phi}_{[last]}.t = \mathrm{period}(\Phi^*)\}$

The dominating paths are the direct cause preventing a shorter schedule. Removing complete communication paths and rerouting them one at a time will not reduce the number of slots in the TDM schedule. Removing a collection of communication paths that prevent each other from being routed earlier might lead to a shorter schedule period when they are rerouted randomly. To make adaptive decisions on which operators to choose, we need a set of diverse operators to choose from. We have implemented the operators: *Dominating paths*, *Dominating rectangle*, *Late paths* and *Random*.

Definition 10 *Three basic selection functions:*
$\mathrm{links} : \vec{\phi} \mapsto \{0 \leq i \leq \vec{\phi}_{[\mathrm{last}]} : \vec{\phi}_{[i]}.l\}$
$\mathrm{touches} : L^*, \Phi^* \mapsto \{\vec{\phi} \in \Phi^* : \mathrm{links}(\vec{\phi}) \cap L^* \neq \emptyset\}$
$\mathrm{rect} : \vec{\phi} \to \mathcal{P}(L)$ *returns the links within the bounding box spanned by* $\vec{\phi}$.

The dominating paths operator selects the dominating paths and the paths that are routed on the same physical links as the dominating paths, no matter which time slot they are routed in.

Definition 11 *The dominating paths operator selects the set* DP *of paths to reroute, where*

$$\mathrm{DP} = \mathrm{touches}\left(\bigcup_{\vec{\phi}\in \mathrm{D}} \mathrm{links}(\vec{\phi}), \Phi^*\right)$$

The dominating rectangle operator selects all paths that are routed on the physical links in the bounding box of each dominating path, no matter which time slot they are routed in.

Definition 12 *The dominating rectangle operator selects the set* DR *of paths to reroute, where*

$$\mathrm{DR} = \mathrm{touches}\left(\bigcup_{\vec{\phi}\in \mathrm{D}} \mathrm{rect}(\vec{\phi}), \Phi^*\right)$$

The late paths operator selects the paths that end in the last time slot (the dominating paths) and the paths that end in the second-last time slot.

Definition 13 *The late paths operator selects the set*
$\mathrm{DL} = \{\vec{\phi} \in \Phi^* : \vec{\phi}_{[last]}.t \geq \mathrm{period}(\Phi^*) - 1\}$

The Random operator selects a random-sized set of randomly selected paths. At least two paths are always selected and up to 10 % of all existing paths can be selected.

4.4 Implementation

Our scheduler is written in C++11, using BOOST 1.49[20] and pugixml 1.0* as 3rd party libraries. The source code of our scheduler can be downloaded at `https://github.com/t-crest/poseidon.git`. The source code builds successfully on Linux and Cygwin. The scheduler reads an input XML file describing the core communication graph of the application and the platform specification. The communication is then scheduled and the resulting schedule is written to an output XML file.

*Source code can be downloaded at `http://pugixml.googlecode.com/files/pugixml-1.0.zip`

Table 4.1: Comparison of the schedule length for the all-to-all case. The table shows the results produced by the symmetric heuristic scheduler [15], the lower bounds from [14], the optimal solutions from [14] and the results we have obtained by the greedy approach, the ALNS approach and the GRASP approach. The schedule lengths are expressed in TDM slots, and the numbers in bold are the best results for the given topology and network size.

Size	Previous work			This work		
	Opt. [14]	Lower bound[14]	Sym. [15]	GREEDY	ALNS	GRASP
	Mesh					
3×3	10	8	28	13	**11**	**11**
4×4	18	16	59	24	**21**	**21**
5×5	34	25	112	41	39	**37**
6×6	–	54	–	66	65	**61**
7×7	–	66	–	98	98	**95**
8×8	–	128	481	144	143	**139**
9×9	–	135	–	201	201	**195**
10×10	–	250	974	271	271	**267**
15×15	–	600	3467	**886**	**886**	900
	Bitorus					
3×3	10	8	11	12	**10**	**10**
4×4	18	15	20	21	**19**	**19**
5×5	28	24	**28**	32	30	30
6×6	–	35	–	45	45	**43**
7×7	–	48	–	64	63	**61**
8×8	–	64	88	87	87	**85**
9×9	–	90	–	**113**	**113**	**113**
10×10	–	125	158	154	154	**151**
15×15	–	420	481	**471**	**471**	477

From the platform specification the scheduler generates a data structure with a notion of time slots in which it can schedule the communication channels from the core communication graph. For each communication channel, the scheduler schedules a communication path for each normalized unit of bandwidth.

