Resource-Efficient Routing and Scheduling of Time-Constrained Streaming Communication on Networks-on-Chip*

Sander Stuijk, Twan Basten, Marc Geilen, Amir Hossein Ghamarian and Bart Theelen
Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems
s.stuijk@tue.nl

Abstract. Network-on-chip-based multiprocessor systems-on-chip are considered as future embedded systems platforms. One of the steps in mapping an application onto such a parallel platform involves scheduling the communication on the network-on-chip. This paper presents different scheduling strategies that minimize resource usage by exploiting all scheduling freedom offered by networks-on-chip. It also introduces a technique to take the dynamism in applications into account when scheduling the communication of an application on the network-on-chip while minimizing the resource usage. Our experiments show that resource-utilization is improved when compared to existing techniques.

1. Introduction

Increasing computational demands from multimedia applications have led to the development of multi-processor systems-on-chip (MP-SoC) which integrate many processing cores and memories. With the growing number of cores integrated into a chip, communication becomes a bottleneck as traditional communication architectures are inherently non-scalable [3]. Networks-on-Chip (NoC) are emerging as a communication architecture which solves this issue as it provides a better structure and modularity [3, 5, 25]. Furthermore, it can provide guarantees on the timing behavior of the communication. This enables the development of systems with a predictable timing behavior which is key for modern multimedia systems [7].

Current NoCs like Æthereal [25] and Nostrum [18] use circuit-switching to create connections through the NoC which offer timing guarantees. Today's routing and scheduling solutions however (a) often do not use all routing flexibility of NoCs and (b) make bandwidth reservations for connections with throughput/latency guarantees that are unnecessarily conservative. To illustrate the first point, for example, the scheduling strategies presented in [12, 16] restrict themselves to minimal length routes. Modern NoCs allow the use of other, more flexible, routing schemes. As an illustration of the second point, consider a simple NoC

*This work was supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES.

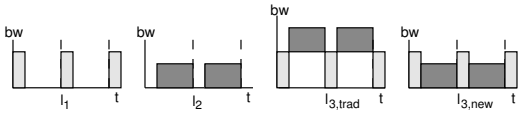


Figure 1. Motivating example.

with three links l_1, l_2 and l_3. The data streams sent over l_1 and l_2, shown in Fig. 1, are both sent over l_3. Traditional NoC scheduling strategies [12, 14] reserve two guaranteed throughput connections on the link ($l_{3,trad}$). However, given the timing of the data streams on l_1 and l_2, it is possible to combine both streams and preserve bandwidth ($l_{3,new}$). The essential idea is not to reserve bandwidth for guaranteed throughput connections permanently during the entire life time of an active data stream but only during certain intervals, typically per communicated message. The use of non-minimal routes and the intelligent reservation of NoC bandwidth leads to a better resource utilization in the NoC.

Modern multimedia applications more and more exhibit dynamism that causes the application to have a number of different communication patterns, called *communication scenarios*. An extension to the techniques of [12, 14] to handle this type of dynamism is presented in [19], which presents a technique to allocate resources for each scenario while guaranteeing that sufficient resources are available when switching between scenarios. It ignores that often information is available on the time needed to switch between the scenarios. This is similar to the situation illustrated in Fig. 1. Using this information, it is possible to minimize the resource usage when switching between scenarios.

This paper explores and compares several new routing and scheduling strategies for data streams that exploit all scheduling freedom offered by modern NoCs and minimize resource usage. It is furthermore shown how communication scenario transitions can be taken into account. Scheduling strategies which minimize resource usage will be able to schedule problems with tighter latency constraints and/or larger bandwidth requirements.

The strategies presented in this paper were first introduced in [27]. This paper present new experimental results

for other topologies than meshes showing the versatility of the approach. The routing and scheduling strategies are improved leading to a better resource utilization of the NoC. Furthermore, this paper formally shows that the routing and scheduling problem is NP-complete. The technique to deal with the dynamism in applications through the use of scenarios is also new in this paper. It is furthermore shown how this technique can be used to schedule the communication patterns resulting from an application specified as a Synchronous Dataflow Graph (SDFG, [17]), which is a commonly used model for multimedia applications.

The remainder of this paper is organized as follows. The next section discusses related work. Sec. 3 presents the application model used for programming NoC-based MP-SoC architectures. The architecture itself is discussed in Sec. 4. The time-constrained scheduling problem is formalized in Sec. 5. An NP-completeness proof of the problem is also provided in this section. Several different scheduling strategies are presented in Sec. 6. The benchmark used to evaluate these strategies is presented in Sec. 7. The experimental results on this benchmark are discussed in Sec. 8. In Sec. 9, a technique is presented to extract multiple communication scenarios from an application described as an SDFG. This technique is used in Sec. 10 to schedule the communication of a realistic multimedia application on a NoC using our scheduling strategies.

2. Related Work

This paper considers scheduling streaming communication on a NoC within given timing constraints while minimizing resource usage. We restrict ourself to communication with timing constraints. In practice, some communication streams in an application may have no timing requirements. Scheduling techniques for these streams are studied in e.g. [24]. Those techniques can be used together with our approach to schedule both the communication without and with timing constraints.

In [16], a state-of-the-art technique is presented to schedule time-constrained communications on a NoC when assuming acyclic, non-streaming communication. That is, tasks communicate at most once with each other. Our application model, presented in Sec. 3, allows modeling of communication streams in which tasks periodically, i.e., repeatedly, communicate with each other.

Scheduling streaming communication with timing guarantees is also studied in [12, 14, 25]. They apply a greedy heuristic and reserve bandwidth for streams, whereas we propose to reserve bandwidth per message and present several different heuristics. Our results show a clear improvement in resource usage. In [19], an extension to [14, 25] is presented to schedule multiple communication patterns onto a single network-on-chip. It assumes that the streams of different communication patterns are independent of each other and no timing relation between them is known. As a result, streams from different patterns cannot share bandwidth. In this paper, we present a technique to share bandwidth between multiple communication scenarios when a timing relation between the scenarios is known.

Many NoCs like Nostrum [18], SPIN [13], Dally and Towles [5] use regular NoC architectures like a mesh, torus or fat-tree. The regular structure of these NoCs fits well with simple routing schemes like XY-routing. These architectures assume that the computational elements connected to the NoC are all of similar size. In practice, existing IP-blocks do not always meet this requirement. Furthermore, applications with irregular communication requirements do not fit well with the regular NoCs. For these reasons, irregular NoC topologies and there accompanying routing and scheduling techniques are studied [15, 20, 21]. In [15] a NoC architecture is studied in which some links from a mesh are removed. A technique to design application-specific NoCs is studied in [20]. [21] studies a mesh-topology with added links that reduce the the long latencies between nodes in the mesh. The routing and scheduling technique presented in this paper can be used in combination with any arbitrary (regular or irregular) NoC topology.

3. Application Modeling

Multimedia applications, for instance an MP3 decoder, operate on streams of data. These applications can be described by an application task graph in which the tasks are executed repeatedly. Whenever a task executes, it exchanges messages with other tasks via (data) streams. Dynamism in the application can cause differences in the time at which tasks consume and produce messages. When the timing difference is small or occurs infrequently, it can be considered as jitter on the communication pattern. To provide timing guarantees, the worst-case communication pattern which includes this jitter must be considered when allocating resources. It is also possible that the dynamism in the application causes changes in the communication pattern which are effective over a longer period of time. An MP3 decoder for example could switch from decoding a stereo stream to a mono stream. This situation could be taken into account by allocating resources for the worst-case communication pattern that can occur. However, this will result in a resource allocation which is too conservative for most situations [10, 19]. The solution to prevent over-allocation of the resources is to consider communication patterns which differ considerably from each other as separate *scenarios*. In an MP3 decoder, for example, the decoding of a stereo and mono stream could be seen as two distinct scenarios. It is possible that a switch from one scenario to another is time-constrained and that the two scenarios overlap for some time. These aspects should be taken into account when allocating resources.

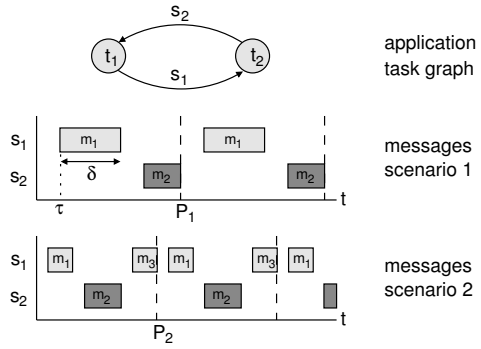


Figure 2. Example application task graph.

