Improved Priority Assignment for Real-Time Communications in On-Chip Networks *

Meng Liu, Matthias Becker, Moris Behnam, Thomas Nolte
Mälardalen University, Västerås, Sweden
Email: {meng.liu, matthias.becker, moris.behnam, thomas.nolte}@mdh.se

ABSTRACT

The Network-on-Chip is the on-chip interconnection medium of choice for modern massively parallel processors and System-on-Chip in general. Fixed-priority based preemptive scheduling using virtual-channels is a solution to support real-time communications in on-chip networks. However, the different characteristics of the Network-on-Chip compared to the single processor scheduling problem prevents the usage of known optimal algorithms (e.g. the Audsley's algorithm) to assign priorities to messages. A heuristic search algorithm based approach (called the HSA) focusing on the priority assignment for on-chip communications has been presented in the literature. The HSA is much faster than an exhaustive search based solution, with a price of missing certain schedulable cases (i.e. non-optimal). In this paper, we present two undirected-graph based priority assignment algorithms, the GESA and the GHSA. In contrast to the previous work, we can decrease the search space significantly by taking the interference dependencies of different messages on the network into account. A number of experiments are generated, in order to evaluate the proposed algorithms. The results show that the GESA can always achieve higher schedulability ratios than the HSA, but may require longer processing time. On the other hand, the GHSA has the same performance as the HSA regarding the schedulability, but can significantly improve the efficiency.

1. INTRODUCTION

The Network-on-Chip (NoC) is the preferred interconnection medium for massively parallel platforms as well as for System-on-Chip (SoC). In contrast to bus-based interconnection mediums, the NoC is scalable up to a large number of elements. It connects multiple *Intellectual Property* (IP) cores, where each IP core can provide different functionalities, possibly from different vendors. In order to function correctly, those IP cores exchange data through the NoC. Thus the NoC is crucial in order to guarantee correct functionality of each IP core and the system in general. Additionally, timeliness of the messages (also called *flows* hereinafter) sent on the NoC becomes more and more important if such platforms are intended for real-time applications. Here each message needs to be delivered within a certain deadline.

*This work has been supported by the Swedish Knowledge Foundation via the research project PREMISE.

RTNS 2015, November 04-06, 2015, Lille, France

DOI: http://dx.doi.org/10.1145/2834848.2834867

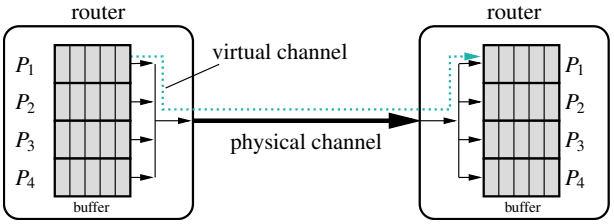

Figure 1: Virtual channels on the NoC router.

The NoC itself consists of different architectural elements [1]. Routers are used to route the messages through the network, and unidirectional channels are used to connect the routers. While different network topologies are possible, the 2-dimensional mesh-based NoC is the most common [2, 3]. Here the IP cores are arranged on a 2-dimensional grid. Each IP core connects to a router and the router in turn connects to its neighbors in the cardinal directions.

Different NoC designs exist for different requirements. In this work we focus on wormhole switched NoC with virtual channels. In wormhole switched networks [4], a message is divided into so called flits, where a flit represents the data size which can be transmitted between two routers during one cycle. The header flit contains most of the routing information such as the path, the flit count, etc.. Once the header flit proceeds from one router to the next, the channel is reserved until the last flit passes, hence the name wormhole routing.

On the router, each channel connects to a buffer, see Figure 1. In order to reduce the footprint on the die and to limit the costs, those buffers are generally small and can not hold complete messages. In other words, a message can span over multiple routers during transmission, giving the impression of a worm traversing through the network. Once a channel is reserved by a message, other messages, competing for the same channel, are blocked. This can lead to long blocking times. Virtual-Channels (VC) [5] have been proposed to overcome this challenge. A VC is an additional buffer connected to the channel. Messages can now use separate buffers and, while one message is blocked, a second message with the same path can still proceed, since the channel is idle and the buffers are not shared. The dotted line in Figure 1 depicts such a virtual channel. However, the channel itself is still an exclusive resource. Switching between virtual-channels can be achieved in different ways (e.g. round-robin, fixed-priority based, etc.). In this paper, we focus on fixed priority based arbitration. The buffers are thus related to different priority levels and each message is assigned

a priority respective to the used buffer. If two messages simultaneously arrive at a router, the message which uses the higher priority is allowed to transmit. Only if the message becomes blocked or transmits all its flits through the link, a message with a lower priority gets access to the channel. On the same way a flow can be preempted, if a higher priority flow arrives at a router competing for the same channel. The priority assignment of the messages therefore becomes crucial for the schedulability of flow sets. Even though one channel has similar characteristics like single processor scheduling, the complete network cannot directly be related to the results from scheduling theory. In specific, Audsley's priority assignment algorithm is not optimal for NoC [6, 7].