The communication channels that is to be scheduled contends for the links in the network. Thus it is intuitive to have data related to each link and router represented only once. This means we use a distributed data structure that stores scheduling information for each link.

One of our simplifying decisions is to only route channels along shortest paths, as this will automatically guarantee in-order packet arrival. For this, we need to compute local routing decisions for all pairs of processors. Since our network graph is very regular, as each router only has a maximum out-degree of four and all links have unit cost, we can easily find the shortest path via a breadth-first search. We simply start at the destination and do a BFS. Computing all these shortest paths can be done in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space.

Chapter 5

Benchmarks

For the benchmarks in this report, we assume that our platform is either a square mesh or a square bi-torus. In addition we assume that the pipeline depth of all routers is one, that the pipeline depth of all links is zero, and that the packet size is one data word. In this way a packet can traverse a router in one clock cycle.

We will experiment with two different types of benchmarks. The first is the special case of all-to-all communication where core communication graphs are fully connected graphs with a bandwidth of one on each channel. The second type is the more general case of application-specific schedules from the MCSL benchmarks suite[21]. In all our experiments, we create a core communication graph and a platform specification. For the platform specification we only change the topology and the network size.

In the MCSL benchmark suite the tasks are already mapped onto processors, so the benchmarks are basically core communication graphs. As some processors execute more tasks, and as these tasks may have different communication behaviors, we associate the maximum data rate as the required bandwidth for the different channels in the core communication graph. The required bandwidth on the communication channels of the core communication graph can be calculated using the following equation:

$$c_i = \max_{t \in \tau} \alpha_i(t),\ c_i \in C$$

where $\alpha_i(t)$ is the data rate between the processors $c_i.p_{src}$ and $c_i.p_{dest}$ at time t, and τ is the time interval when the application is executing. The normalized core communication graph is found as described in Section 3.

If the produced schedule is longer than the number of time slots supported by the hardware platform, the schedule needs to be shortened/compressed. One way to achieve this is to normalize the bandwidths with a larger con-

stant than the smallest bandwidth. Another normalization function with a normalization factor as a parameter can be seen in Equation 5.1. The normalization function normf takes the bandwidth and the new normalization factor as input parameters.

$$\text{normf} : b, \sigma \mapsto \left\lceil \frac{b}{\sigma \min\limits_{c_m \in C} c_m.b} \right\rceil \tag{5.1}$$

Where, σ is the normalization factor. Normalizing with a large σ degrades the relationship between bandwidths, because the minimum bandwidth is always 1. As σ increases at some point the overall performance drops. Taken to the extreme, the shortest possible schedule is when we normalize with a factor equal to the bandwidth of the largest communication channel. This results in a normalized core communication graph where all channels have a required bandwidth of one packet per TDM period. The TDM period needs to be repeated σ times to provide the same amount of data to be transferred as the uncompressed schedule.

Chapter 6

Results

The scheduler always produces a solution that satisfies the bandwidth requirements of the normalized core communication graph, and the goal of the scheduler is to minimize the schedule period. This again minimizes the frequency of the TDM clock at which the absolute required bandwidths are met. Below we present results for the all-to-all communication and for the core communication graphs derived from the MCSL benchmark applications.

6.1 All-to-All Communication

To evaluate the performance of our scheduler in the special case of all-to-all communication, we schedule the communication patterns for different network sizes on both mesh and bi-torus topologies. Table 4.1 shows the results and compares against the results of the heuristic scheduler in [15] and against the optimal results and theoretical lower bounds given in [14]. All our results have been obtained by running our metaheuristic scheduler for two hours on a computer with an i7-3630QM (4 cores @ 2.4 GHz) with 16 GB of memory.