Table 1. Scenario overlap during switching

	scenario 1	scenario 2
scenario 1	-	5
scenario 2	3	-

Fig. 2 shows an example of a simple application task graph consisting of two tasks t_1 and t_2. The application contains two scenarios. In the first scenario, task t_1 sends a message m_1 through stream s_1 to task t_2 and t_2 sends a message m_2 through s_2 to t_1. This pattern is repeated with a period P_1. The second scenario has a different communication pattern and period. In the second scenario, task t_1 sends every period P_2 two messages m_1 and m_3 to task t_2 and t_2 sends a message m_2 to t_1. Note that also the time at which messages are sent and size of them is different in the two scenarios. When running, the application may switch from one scenario to another. During this switching period, due to the streaming nature of the applications, multiple scenarios may be active simultaneously. Tab. 3 gives the overlap when switching between the two scenarios of our example application. It shows, for example, that when switching from scenario 1 to scenario 2, both communication patterns overlap for 5 time-units. It is also possible that a switch from one scenario to another scenario cannot occur or happens without overlap.

To meet the computational requirements of modern multimedia applications, multi-processor systems are used. The tasks, from an application graph, are mapped to the various processors in the system. Whenever multiple tasks are mapped to one processor, the execution order of these tasks is fixed through a schedule. These schedules and the timing constraints imposed on the application determine time bounds within which each task must be executed. Similarly, they determine time bounds within which messages must be communicated between the tasks. Identification of different communication scenarios can be done using the technique described in [10]. This paper presents techniques to schedule messages from multiple scenarios, which are specified with time bounds, on the NoC.

4. Architecture Platform

Multiprocessor systems-on-chip, like Daytona [1], Eclipse [26], Hijdra [2], and StepNP [22], use the tile-based multiprocessor template described by Culler [4]. Each *tile* contains one or a few processor cores and local memories. The architecture template used in our work fits also in this template. A network-on-chip (NoC) is used to interconnect the different tiles. Each tile contains a *network interface* (NI) through which it is connected with a single router in the NoC. The *routers* can be connected to each other in an arbitrary topology. The connections between routers and between routers and NIs are called *links*.

In this paper, the connections between the processing elements and the NI inside a tile are ignored. We assume that these connections introduce no delay, or that the delay is already taken into account in the timing constraints imposed on communications, and that there is sufficient bandwidth available. Hence, the NI can be abstracted away into the tile. Given this abstraction, the architecture can be described with the following graph structure.

Definition 1. (ARCHITECTURE GRAPH) *An architecture graph (N,L) is a directed graph where each node $u,v \in N$ represents either a tile or a router, and each edge $l = (u,v) \in L$ represents a link from node u to node v.*

Communication between tiles involves sending data over a sequence of links from the source to the destination tile. Such a sequence of links through the architecture graph is called a *route* and is defined as follows.

Definition 2. (ROUTE) *A route r between node u and node v with $u \neq v$ is a path in the architecture graph of consecutive links from u to v without cycles. The operators src and dst give respectively the source and destination node of a route or a link. The length of a route r is equal to the number of links in the path, and denoted $|r|$. We use $l \in r$ to denote that the link l appears in the route r.*

Links can be shared between different communications by using a TDMA-based scheduler in the routers and NIs. All links have the same number of TDMA *slots*, N, and each slot has the same bandwidth. At any moment in time, at most one communication can use a slot in a link. This guarantees that the NoC schedule is contention-free. Hence, no deadlock will occur. The data transfered over a link in a single slot is called a *flit* and it has size sz_{flit} (in bits). To minimize buffering in routers, a flit entering a router at time-slot t must leave the router at slot $t+1$. Not all slots in a link may be available for use by a single application. Part of the slots may already be used by other applications mapped to the system.

Wormhole routing [6] is used to send the flits through the network. This technique requires limited buffering re-

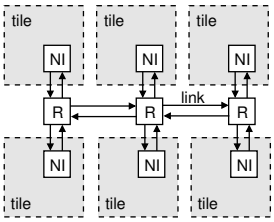


Figure 3. NoC-based MP-SoC architecture.

sources and offers strict latency bounds. A message is divided by the sending NI into flits. The flits are then routed through the network in a pipelined fashion. This reduces the communication latency considerably. All flits which belong to the same message and are sent in consecutive slots form a *packet*. The first flit in a packet (header flit) contains all routing information and leads the packet through the network. The header has a fixed size of sz_{ph} bits ($sz_{ph} \leq sz_{flit}$). The remaining $sz_{flit} - sz_{ph}$ bits in the header flit can be used to send (a part of) the actual message. The size of the header must be taken into account when allocating resources in the NoC. Two messages, possibly sent between different source and destination tiles but over one link at non-overlapping moments in time can use the same slot. For messages that use the same slot in the link between the source tile and the first router but a different route, the routing information stored in the NI for this slot must be changed. This can be implemented efficiently by sending a message from a processor or communication assist [4] inside a tile to its NI to change the routing information. The time required to reconfigure the NI is T_{reconf}. During this reconfiguration time, the slot may not be used to send messages.

Tasks in an application communicate with each other through streams of messages. The ordering of the messages in a stream must be preserved. To realize this, the NIs send messages onto the network in the same order as they receive them from the processors. The scheduling of communications on the NoC must also guarantee that the messages are received in the same order. No reordering buffers are thus needed in the NIs, which simplifies their hardware design. The NoC further requires that when the communication of a message is started, slots are claimed in the links it is using. These slots are only freed after the communication has ended. Preemption of a communication is not supported.

5. Time-Constrained Scheduling Problem

5.1. Overview

Informally, this paper tries to find a schedule for a set of scenarios CS, such that the schedule of each scenario has no conflicting resource requirements with the other scenarios and other applications, and that the set of messages which make up a scenario are sent between different tiles in a system within given timing constraints. First, we formalize in Sec. 5.2 the problem of scheduling a single scenario $s \in CS$. The formalization is such that only a single period of the scenario needs to be scheduled. This schedule can then be repeatedly executed as often as necessary. It uses a function $\mathcal{U} : L \times \mathbb{N} \rightarrow \{used, not\text{-}used\}$ which indicates for every link at every moment in the time-span of one scenario period of the schedule whether a slot is occupied. The function $\mathcal{U}$ captures the resource constraints due to other applications using the same platform and from the schedules of other scenarios on the messages being scheduled from scenario s. Sec. 5.3 explains how the function $\mathcal{U}$ is constructed and used when scheduling multiple scenarios on the NoC. The complexity of the single-scenario scheduling problem is studied in Sec. 5.4.

5.2. Scheduling a Single Scenario

A communication scenario consists of a set of messages which must be scheduled on the NoC within their timing constraints. A message is formally defined as follows.

Definition 3. (MESSAGE) *Given an architecture graph (N,L), a set of streams S and a period P. A message m is a 7-tuple (u,v,s,n,τ,δ,sz), where $u,v \in N$ are respectively the source and the destination tile of the n-th message sent through the stream $s \in S$ during the period P. The earliest time at which the communication can start, relative to the start of the period, is given by $\tau \in \mathbb{N}$ ($0 \leq \tau < P$). The maximum duration of the communication after the earliest start time is $\delta \in \mathbb{N}$ ($\delta \leq P$). The size (in bits) of the message that must be communicated is $sz \in \mathbb{N}$.*

In the application task graph shown in Fig. 2, a message $m_1 = (u,v,s_1,n,\tau,\delta,sz)$ is sent each period P_1 through the stream s_1 of scenario 1. This communication can start at time τ and must finish before $\tau + \delta$. Note that a communication may start in some period and finish in the next period. This occurs when $\tau + \delta > P_1$.

In practice, messages may not always have a fixed earliest start time, duration, or size. Conservative estimates on these figures should be used to construct the set of messages in order to guarantee that all communications fall within the timing and size constraints. Resources that are claimed but not used, due to for example a smaller message size, can be used to send data without timing requirements between tiles without providing guarantees, i.e. best-effort traffic.

A message specifies timing constraints on the communication of data between a given source and destination tile. It does not specify the actual start time, duration, route and slot allocation. This information is provided by the *scheduling entity*.

Definition 4. (SCHEDULING ENTITY) *A scheduling entity is a 4-tuple (t,d,r,st), where $t \in \mathbb{N}$ is the start time of the scheduled message relative to the start of the period and $d \in \mathbb{N}$ is the duration of the communication on a single link.*

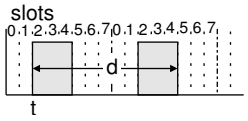


Figure 4. Scheduling entity on a link.

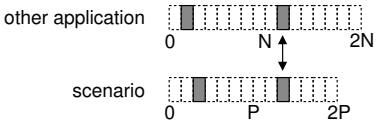


Figure 5. Contention in second period.

The scheduled message uses the route r in the network and the slots it uses from the slot table of the first link $l \in r$ are given by contained in the set $st \subseteq \{0,\ldots,N-1\}$ with N the slot-table size.

The slots given in *st* are claimed on the first link of the route r at time t for the duration d. On the next link, the slot reservations are cyclically shifted over one position. So, these slots are claimed one time-unit later, i.e., at $t+1$, but for the same duration d. The complete message is received by the destination at time $t+d+|r|-1$. Fig. 4 shows a scheduling entity which sends a message over a link with a slot-table of 8 slots. Starting at time $t=2$, the slots 2, 3, and 4 are used to send the message. The communication ends after $d=11$ time units. In total two packets consisting both of three flits are used to send the message.

The relation between a message and a scheduling entity is given by the *schedule function*, formally defined below in Def. 5. Among all schedule functions, those respecting the constraints in Def. 5 are called *feasible*. One of the conditions that a feasible schedule needs to satisfy is that it is contention free, i.e., slot-tables should not be simultaneously reserved by different messages, different scenarios or different applications. An important aspect in this context is the relation between the slot-table size N and the period of the scenario P. Fig. 5 shows an example of a link l with slot table size $N=8$. The second slot from the slot table is occupied by another application. The message(s) from a scenario with period $P=7$ are also scheduled on the link l. In the first period, the scenario uses the third slot from the slot table. In the next period, the scenario uses the second slot from the slot table. However, this slot is already occupied by another application. Hence, there is contention on the link at this moment in time as both schedules want to use the same slot at the same moment in time. This example shows that it is in general not sufficient to guarantee that the first period of some scenario is contention-free; also following periods must be free of contention. In fact, the number of periods of a schedule with period P which must be checked for contention is equal to the least common multiple of P and N, $\mathrm{lcm}(P,N)$, divided by P. After this number of periods, the first time-unit of the scenario coincides again with the first slot of the slot table. For simplicity, we assume that the period of a scenario is a multiple of the size of the slot table. It is then sufficient to check only a single period for contention, which simplifies the formulas in the remainder. This is not a restriction as any scenario whose period P which does not adhere to this requirement can be concatenated for $\mathrm{lcm}(P,N)/P$ times before scheduling it. The period of the concatenated scenario is then a multiple of the slot table with size N.

Consider now Def. 5. The first two constraints make sure that the communication takes place between the correct source and destination tile. The third and fourth constraint guarantee that the communication falls within the timing constraints given by the message. The fifth constraint ensures that enough slots are reserved to send the message and packet headers over the network. It uses a function $\pi(e)$ which gives for a scheduling entity $e=(t,d,r,st)$ the number of packets which are sent between t and $t+d$ on the first link of the route r considering the slot reservations *st* and assuming that at time 0 the first slot of the slot table is active. The function $\varphi(e)$ gives the number of slots reserved by e between t and $t+d$. The sixth constraint makes sure that a scheduling entity does not use slots in links which are at the same moment in time used by other applications or scenarios. It uses a function $\sigma(e,l_k,x)$ which indicates for a scheduling entity e and the k-th link l_k on the route r of e whether it uses a slot from the slot-table with size N of l_k at time x; $\sigma(e,l_k,x) = \textit{used}$ when $(x+k) \bmod N \in (s+k) \bmod N$, else $\sigma(e,l_k,x) = \textit{not-used}$. The seventh constraint requires that the schedule is contention-free. The next constraint makes sure that there is enough time to reconfigure the NI between two messages which originate at the same NI and use the same slot but different routes. The last constraint enforces that the ordering of messages in a stream is preserved.

Definition 5. (SCHEDULE FUNCTION) *A schedule function is a function $\mathcal{S}: M \to E$ where M and E are respectively the set of messages and scheduling entities. We call $\mathcal{S}$ feasible if and only if, for all messages $m=(u,v,s,n,\tau,\delta,sz) \in M$ associated to scheduling entity $\mathcal{S}(m)=e=(t,d,r,st)$,*

1. *the route starts from the source tile: $u = src(r)$,*
2. *the route ends at the destination tile: $v = dst(r)$,*
3. *the communication does not start before the earliest moment in time at which the data is available: $t \geq \tau$,*
4. *the communication finishes not later than the deadline: $t+d+|r|-1 \leq \tau+\delta$,*
5. *the number of allocated slots is sufficient to send the data: $sz + sz_{ph} \cdot \pi(e) \leq sz_{flit} \cdot \varphi(e)$,*
6. *the communication uses no slots occupied by other applications or scenarios: for all links $l \in r$ and time*

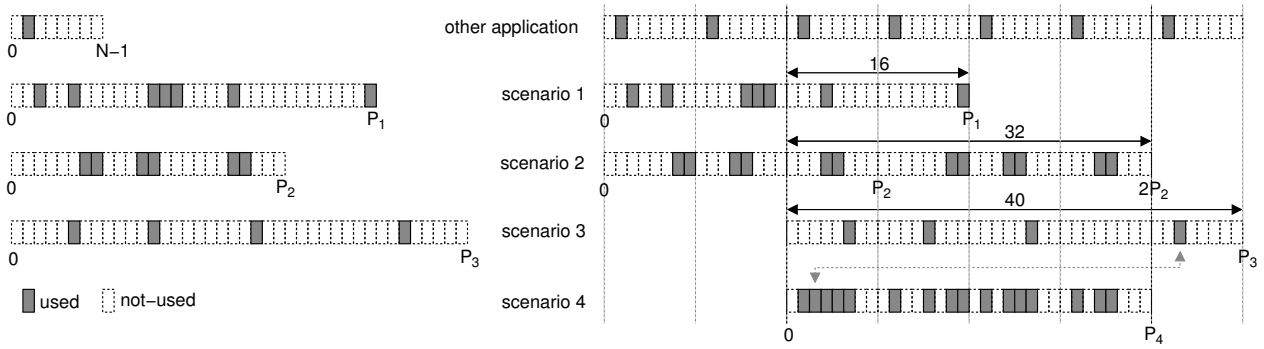


Figure 6. Constraints for link l on scenario 4.

instances x with $0 \leq x < P$, $\sigma(e,l,x) =$ *not-used when* $\mathcal{U}(l,x) =$ *used,*

and for each pair of messages $m_1, m_2 \in M$ *with* $m_1 \neq m_2$, $m_1 = (u_1, v_1, s_1, n_1, \tau_1, \delta_1, sz_1)$, $\mathcal{S}(m_1) = e_1 = (t_1, d_1, r_1, st_1)$, $m_2 = (u_2, v_2, s_2, n_2, \tau_2, \delta_2, sz_2)$, *and* $\mathcal{S}(m_2) = e_2 = (t_2, d_2, r_2, st_2)$,

7. *for all* $l \in r_1 \cap r_2$ *and for all* x *with* $0 \leq x < P$, $\sigma(e_1,l,x) =$ *not-used or* $\sigma(e_2,l,x) =$ *not-used,*

8. *if* $u_1 = u_2$, $r_1 \neq r_2$, *and* $st_1 \bigcap st_2 \neq \emptyset$, *then there is enough time to reconfigure the NIs:* $(t_2 - t_1 - d_1) \bmod P \geq T_{reconf}$,

9. *if* $s_1 = s_2$ *and* $n_1 < n_2$, *then the ordering of these messages is preserved:* $t_1 + d_1 < t_2 \wedge t_1 + d_1 + |r_1| - 1 < t_2 + |r_2|$.

If a schedule function is not feasible, it means that one or more of the above rules are violated in at least one associated scheduling entity. Such a schedule is called *infeasible*. By construction, any feasible schedule is contention-free and hence free of deadlock and livelock [11].

5.3. Scheduling Multiple Scenarios

The problem of scheduling a set of scenarios CS consists of finding a feasible scheduling function $\mathcal{S}$ for the streams in all scenarios $s \in CS$. Feasibility requires that resource constraints from a scenario $s_i \in S$ are considered when scheduling another scenario $s_j \in S$ which overlaps with s_i. Of course, slots occupied by other applications must also be taken into account. When scheduling a single scenario, all these constraints are taken into account through the function $\mathcal{U}$. This subsection explains with an example, illustrated in Fig. 6, how the function $\mathcal{U}$ is constructed.

Consider the situation in which messages of four scenarios must be scheduled on a link l. Assume that the link has a slot table of size 8 of which the second slot in the slot table is already occupied by another application. This slot cannot be used for any scenario. Assume now that the first three scenarios are already scheduled on l. To schedule the messages of the fourth scenario, a function $\mathcal{U}$ must be constructed which takes into account the slots used by the three already scheduled scenarios and the slots occupied by other applications. Let the scenarios 1, 2 and 3 have respectively a period of 32, 24 and 40 time-units. The slots occupied by the scheduled scenarios 1, 2 and 3 are shown on the left-hand side of Fig. 6. The fourth scenario has a period of 32 time-units and it has an overlap with the scenarios 1, 2 and 3 of respectively 16, 32 and 40 time-units. Note that a consequence of the earlier assumption that the start of a scenario aligns with a slot-table rotation is that the overlap between two scenarios is a multiple of the slot-table size. This means that a new scenario can only be started at the beginning of a slot-table rotation. If desirable, this restriction can be relaxed to allow arbitrary overlap, but this makes the construction of $\mathcal{U}$ more tedious and it seems of little practical value.