As seen in Table 4.1 our scheduler produces better results than the symmetric scheduler [15] in all cases except the 5×5 bi-torus. This conforms to the fact that the symmetric scheduler is restricted to produce solutions where all routers execute the same schedule, whereas our scheduler has more freedom and hence is able to find solutions with a shorter schedule period.

For the bi-torus our schedules are approx. 30 % longer than the analytical lower bound from [14]. As seen in Table 4.1, the optimal schedule period for a 3×3 bi-torus network is 10, and both our ALNS and GRASP schedulers are able to find schedules with this period. For the other cases for which optimal schedules are known, our scheduler finds solutions whose schedule

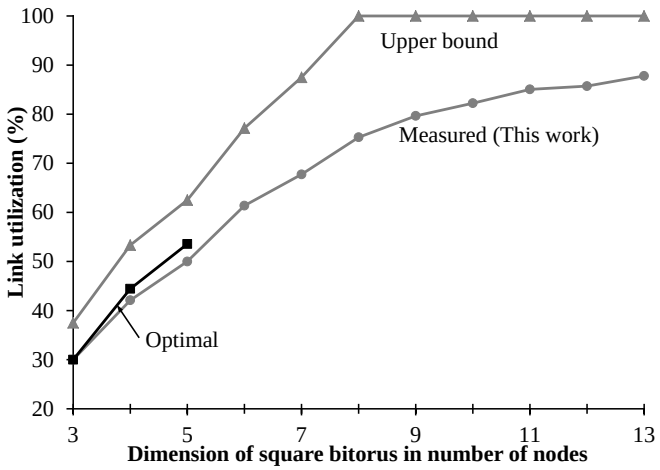

Figure 6.1: The link utilization of a greedy all-to-all schedule on a network with a bi-torus topology. The optimal link utilization is calculated from[22], along with the upper bound on the link utilization that is derived from the lower bound of the schedule period.

periods are only slightly larger.

Comparing the GREEDY solutions with the metaheuristic solutions (ALNS and GRASL) produced in Table 4.1, we see that the metaheuristic algorithms produce better results in most cases. For the large network sizes, the improvement by the metaheuristics diminishes. There are several reasons for this. Firstly the link utilization increases with the number of nodes as seen in Figure 6.1 for the bi-torus and the greedy scheduler. The link utilization increases because the average communication channel length grows with the network size. This limits the ability to reroute paths.

6.2 Application-Specific Schedules

In this section we investigate the general case of scheduling arbitrary communication patterns. The communication patterns of interest are communication graphs from real applications. The MCSL NOC Benchmark Suite [21] provides statistical traffic patterns for seven different applications mapped onto different topologies of different sizes. The seven benchmarks represent different types of traffic patterns, such as one-to-many, many-to-many, grid-like patterns and combinations of these. These communication patterns are also valid for real-time systems, as they contain a dynamic control applica-

tion, encoding and decoding of error-correcting codes and general mathematical operations.

The normalized communication patterns are scheduled with the presented scheduler on mesh and bi-torus topologies of different sizes. In Table 6.1 we show the obtained schedule period of the benchmark applications generated with ALNS and GRASP.