The overlap of 16 time-units between scenarios 1 and 4 implies that the last 16 time-units of the last period of scenario 1 overlap with the first 16 time-units of the first period of scenario 2. After these 16 time-units, scenario 1 is no longer repeated. So, the slots occupied in the last 16 time-units of scenario 1 cannot be used for scenario 4. Due to the overlap between scenarios 2 and 4, more than one period of scenario 2 overlaps with scenario 4. The slots occupied by scenario 2 cannot be used for scenario 4. The overlap of scenario 3 on scenario 4 is even larger than a complete period of scenario 4. Slots occupied after a complete period of scenario 4 do constrain the next period of the scenario and should therefore also be considered as a constraint on the available slots. The dotted arrow between scenario 3 and 4 gives an example of such a constraint. Only slots which are not occupied by other applications or any of the overlapping scenarios can be used to schedule the messages of scenario 4. These slots are colored white in Fig. 6.

5.4. Complexity

In this section, we prove that the single-scenario scheduling problem is NP-complete.

Theorem 1. *Given an architecture graph $G(N,L)$ and a scheduling problem consisting of a set of messages M with a period P. The problem of finding a feasible schedule function is NP-complete.*

We first show that the problem belongs to NP. The verification algorithm must check whether the scheduling entities satisfy the constraints of Def. 5. The first five constraints can be checked in $O(|M|)$ time. The other constraints can be easily checked in polynomial time for a single link $l \in L$, but need to be repeated for all $l \in L$. However, this is still polynomial in the problem size.

To prove that the problem is NP-complete, we show that the disjoint-path problem [8] can be reduced in polynomial time to our scheduling problem. The disjoint-path problem was proved to be NP-complete (even for planar graphs) [8]. In the disjoint-path problem, a set of edge-disjoint paths in a given graph must be found between a set of pairs of vertices. The reduction of the disjoint-path problem to our scheduling problem works as follows. Let G be the graph and the set $\{(u_1,v_1)\dots(u_k,v_k)\}$ the pairs of vertices that form an instance of the disjoint-path problem. We construct an instance of the scheduling problem with architecture graph G and the number of slots in each link equal to 1. We ignore the ordering of the messages in the streams, i.e. we assume that each message belongs to a different stream. The length of the period is equal to 1. The set of messages $M = \{(u'_1,v'_1,s_1,1,0,1,1),\dots(u'_k,v'_k,s_k,1,0,1,1)\}$, i.e., all messages have sequence number 1, starting time 0, deadline 1, and size 1. This construction can be done in polynomial time. Suppose that there are edge-disjoint paths $p_1,\cdots,p_k$ between $(u_1,v_1)\dots(u_k,v_k)$. For each i, the path p_i, which is in fact a route from node u'_i to v'_i in the scheduling problem, is exclusively dedicated to the message $(u'_i,v'_i,s_i,1,0,1,1)$. The message is scheduled to be sent at time 0 using all bandwidth in the links on its route. It is easy to see that this schedule function is feasible. Conversely, suppose that there is a feasible schedule function for the set M; then, the set of scheduling entities cannot share bandwidth, as sharing bandwidth leads to missing deadlines. Since bandwidths are all set to be 1, it trivially follows that the routes are all disjoint. Hence, any feasible schedule is a solution to the disjoint-path problem.

6. Scheduling Strategies

6.1. Overview

Given a set M of messages, a scheduling strategy for a single scenario must find a schedule entity e for each message $m \in M$ and the set E of scheduling entities must form a feasible schedule function (i.e., all constraints from Def. 5 must be met). Given that an exhaustive approach is not tractable, we present several heuristic approaches. The heuristics allow the user to trade off quality of solutions and effort spent on solving problems. First, a greedy strategy is presented in Sec. 6.2. Typically, the greedy approach gives a solution quickly. However, it also excludes a large part of the solution-space. The second strategy, ripup, adds backtracking to the greedy approach. This improves the quality (number of feasible solutions found for a set of problems), but it also increases the run-time. The backtracking tries to resolve scheduling conflicts when they occur. The third strategy, presented in Sec. 6.4, tries to avoid conflicts by estimating a priori the usage of all links. This should steer the routing process to avoid scheduling conflicts and as such minimizes the use of the backtracking mechanism. A feasible schedule for the messages of a single scenario takes into account the constraints that originate from other applications and scenarios. These constraints are captured in the function $\mathcal{U}$. A scheduling strategy for multiple scenario based on the scheduling strategies for a single scenario is presented in Sec. 6.5.

6.2. Greedy

The greedy strategy explores a small part of the solution-space. As a result, it has a small run-time. However, it may miss solutions or find non-optimal ones in terms of resource usage. The greedy strategy essentially tries to schedule the largest, most time-constrained messages first, via the shortest, least congested route that is available. It works as follows. First, all messages $m = (u,v,s,n,\tau,\delta,sz) \in M$ are assigned a cost using Eqn. 1 and sorted from high to low based on their cost. The cost function guarantees that messages are ordered according to their (integer) size (larger size first) and that two messages with the same size are ordered with respect to the duration (tighter constraint first).

$$\text{cost}_M(m) = sz(m) + \frac{1}{\delta(m)} \tag{1}$$

Next, a schedule entity $e = (t,d,r,st)$ must be constructed for the first message $m = (u,v,s,n,\tau,\delta,sz) \in M$. To minimize the resource usage, the scheduling strategy must try to minimize the length of the routes. For this reason, the greedy strategy determines a list R of all routes from u to v with the shortest length and assigns a cost to each route r using the following cost function that determines the minimum number of available slots in any link in a route during the time-span that the link might potentially be used by the message.

$$\text{cost}_R(r,m) = \min_{l_k \in r} \sum_{\tau(m)+k \le x \le \tau(m)+\delta(m)+k-|r|} \mathcal{F}(l_k,x) \tag{2}$$

with $\mathcal{F}(l_k,x) = 1$ when $\mathcal{U}(l_k,x) = \textit{not-used}$ and $\mathcal{F}(l_k,x) = 0$ otherwise. The routes are sorted from low to high cost giving preference to the least congested routes. Next, a schedule entity e is constructed using the first route r in

R. The scheduling strategy should avoid sending data in bursts as this increases the chance of congestion. Therefore, the start time, t, of e is set equal to the earliest possible time respecting the third and last constraint from Def. 5. Given t and the fourth and last constraint from Def. 5, the maximal duration d of e can be computed. All slots available between t and the maximal duration on the first link of the route, respecting the sixth, seventh and eighth constraint from Def. 5, are located. From these slots, a set of slots, st, is selected which offer sufficient room to send the message and the packet headers. The scheduler tries to minimize the number of packets that are used by allocating consecutive slots in the slot table. This minimizes the overhead of the packet headers, which in turn minimizes the number of slots needed to sent the message and its headers. This leaves as many slots as possible free for other messages. It is possible that no set of slots can be found which offer enough room to send the message within the timing constraints. If this is the case, the next route in R must be tried. In the situation that all routes are unsuccessfully tried, a new set of routes with a length of the minimum length plus one is created and tried. This avoids using routes longer than needed and it never considers a route twice. A route which uses more links than the minimum required is said to make a *detour*. The length of the detour is equal to the length of the route minus the minimum length. If no set of slots is found when a user-specified maximum detour of X is reached, then the problem is considered infeasible. If a set st of slots is found, the minimal duration d needed to send the message via the route r, starting at time t using the slots st is computed using the fifth constraint from Def. 5. The scheduling entity $e = (t, d, r, st)$ is added to the set of schedule entities E. The new set of schedule entities $E \cup \{e\}$ is guaranteed to respect all constraints from Def. 5. The next message can be handled. The process is repeated till a schedule entity is found for all messages in M, or until the problem is considered infeasible (a message cannot be scheduled).

6.3. Ripup

The ripup strategy uses the greedy strategy described in the previous section to schedule all messages. This guarantees that all problems that are feasible for the greedy strategy are also solved in this strategy. Moreover, the same schedule function is found. As soon as a conflict occurs (i.e. no schedule entity e_i can be found for a message m_i which meets the constraints given in Def. 5), an existing schedule entity e_j is removed from the set of schedule entities E. To choose a suitable e_j, the heuristic calculates for each schedule entity $e_j \in E$ the number of slots it uses in the links that can also be used by e_i. The higher this number, the larger the chance that e_j forms a hard conflict with e_i. A schedule entity e_j with the largest conflict is therefore removed from E. This process is continued until a schedule entity e_i for the message m_i can be created that respects the constraints given in Def. 5. After that, the messages of which the corresponding schedule entities were removed are re-scheduled in last-out first-in order. On a new conflict, the ripup mechanism is activated again. The user specifies the maximum number of times a ripup may be performed. This allows a trade-off between quality and run-time of the strategy.

6.4. Global knowledge

The ripup scheduler does not know a priori which unscheduled messages need to use links in the route it assigns to the message it is scheduling. It can only use local information to avoid congestion. The *global knowledge* strategy tries to estimate, before scheduling messages, the number of slots that are needed in each of the links. This gives the scheduling strategy global knowledge on the congestion of links. This knowledge is used to guide the route selection process when scheduling the messages.

Communication of a message m can take place at any moment in time within the time interval specified by m. Within this interval it requires at least $\left\lceil \frac{\lceil sz(m)/sz_{flit} \rceil}{\max(\lfloor \delta(m)/N \rfloor, 1)} \right\rceil$ slots in each link of the route it uses. In the optimal situation, all scheduled messages use a route with the shortest length. To estimate the congestion on all links in the NoC, the strategy assumes that only shortest length routes are used. For each link $l \in L$, the strategy computes the minimal number of slots required at each moment in time when all messages which can use l, as it is part of at least one of their shortest routes, would use the link l. The function $\mathcal{C} : L \times \mathbb{N} \rightarrow \mathbb{N}$ gives the estimated number of slots used in a link $l \in L$ at a given time x.

The global knowledge strategy uses the same algorithm as the ripup strategy. However, a different cost function is used to sort the routes it is considering when scheduling a message. The cost function used by the greedy and ripup strategy (Eqn. 2) is replaced by the following cost function.

$$\text{cost}_R(r,m) = \sum_{l_k \in r} \max_{\tau(m)+k \leq x \leq \tau(m)+\delta(m)+k-|r|} \mathcal{C}(l,x) \qquad (3)$$

This cost function ensures that the routes are sorted based on the estimated congestion of the links contained in the routes. Routes containing only links with a low estimated congestion are preferred over routes with links that have a high estimated congestion. This minimizes the number of congestion problems which occur during scheduling. As such, it makes more effective use of the allowed ripups.

6.5. Multiple Scenarios

Given a set S of scenarios, a scheduling strategy for multiple scenarios must find a feasible scheduling function for each scenario while taking into account the resource constraints of the scenarios on each other. These constraints are captured in the function $\mathcal{U}$ (See Sec. 5.3).

Our multi-scenario scheduling strategy must first decide on the order in which it schedules the scenarios. A scenario which overlaps with many other scenarios has potentially tight resource constraints as many other scenarios share resources with it. When this scenario is scheduled first, the scheduling functions can minimize the number of packets which it needs to sent its messages. This minimizes the resource usage of the scenario. The overlap in time-units between two scenarios is given by the function $o : S \times S \rightarrow \mathbb{N}$. To sort the scenarios according to the overlap which they have with each other, the multi-scenario scheduling strategy assigns a cost to each scenario using Eqn. 4 and sorts them from high to low based on their cost.

$$\text{cost}_S(s_i) = \sum_{s_j \in S} o(s_i, s_j) + o(s_j, s_i) \tag{4}$$

Next, a set of scheduling entities E_i must be constructed for the set of messages M_i which make up the first scenario s_i in the ordered set of scenarios. This is done using a single-scenario scheduling function. The greedy, ripup or knowledge strategy can all be used for this purpose. The function $\mathcal{U}$ contains at that moment only the constraints which originate from other applications. When the used single-scenario scheduling strategy finds a feasible scheduling function $\mathcal{S}$, the multi-scenario strategy updates the function $\mathcal{U}$ to include the constraints which originate from $\mathcal{S}(M_i)$. This is done using the procedure described in Sec. 5.3. The multi-scenario strategy continues by scheduling the next scenario in the ordered list of scenarios. This process of scheduling a scenario and updating the constraint function $\mathcal{U}$ continues till either no feasible scheduling function is found for a scenario or all scenarios are scheduled. In the latter case the problem is called feasible; else it is called infeasible.

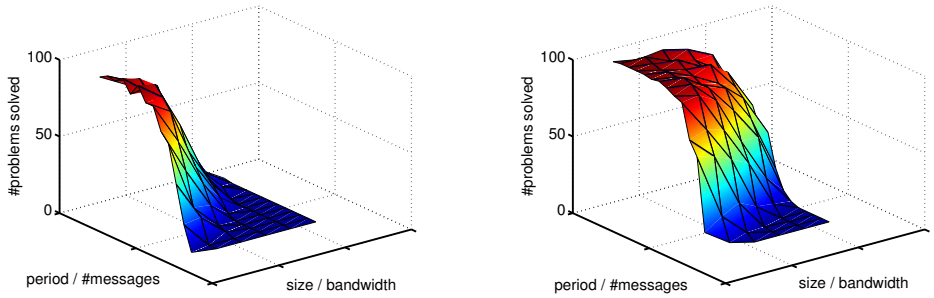
7. Benchmark

A benchmark is needed to test the quality of the scheduling strategies. It must contain a set of problems that covers a large part of the problem space typical of realistic applications. It should also be large enough to avoid optimization towards a small set of problems. It is not possible to construct a benchmark containing only real existing applications. Profiling these is too time-consuming and they are not representative for more demanding future applications at which NoCs are targeted. Therefore, a benchmark which consists of a set of randomly generated problems is used. A method to generate synthetic workloads for NoC performance evaluation is introduced in [29]. It assumes a communication model in which tasks exchange data-elements with each other through channels. The communication of data-elements has a periodic time behavior with some jitter on it. We use a benchmark generator to evaluate the scheduling strategies which is based on a similar idea.

Many NoCs use a regular topology like a mesh [12, 14, 16, 18] or torus [5]. Tiles located at the edge of a mesh are restricted in the links that can be used as at least one direction is not available because of the topology. In a 3x3 mesh this holds for all tiles except for one. In a 5x5 mesh there are 16 edge tiles and 9 non-edge tiles and a 7x7 mesh has 24 edge tiles and 25 non-edge tiles. The ratio of edge to non-edge tiles can possibly influence the scheduling strategies. To study this effect, problem sets are generated for a 3x3, 5x5 and a 7x7 mesh. All tiles in a 2D-torus have the same number of links. So, a torus has compared to a mesh more scheduling freedom as it has more links. To study, the effect of the additional scheduling freedom on the scheduling strategy, torus topologies with the same dimensions as the mesh topology are included in the benchmark. The NoC topology can be optimized when the applications running on it are known at design-time. This may result in the use of irregular NoC topologies [15, 20, 21]. To study how our strategies behave on irregular topologies, two topologies with respectively 9 and 25 tiles are added to the benchmark. Following [15], these topologies are based on a regular mesh topology in which 10% of the links are removed. These irregular topologies are constructed such that communication between any pair of tiles remains possible.

A traffic generator is developed which creates a user-specified number of streams of messages between randomly selected source and destination tiles. The streams can model uniform and hotspot traffic. All messages in a stream are assigned a start time, size, and duration which consists of a randomly selected base value which is equal for all messages in the stream plus a random value selected for each individual message in the stream. The first part can be used to steer the variation in message properties between streams. The second part can be used to create variation between messages in a single stream (i.e. jitter).

The problem space can be characterized in a 2-dimensional space. The first dimension is determined by the number of messages which must be communicated within a period. The second dimension is determined by the ratio of the size of the messages communicated and the available bandwidth. When constructing the problem sets, we found that there is an area in the problem-space where problems change from being easy to solve to unsolvable. We selected 78 equally distributed points around this area in the problem-space. For each point we generated 100 problems. This gives a benchmark with a set of 7800 different problems per topology-size and traffic model. The mesh, torus and irregular topologies have similar topology sizes and can thus share the problem sets. Fig. 7 shows for each point in the problem-space of the 5x5 mesh with uniform traffic how many problems are solved with the greedy and global knowledge strategies. The results for the greedy strategy show that most problems do not have a trivial solution. A



(a) 5x5 mesh with greedy.

(b) 5x5 mesh with knowledge.

Figure 7. Feasible problems in the problem-space.

solution is found for only 30% of the problems. The results of the global knowledge strategy show that 57% of the problems can be solved (and already suggest that global knowledge performs better than greedy). So, our benchmark contains problems which are not trivial to solve (i.e. greedy does not find a solution), but a solution does exist (i.e. global knowledge finds a solution). Note that when the problems get harder to solve, the demands on the resources are increased. More latency sensitive messages and/or larger messages (more bandwidth) need to be scheduled. Scheduling strategies which are more resource efficient will be able to solve more problems.