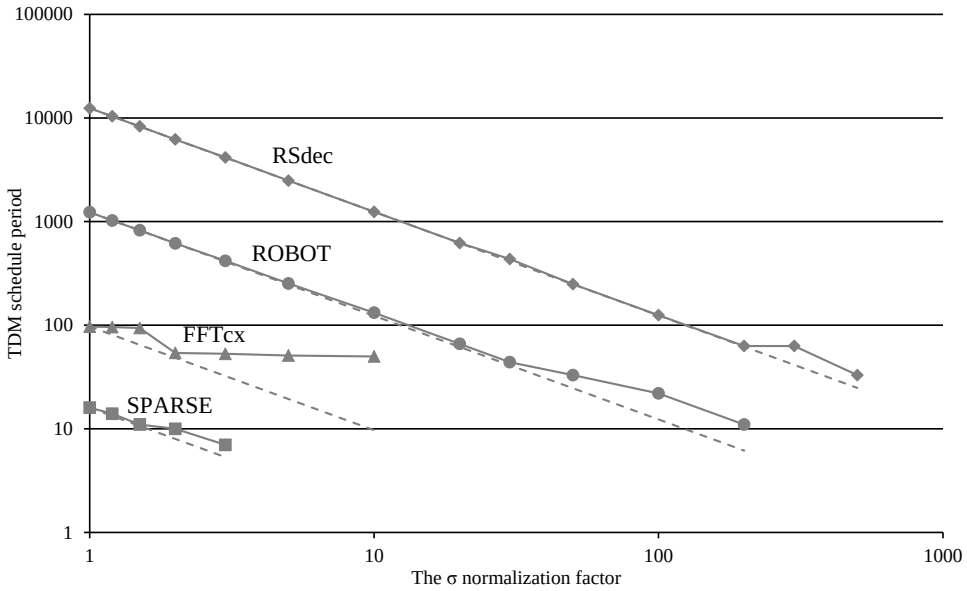

Figure 6.2: The measured and ideal schedule period as a function of σ of all the benchmark applications mapped on an 8×8 bi-torus platform.

The long schedules shown in Table 6.1 are bound by IO of the most communication-intensive processors in the network. Another observation that can be made from the table is the irregularity of the schedule lengths. There is no correlation between the schedule period of an application mapped to a mesh topology or a bi-torus topology of the same size.

In practice it is not feasible to implement the hardware tables needed to support the longest schedules shown in Table 6.1 (1000+ time slots). Based on the hardware complexity of the NOC-implementation, we consider RAM or ROM tables with 64-128 entries to be acceptable and tables with 256 entries to be a on the high side.

Therefore, it is interesting how much the schedule period can be compressed before the overall performance decreases. Varying the normalization factor, we can observe how the resulting schedule periods change. In Figure 6.2 we show how the schedule period changes as a function of the normalization factor σ for the benchmark applications in an 8×8 bi-torus platform.

We picked this topology because it reflects a likely topology. We have also experimented with a 16×16 bi-torus platform, where we found similar results. The dotted lines are the initial schedule period divided by the increasing σ. They indicate how the schedule period would decrease in the ideal case, where bandwidths are not affected (c.f., the discussion in Chapter 3). If the measurements are above the dotted line, the schedule period is longer than what corresponds to the compression factor. This represents a decrease in bandwidth (in the normalized domain) and this must be compensated for by an increase in the frequency of the TDM clock.

We see that the curves for all the applications follow the ideal lines well below 1000 time slots. When the curves break off from the ideal lines, this is because the increasing compression factor causes an over-allocation of bandwidth to the communication channels with the smallest requirements. It is seen that all applications can be scaled down to less than 100 TDM slots with a negligible performance degradation compared to the unscaled version. In most cases this can be compensated for by increasing the frequency of the TDM clock used in the NOC. We consider this insight and the idea of compressing the schedules an important contribution of the report.

Table 6.1: The number of TDM slots in a period for the MCSL benchmark applications [21]. The applications are: (1) Reed-Solomon encoding, (2) Reed-Solomon decoding, (3) ROBOT Dynamic control problem, (4) SPEC95 multi-electron derivatives, (5) 1024 point Fast Fourier transform of complex numbers, (6) Sparse matrix multiplication, (7) H.264 720p video decoding. The results are generated by the ALNS and GRASP metaheuristics, only the shortest schedule period is shown. Unmarked numbers mean that ALNS-generated and GRASP-generated schedules of the same length in time slots. Numbers marked with a *and a †are generated by ALNS and GRASP, respectively.

Network size	(1)	(2)	(3)	(4)	(5)	(6)	(7)
				Mesh			
3×3	197	†4,249	529	75	29	†17	473
4×4	91	7,951	1,009	114	43	†15	886
5×5	91	*12,723	1,345	145	†82	19	*1,422
6×6	91	†22,492	1,345	*519	†112	15	2,066
7×7	91	22,584	1,345	†650	†143	20	†2,833
8×8	91	†16,431	1,345	†1,571	†129	14	†3,718
9×9	90	22,497	1,344	†630	†183	13	–
10×10	80	16,536	1,344	*1,374	†183	†14	–
11×11	80	†16,431	1,344	†601	†197	†15	–
12×12	†80	22,476	1,344	514	†182	†15	–
13×13	†85	22,476	1,344	594	†189	†17	–
14×14	†95	22,531	1,344	†1,322	†178	†15	–
15×15	†98	†22,476	1,344	1,622	†189	14	–
16×16	†101	†16,430	1,344	†565	†145	†15	–
				Bi-torus			
3×3	225	*3,202	529	40	24	15	305
4×4	393	7,951	1,248	184	41	16	571
5×5	561	†9,601	1,233	2,306	68	13	†1,417
6×6	161	†12,401	1,233	389	84	†13	1,331
7×7	151	†22,477	1,233	2,018	†99	16	1,825
8×8	151	†12,417	1,233	†1,103	98	17	2,395
9×9	150	†22,476	1,232	1,192	†117	11	–
10×10	140	22,502	1,232	†1,466	†115	12	–
11×11	110	†12,402	1,232	†1,382	*122	†16	–
12×12	392	†23,200	1,232	†1,291	†116	12	–
13×13	420	†23,200	1,232	†718	114	†12	–
14×14	420	†16,960	1,232	†605	113	15	–
15×15	392	†16,960	1,232	†1,585	†111	†13	–
16×16	392	†17,138	1,232	†1,939	88	†13	–

Chapter 7

Discussion

Summarizing the results, we see that our scheduler produces very good results for the special case of all-to-all communication. The all-to-all schedules are hard to optimize because their link utilization is very high and their core communication graph is fully connected. When looking at the results of the all-to-all schedules, we can see that the schedule period length grows polynomially with the number of nodes in the network. For large networks it might not be feasible to support an all-to-all schedule. First of all, the latency is very high and the bandwidth to a single core is very limited. Secondly, the tables would be very large, increasing the size of the interconnect hardware. Finally, an application mapped onto a 16×16 platform, where all cores need to communicate to all other cores, is quite unlikely. In the tool flow, an all-to-all schedule would lead to simplifications. The mapping of tasks to processors is simplified because the bandwidth is equally low to all cores. Therefore, the mapping of tasks has very little effect on the performance, only the latency changes depending on the mapping.

From the application-specific schedules of the benchmark application, it looks plausible that we can set a limit on the number of time slots that a general-purpose platform needs to support. For the applications in the benchmark suite it seems a good limit on the number of time slots is around 100. If we limit the number of time slots to 100, we can schedule all of the applications from the benchmark with only a negligible performance decrease compared to the unlimited case. Given a limit on the time slots available in the hardware platform, the tool flow should do a binary search for the optimal normalization factor.

Overall the application-specific schedules provide an excessive amount of bandwidth compared to the all-to-all schedules. The excessive amount of bandwidth can be removed by reducing the clock frequency of the network or reducing the width of the links, a combination of the two will save both

power and area.

7.1 Conclusion

The report presented a metaheuristic scheduler for inter-processor communication in multi-core platforms using time division multiplexed (TDM) networks on chip (NOC). This scheduling problem is NP-complete and we use a metaheuristic approach to solve it. Our scheduler supports two approaches: a greedy randomized adaptive search procedure and an adaptive large neighborhood search. The scheduler is intended for use in a design flow where an application is mapped onto a pre-designed and therefore fixed platform. Input to the scheduler is a specification of the NOC in the platform and a specification of the application in the form of a core communication graph (including the binding of nodes in this graph to processors in the platform).