The benchmark must also contains multi-scenario scheduling problems in order to benchmark the multi-scenario scheduling strategy. These problems can be constructed by combining multiple single-scenario scheduling problems into one multi-scenario scheduling problem and selecting a random overlap between them. Scheduling problems from the 5x5 topologies are used for this purpose. Only scheduling problems are used for which all scheduling strategies, including the reference strategy [14] introduced in Sec. 8.1, find a feasible scheduling function. This guarantees that when the multi-scenario scheduling strategy cannot find a feasible scheduling function, this must be due to the resource constraints which result from scenario overlaps. In total, 500 random combinations of two problems are selected with a random overlap value between 5% and 50% of the period of the problems.

8. Experimental Results

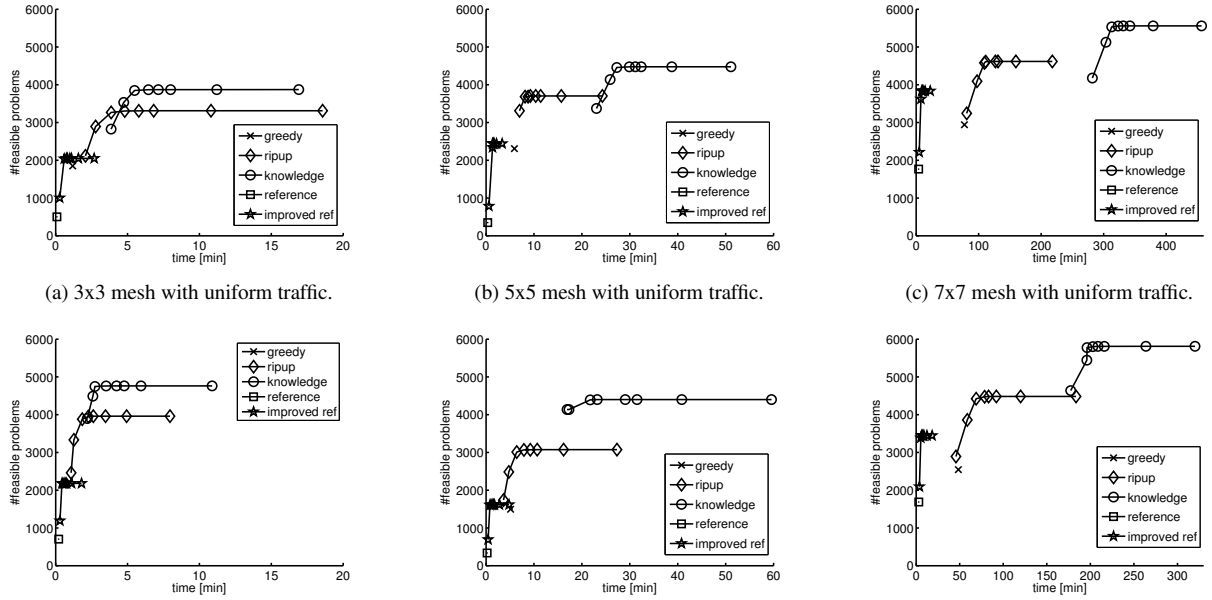
8.1. Single-Scenario Reference Strategies

A state-of-the-art scheduling strategy is presented in [14]. The strategy allows the use of non-shortest routes but it assumes that slots cannot be shared between different streams. Reconfiguration of the NIs is not possible. As in our greedy strategy, this strategy does not reconsider scheduling decisions when a conflict occurs. We used this strategy in our experiments as our *reference* strategy. It is implemented using the greedy strategy with an adapted cost function to sort the routes and three restrictions imposed on it. One, messages in one stream must use the same route. Two, streams are not allowed to share slots. Three, the reconfiguration time is equal to a period. This makes it impossible to reconfigure the NIs. The cost function that is used to sort the routes computes the ratio between the number of slots that are currently not-used in the links of a route and the total number of slots in the links of the same route. The experimental results suggest that using a backtracking mechanism is very effective. For this reason, we extended the reference strategy with our ripup mechanism. This strategy is used in the experiments as the *improved reference* strategy.

8.2. Single-Scenario Scheduling

All single-scenario scheduling strategies have been tested on the benchmark problems. The ripup, global knowledge and improved reference strategies have been tested with a number of different values for the maximum number of ripups (1, 10, 50, 100, 150, 200, 400, 800) to study the trade-off between the number of problems for which a solution is found and the run-time. A slot-table size of 8 slots is used in all experiments and the maximum detour (X) is initially set to 0. Note that $X = 0$ guarantees that any solution uses only shortest routes. This allows us to study for how many problems each strategy is able to find a solution with minimal resource requirements. The reconfiguration time of the NI, T_{reconf}, is set to 32 time units. This gives tiles 4 complete rotations of the slot table to reconfigure the NI. A processor or communication assist must send a message to update the routing information in the NI. The size of this message is less than the size of a single flit (i.e. it needs one time unit to be sent), so the value for T_{reconf} is conservative.

The trade-off between the run-time and the number of problems that is solved with the various strategies on the



(a) 3x3 mesh with uniform traffic.
(b) 5x5 mesh with uniform traffic.
(c) 7x7 mesh with uniform traffic.
(d) 3x3 mesh with hotspot traffic.
(e) 5x5 mesh with hotspot traffic.
(f) 7x7 mesh with hotspot traffic.

Figure 8. Trade-off between feasible problems and run-time.

mesh topologies is shown in Fig. 8. The trade-off curves for the other topologies are similar and therefore omitted. Tab. 2 summarizes the results for all strategies and all topologies assuming that 800 ripups are allowed. The column 'Improvement' shows the percentage of additional problems that is solved by all strategies compared to the reference strategy. The column 'Avg time' gives for each strategy the average run-time on a problem.

Looking at the number of problems solved, the results show that the reference strategy is outperformed by the improved reference strategy. This shows that adding backtracking to the state-of-the-art scheduling algorithm presented in [14] improves the results considerably. The results show further that the reference strategy solves less problems than greedy and the improved reference strategy solves less problems than the other two strategies using ripups. From this, we conclude that not using the ability of NoCs to reconfigure their connections is a limiting factor. As modern NoCs do not have this limitation, problems scheduled using the reference strategies may unnecessarily be considered infeasible or use unnecessarily many resources. Slot sharing is especially advantageous for hotspot traffic. For this type of traffic, our strategies are able to solve up-to 81% more problems than the improved reference strategy. This shows that slot sharing reduces the problem of contention on links connected to a hotspot.

The cost-functions in the greedy, ripup and global knowledge strategy which sort the routes have been adapted in this paper compared to our previous work [27]. Due to these changes, the three strategies solve currently respectively 48%, 9% and 9% more problems on the same meshtolopolgies and problem sets as compared to [27]. This shows that the improved cost-functions are more effective in sorting the routes which are tried in the scheduling strategies.

The results in Tab. 2 show that the global knowledge strategy always outperforms the other strategies. However, the average run-time on a problem is larger for this strategy than for the other strategies. This is caused by the congestion estimation made at the start of the strategy. Simpler estimates might be used to reduce its run-time. The reference and improved reference strategy have always the lowest run-time. This is logical as route selection is done only once for all messages in a stream and the slot allocation does not have to consider reconfiguration of slots.

Modern NoCs allow the use of flexible routing schemes (i.e. routes may use a detour). More problems may be solved when this flexibility is used. To quantify this gain, we tested all strategies with a maximum detour of 2 on all problems in our benchmark. The results of this experiment are shown in column 'Detour' of Tab. 2, which shows the improvement in the number of problems solved when compared to the reference strategy with detour zero. It shows that using non-shortest routes helps in solving additional problems.

8.3. Multi-Scenario Scheduling

The multi-scenario strategy can use all available single-scenario scheduling strategies. For the experiments, the

Table 2. Results single scenario problem.

Mesh-topology			
	Improvement	Avg time [ms]	Detour ($X = 2$)
Greedy	118%	178	304%
Ripup	277%	615	365%
Knowledge	371%	1201	420%
Reference	0%	9	5%
Improved ref.	213%	80	263%
Torus-topology			
	Improvement	Avg time [ms]	Detour ($X = 2$)
Greedy	284%	54	434%
Ripup	417%	111	473%
Knowledge	458%	196	487%
Reference	0%	5	1%
Improved ref.	242%	14	255%
Arbitrary topology			
	Improvement	Avg time [ms]	Detour ($X = 2$)
Greedy	58%	66	295%
Ripup	260%	602	454%
Knowledge	449%	905	595%
Reference	0%	6	68%
Improved ref.	148%	167	339%

knowledge strategy with 800 ripups is used as this strategy solves the largest number of single-scenario scheduling problems. This indicates that this strategy is the most resource efficient strategy. To compare our multi-scenario scheduling strategy, we use the state-of-the-art strategy presented in [19]. This strategy assumes that two scheduling problems cannot share slots when the scheduling problems overlap. In [19], the strategy is used in combination with our reference strategy. The experiments on the single-scenario scheduling strategies showed that all our strategies are more resource efficient than this strategy. To exclude the influence of the reference strategy on the results obtained with the strategy from [19], we use the strategy from [19] in combination with our best strategy, the knowledge strategy with 800 ripups.

All multi-scenario problems from our benchmark are scheduled on a 5x5 mesh topology. Our scheduling strategy is able to find a feasible scheduling function for 74% of the problems. The strategy from [19] fails to find a feasible scheduling function for any of the 500 problems. This is due to the fact that the number of slots occupied by each individual single-scenario problem is too large to allow a combination of both problems on the NoC. To schedule the multi-scenario scheduling problems using the strategy from [19] a NoC with more resources (e.g. larger bandwidth, larger slot tables) is needed. Using our multi-scenario strategy, many problems can be scheduled within the available resources. This shows that the impact of slot sharing between scenarios can have a large impact on the required resources.

8.4. Cost functions

Cost functions are used in the scheduling strategies to sort the messages M and routes R. The cost functions should minimize the chance of having a conflict when scheduling messages. They are constructed in such a way that the most resource constrained messages are handled first and that the resource usage is balanced over all links in the NoC. However, by doing so, they up-front exclude points from the solution-space. To circumvent this problem, randomly ordered sets M and R can be used as an alternative for the cost functions.

To test the impact of the cost functions on the quality of the strategies, the cost functions in the ripup strategy are replaced with a mechanism which assigns random costs to messages and routes. Experiments showed that the number of times we this randomized the strategy with a fixed number of ripups on a given problem set did not have an influence on the number of problems for which a feasible schedule function is found. However, the number of problems solved within a limited run-time and randomly ordered messages and routes is far lower than the number of problems solved by any of the heuristics in the same time. This shows that the cost functions in the heuristics are effective in ordering the messages and routes.

8.5. Scalability

The experiments showed that the run-time of the various strategies increases when the size of the topology increases (see Fig. 8). This is caused by the fact that the number of links in a route increases and that the number of routes which is considered when a scheduling strategy tries to find a schedule entity for a message increases. Therefore we also did some experiments for a 9x9 mesh. When going from a 5x5 mesh to a 9x9 mesh and assuming that each tile has an equal chance of being the source/destination of a stream, the average number of links in a route goes from 3.33 links to 6.00 links and the average number of routes which is considered for a single scheduling entity increases from 5.41 routes to 113.73 routes. So, with increasing mesh-size, a scheduling strategy has to consider potentially more links and more routes when it tries to find a scheduling entity for a message. Fig. 9 shows the relation between the run-time of the scheduling strategies and the average number of routes which is considered when scheduling a message on a mesh-topology. It shows that the run-time of the strategies increases more than linearly with increasing mesh-size. This could potentially lead to large run-times for the scheduling strategies when the size of the NoC increases. However, in many practical situations, an algorithm that maps tasks to tiles will try to keep the source and destination of a stream close to each other. In other words, the algorithm will try to map an application to a region of the platform and not utilizing resources all across the platform. As a result, the average length of a route and the number of routes which is considered when scheduling a message is in many practical cases not proportional with the topology-size. It grows (much) less rapidly. Furthermore, the experimental results presented in Tab. 2 show that the run-time of all strategies is

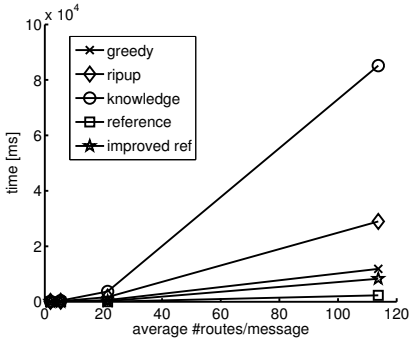


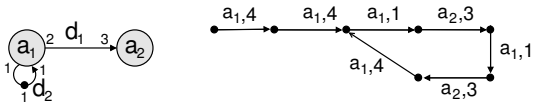
Figure 9. Relation between run-time and mesh-size.

still within seconds when an application is mapped to a region of 7x7 tiles. So, all scheduling strategies can definitely be used when an application is mapped to a region of this size.

9. Extracting Communication Scenarios from SDFGs

The application model presented in Sec. 3 has a communication centric view on an application. Typically, an application is not described in this model when mapping the application onto a NoC. Concurrent multimedia applications which are realized using MP-SoCs are often described with dataflow models [23]. Synchronous Dataflow Graphs (SDFGs) [17] is a popular dataflow model as techniques exist to study, for example, the throughput [9] and storage requirements [28] of an SDFG. This section explains how an application modeled as an SDFG can be converted to the application model presented in Sec. 3.

An SDFG consists of a set of actors (tasks) which communicate with each other by sending tokens (messages) to each other via dependency edges. Every time an actor fires (executes) it consumes a fixed number of tokens from its inputs (consumption rates) and after the execution time of the actor has elapsed it produces a fixed number of tokens on its outputs (production rates). An example of an SDFG is shown in Fig. 10(a). It consists of an actor a_1 with an execution time of 4 time-units, an actor a_2 with an execution time of 2 time-units, two dependency edges d_1 and d_2 and one initial token on d_2. Rates are associated with the source and sink of edges. Edge d_1 has a (worst-case allowed) latency of 1 time-unit. Edge d_2 has no latency as it only constrains the execution of actor a_1 and does not involve any true communication. Actor a_1 can fire when there is a token on d_2. At the start of the firing, it consumes the token. When the firing ends after 4 time-units, it produces the token again on d_2, while also producing 2 tokens on d_1. The token on d_2 enables the next firing of a_1. Hence, a_1 can fire every 4 time-units. Actor a_2 can fire as soon as there are 3 tokens on the dependency edge d_1. At the start of a firing, it simply consumes these tokens from d_1.



(a) SDFG. (b) Self-timed execution taking into account the latency of d_1.

Figure 10. Example of an SDFG with its execution.

The execution of an SDFG can be captured in a state-space traversal [9]. The state-space of an execution of our example SDFG is shown in Fig. 10(b). States are represented by black dots and state transitions are indicated by edges. The label with a transition indicates which actors start their firing in this transition and the elapsed time till the next state is reached. In our example, the first firing of actor a_1 is started at time 0. This firing is completed after 4 time-units. The execution proceeds in a self-timed manner meaning that all actors are fired as soon as possible while respecting dependency, execution time and latency constraints. It is known that this type of execution achieves the highest possible throughput. In general, an execution of an SDFG always starts with a, possibly empty, transient part followed by a periodic execution of the actors. In our example, the transient phase takes 8 time-units and is followed by a periodic phase of 12 time-units.

The behavior of actor executions is different in the transient and periodic phase of the SDFG execution. Both phases define when actors are fired and therefore when tokens are sent and received. The communication behavior of each phase can be captured in a communication scenario. The first scenario captures the communications related to the transient phase. The second scenario contains the communications that belong to the periodic phase. The communication pattern of both scenarios and their overlap can be derived from the execution of the SDFG.

Consider as an example the SDFG and its self-timed execution shown in Fig. 10. The execution of the graph starts with firing actor a_1 (see Fig. 10(b)). After 4 time units, the firing of a_1 ends and 2 tokens are produced on edge d_1. These tokens are consumed by the first firing of a_2 which occurs 9 time-units after the start of the execution of a_1. In our application model (see Sec. 3), the tokens are captured as two messages which both have an earliest start-time at time-unit 4 and a duration of 5 time-units (see Fig. 11); note that this duration exceeds the worst-case allowed latency of dependency d_1, but this is allowed and even desirable because a_2 can only be fired after another firing of a_1. The second time that actor a_1 produces 2 tokens on d_1 is at time-unit 8. One of the tokens which is produced at that moment is used to fire a_2 at time-unit 9. This token has to respect the latency constraint of dependency d_1. The other token is used in the next firing of a_2 which occurs at time-unit 13. Both the state in which the tokens of this sec-

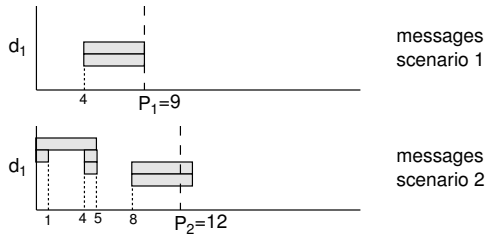


Figure 11. Scenarios in the SDFG execution.