The input formats used to specify the core communication graph of the application, and the NOC used in the platform, are highly parameterized and allow specification of graphs of arbitrary topologies. For the NOC, it is possible to specify the pipeline depth in the routers and in the individual links. For the application it is possible to specify a packet size for each channel. Compared to previous work, the scheduler therefore handles a broader and more general class of applications and a broader and more general class of platforms.

For the special case of all-to-all communication with identical bandwidth, our scheduler produces better results than reported in previous work. The scheduler was also evaluated for a set of larger and non-symmetric applications from the MCSL NOC benchmark suite. Among our results here is the observation that our metaheuristics perform better than the greedy solution.

The scheduler uses a dimensionless and normalized representation of bandwidth requirements, and the report shows that the period of the TDM schedule can be reduced by assigning excess bandwidth to channels requiring little bandwidth. For the MCSL NOC benchmarks, the schedule lengths can in many cases be compressed to less than 100 TDM slots with close to no increase in the TDM-clock frequency.

Bibliography

[1] W. Dally, "Route packets, not wires: On-Chip interconnection networks," in *Proceedings of the 2001 Design Automation Conference (DAC-01)*. New York: ACM Press, Jun. 2001, pp. 684–689.

[2] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[3] K. Goossens and A. Hansson, "The AEthereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC 2010)*, 2010, pp. 306 –311.

[4] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, no. 10, pp. 1374–1396, 1995.

[5] J. Sparsø, "Networks-on-chip for real-time multi-processor systems-on-chip," in *Proc. International Conference on Application of Concurrency to System Design (ACSD)*, Jun. 2012, pp. 1–5.

[6] S. Hansen, "T-crest project," 2012, project webpage http://t-crest.org. [Online]. Available: http://t-crest.org

[7] J. Sparsø, E. Kasapaki, and M. Schoeberl, "An Area-efficient Network Interface for a TDM-based Network-on-Chip," in *Proc. Design Automation and Test in Europe (DATE)*, 2013, pp. 1044–1047.

[8] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, Grenoble, France, March 2011, pp. 11–20.

[9] S. Murali, G. D. Micheli, A. Jalabert, and L. Benini, "XpipesCompiler: A tool for instantiating application specific networks-on-chip," in *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, 2004, pp. 884–889.

[10] R. Marculescu, U. Ogras, L.-S. Peh, N. Jerger, and Y. Hoskote, "Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, 2009.

[11] S. Even, A. Itai, and A. Shamir, "On the complexity of time table and multi-commodity flow problems," in *Foundations of Computer Science, 1975., 16th Annual Symposium on*, oct. 1975, pp. 184 –193.

[12] A. Hansson, K. Goossens, and A. Radulescu, "A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic," *VLSI Design*, vol. 2007, p. 16, 2007.

[13] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2, feb. 2004, pp. 890 – 895 Vol.2.

[14] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*. Lyngby, Denmark: IEEE, May 2012.

[15] F. Brandner and M. Schoeberl, "Static routing in symmetric real-time network-on-chips," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12. New York, NY, USA: ACM, 2012, pp. 61–70.

[16] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sep. 2003.

[17] T. A. Feo and M. G. Resende, "Greedy randomized adaptive search procedures," *Journal of Global Optimization*, vol. 6, no. 2, pp. 109–133, March 1995.

[18] D. Pisinger and S. Ropke, "Large neighborhood search," in *Handbook of Metaheuristics*, ser. International Series in Operations Research and

Management Science, M. Gendreau and J.-Y. Potvin, Eds. Springer US, 2010, vol. 146, pp. 399–419.

[19] E. Burke and G. Kendall, *Search methodologies: introductory tutorials in optimization and decision support techniques.* Springer Science+ Business Media, 2005.

[20] J. Siek, L.-Q. Lee, and A. Lumsdaine, "Boost graph library," http://www.boost.org/libs/graph/, June 2000.

[21] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang, "A noc traffic suite based on real applications," in *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, july 2011, pp. 66 –71.

[22] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *Proceedings of the International Symposium on Networks-on-Chip (NOCS).* IEEE, 2012, pp. 152–160.