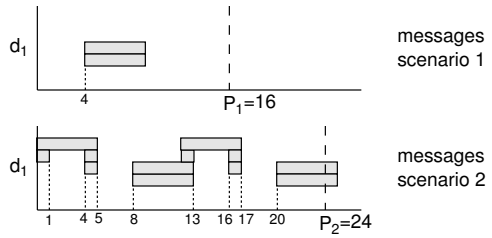


Figure 12. Scenarios with extended periods.

ond firing of a_1 are produced and the state in which these tokens are consumed belong to the periodic phase of the execution. Therefore, the communication of tokens between the actor firings belongs to the communication scenario of the periodic phase. Hence, the communication pattern related to the transient phase (scenario 1) contains only two messages which can be sent starting from time-unit 4. The scenario ends at time-unit 9 with the consumption of both messages. The period of the scenario is therefore equal to 9 time-units. The communications related to the periodic phase are captured in scenario 2 (see Fig. 11). This scenario starts at time-unit 8 with the production of two tokens by a_1 on d_1. One token needs to be delivered within 1 time-unit; the other within 5 time-units. The start times and durations of the other messages can be derived in a similar way from the periodic part of the (self-timed) execution. The length of the cycle in the state-space is 12 time-units. So, the period of scenario 2 is equal to 12 time-units. Scenario 1 ends at time-unit 9 and scenario 2 starts at time-unit 8. The overlap between them is thus 1 time-unit.

The method described above can be used to extract the communication scenarios and their overlap from the execution of an SDFG. It cannot be guaranteed that the length of the periods and their overlap is a multiple of the slot table size. However, as explained in Sec. 5, the scheduling problem assumes that both the period and overlap are a multiple of the slot table size. To guarantee that the overlap between the scenarios and the period of scenarios are a multiple of the slot table size, the length of scenario 1 (transient) must be extended such that this requirement is met. First, the end of scenario 1 is simply extended to guarantee that the overlap becomes a multiple of the slot table size. For our example SDFG, this means an extension of the period of scenario 1 from 9 time-units to 16 time-units. When needed, the start of scenario 1 is also shifted such that the total period becomes a multiple of the slot table size. This is not needed for our example. To solve the potential mismatch between the period of scenario 2 (P_2) and the slot table size (N), the scenario should be concatenated a number of times ($\text{lcm}(P_2,N)/P_2$) till the period of the resulting scenario is equal to $\text{lcm}(P_2,N)$. The resulting scenarios with their periods for our example are shown in Fig. 12. The overlap between the two scenarios is 8 time-units.

10. Experiments on a multimedia system

Besides the synthetic streams, a realistic multimedia application consisting of an MPEG-4 decoder and an MP3 decoder is used in the experiments. Both applications are modeled together in an SDFG which contains 15 actors (tasks) and 16 channels (streams). All actors in the application task graph are mapped and scheduled (manually) onto a 2x2 mesh. Actor execution times are obtained via worst-case execution time analysis, and worst-case dependency latencies from the NoC. Using the method described in the previous section, communication scenarios are derived for the transient and periodic phase of the SDFG execution. For each scheduling strategy, the minimal slot table size is determined for which a feasible schedule is found. When only shortest routes are used, both the reference strategy and the improved reference strategy require a slot table with 8 slots. Our strategies require a slot table with only 2 slots. This shows that for a realistic application slot sharing reduces the resource requirements on the NoC (i.e. fewer slots need to be allocated for the application). When the maximum detour is 2 links, the reference strategies require a slot table with 6 slots and our strategies require a slot table with 1 slot. This confirms that using non-minimal routes reduces the requirements on the NoC.

11. Conclusion

This paper studies the problem of scheduling time-constrained communication of streaming applications on a NoC. Several new strategies are presented to route and schedule streaming communication. The scheduling strategies use all routing and scheduling flexibility offered by modern NoCs while limiting resource usage. Short routes and the reservation of consecutive slots in slot tables minimize resource usage and packetization overhead. However, they also create potential bottlenecks in the NoC, which may render some resources unusable for scheduling other streams. The use of non-minimal routes and non-consecutive slot reservations might increase scheduling freedom for remaining streams. Our strategies try to find a good compromise in the allocation of routes and slot-table slots. The experiments show that our strategies perform better than the state-of-the-art strategy of [14]. The reason is that our strategies exploit freedom offered by modern NoCs

not used in the existing strategy. Additionally, we found that adding backtracking to this state-of-the-art strategy improves its results considerably with only a small overhead on its run-time.

This paper also shows that the dynamism in communication patterns of an application can be captured in a set of time-constrained scheduling problems. Sharing slots from a slot-table between these problems is possible when the timing relations between the problems are taken into account. The experimental results show that this reduces the amount of resources needed to schedule an application onto a NoC outperforming the technique of [19].

The presented scheduling strategies assume that tasks have already been mapped and scheduled onto the tiles. This is an important step in a systems-on-chip design flow. In our future work, we want to consider this mapping and scheduling step and study the phase coupling with the routing and scheduling problem presented in this paper.

References

[1] B. Ackland, et al. A single chip 1.6 billion 16-b MAC/s multiprocessor DSP. *IEEE Journal of Solid-State Circuits*, 35(3):412–424, 2000.

[2] M. Bekooij, et al. Predictable multiprocessor system design. In *SCOPES'04, Proc.*, pages 77–91. Springer, 2004.

[3] L. Benini and G. de Micheli. Networks on chips: A new SoC paradigm. *IEEE Comp.*, 35(1):70–78, 2002.

[4] D. Culler, et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[5] W. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *DAC'01, Proc.*, pages 684–689. ACM, 2001.

[6] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of distributed computing*, 1(3):187–196, 1986.

[7] O. Gangwal, et al. *Dynamic and Robust Streaming In and Between Connected Consumer-Electronics Devices*, chapter Building Predictable Systems on Chip: An Analysis of Guaranteed Communication in the AEthereal Network on Chip, pages 1–36. Springer, 2005.

[8] M. R. Garey and D. S. Johnson. *Computers and interactability: a guide to the theory of NP-completeness*. W.H. Freeman and Co., 1979.

[9] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *ACSD'06, Proc.*, pages 25–36. IEEE, 2006.

[10] S. Gheorghita, T. Basten, and H. Corporaal. Profiling driven scenario detection and prediction for multimedia applications. In *SAMOS'06, Proc.*, pages 63–70. IEEE, 2006.

[11] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *Computer Architecture, Proc.*, pages 278–287, 1992.

[12] K. Goossens, et al. A design flow for application-specific networks on chip with guaranteed performance to accelerate SoC design and verification. In *DATE'05, Proc.*, pages 1182–1187. IEEE, 2005.

[13] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *DATE'00, Proc.*, pages 250–256. IEEE, 2000.

[14] A. Hansson, et al. A unified approach to constrained mapping and routing on network-on-chip architectures. In *CODES+ISSS'05*, pages 75–80. IEEE, 2005.

[15] R. Holsmark, M. Palesi, and S. Kumar. Deadlock free routing algorithms for mesh topology NoC systems with regions. In *DSD'06, Proc.*, pages 696–703. IEEE, 2006.

[16] J. Hu and R. Marculescu. Communication and task scheduling of application-specific networks-on-chip. *IEE Proc. Computers and Digital Techniques*, 152(5):643–651, 2005.

[17] E. Lee, et al. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[18] M. Millberg, et al. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE'04, Proc.*, pages 890–895. IEEE, 2004.

[19] S. Murali, et al. A methodology for mapping multiple use-cases onto networks on chip. In *DATE'06, Proc.*, pages 118–123. IEEE, 2006.

[20] C. Neeb and N. Wehn. Designing efficient irregular networks for heterogeneous systems-on-chip. In *DSD'06, Proc.*, pages 665–672. IEEE, 2006.

[21] U. Ogras and R. Marculescu. Application-specific network-on-chip architecture customization via long-range link insertion. In *ICCAD'05, Proc.*, pages 246–253. IEEE, 2005.

[22] P. Paulin, et al. Application of a multi-processor SoC platform to high-speed packet forwarding. In *DATE'04, Proc.*, pages 58–63. IEEE, 2004.

[23] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES'03, Proc.*, pages 63–72. ACM, 2003.

[24] E. Rijpkema, et al. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEE Proceedings: Computers and Digital Techniques*, 150(5):294–302, 2003.

[25] A. Rădulescu, et al. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Trans. on CAD of ICs and systems*, 24(1):4–17, 2005.

[26] M. Rutten, et al. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, 2002.

[27] S. Stuijk, T. Basten, M. Geilen, A. Ghamarian, and B. Theelen. Resource-efficient routing and scheduling of time-constrained network-on-chip communication. In *DSD'06, Proc.*, pages 45–52. IEEE, 2006.

[28] S. Stuijk, et al. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC'06, Proc.*, pages 899–904. ACM, 2006.

[29] R. Thid, et al. Flexible bus and NoC performance analysis with configurable synthetic workloads. In *DSD'06, Proc.*, pages 681–688. IEEE, 2006.