The contributions presented in this paper are:

1. We present two algorithms (called the GESA and the GHSA) for priority assignment of NoC messages. The proposed algorithms use an undirected-graph based search algorithm, which can significantly improve the efficiency of priority assignment process.

2. We identify, with illustrative examples, the limitations of the current state-of-the-art alternative priority assignment algorithm for wormhole switched NoC, called the HSA.

3. We perform an extensive evaluation of the priority assignment algorithms regarding their performance in terms of schedulability and computational complexity. The results of this evaluation clearly show that the proposed algorithm GHSA can be much faster than the HSA while achieving the same schedulability ratio. On other hand, the GESA, which can achieve higher schedulability ratios compared to the HSA, is slower than the HSA but much faster than an exhaustive search solution.

The remainder of this paper is organized as follows. In Section 2 we present the related work. The system model assumed in this paper is shown in Section 3. In Section 4 we revisit the existing priority assignment algorithm for on-chip communications, along with several motivating examples. Section 5 presents the new undirected-graph based priority assignment algorithm. The evaluation of the proposed algorithms is presented in Section 6. Finally, in Section 7 we conclude the paper.

2. RELATED WORK

The NoC is implemented as interconnection medium on several massively parallel processors in order to cope with the increased message volume [8, 2, 3]. As many platforms target the embedded industry, an increased interest lies on the real-time requirements for messages on NoCs [9, 10].

Wormhole switching/routing is not a new technology [4]. Recently it becomes more and more popular for on-chip networks due to its low buffer requirements and high scalability.

Wormhole switched networks with virtual-channels are first presented by Dally in [5]. Later Song et al. examined such networks and their applicability for real-time traffic by introducing preemption to the fixed-priority flows [11].

Priority assignment for single processor systems is a widely studied problem. In [12], Liu and Layland presented the Rate Monotonic (RM) priority assignment, and Leung and Whitehead later presented the Deadline Monotonic (DM) priority assignment to fit more general task models [13]. Audsley's optimal priority assignment algorithm [6, 14] guarantees to find a priority assignment in a polynomial number of schedulability tests, if one exists.

In the context of fixed-priority preemptive NoC, Mutka was the first to develop priority assignment algorithms [15]. Lu et al. [16] use contention trees to separate directly and indirectly interfering flows to decide the feasibility of the message set on a NoC. The major work on priority assignment for fixed-priority preemptive NoC was done by Shi and Burns in [7]. They present a branch and bound based algorithm (HSA) to assign priorities to flows. In order to decrease the search space, they introduce heuristic functions to guide the selection of flows. As mentioned in [17], this approach is heuristic but not optimal in the sense that it cannot guarantee to find a valid priority assignment if one exists.

The drawback of most of the works is the assumption that an exclusive priority, i.e. VC, is needed for each message. This means that each router requires enough VCs to accommodate all priority levels. In [18] Nicolić et al. show that this assumption does not hold, and they present an algorithm to reduce the number of needed virtual channels without affecting the schedulability of the systems. Shi and Burns also extend their work [7] to support NoCs with limited virtual-channels by using a priority sharing policy [19].

While fixed-priority preemptive scheduling is well researched, EDF arbitration for wormhole switched NoC was proposed as well [20]. When flows only traverse single hops, EDF is optimal as for single processor scheduling and dominates the other approaches. Similar to fixed-priority preemptive scheduling, the single processor results do not hold anymore when longer message paths are considered. In their experiments it is shown that for flows with multi-hop paths, the HSA assignment by Shi and Burns [7] always performs better.

3. SYSTEM MODEL

The on-chip network considered in this work uses the wormhole-switching technique. The network contains a set of n periodic or sporadic real-time message flows $\mathcal{F} = \{f_1,\ f_2, \ldots\ f_n\}$ $(n \in N^+)$. Each flow f_i can be characterized by $(C_i, T_i, D_i, \mathcal{R}_i)$. C_i represents the basic transmission latency of f_i which is the transmission time without any blocking or interference. T_i denotes the minimum inter-arrival time between two successive instances for sporadic flows, or the period for periodic flows. D_i is the relative deadline of f_i. A flow is schedulable only if its transmission can meet the deadline (i.e. the response time is no larger than the relative deadline). In this paper, we assume that all the flows have constrained deadlines (i.e. $D_i \leq T_i$). Each flow has a fixed transmission route which is denoted as $\mathcal{R}_i$. Moreover, the NoC uses a priority-based preemptive scheduling. We assume that each flow has a unique priority, and a single virtual-channel is assigned to each priority level. In other words, the NoC considered in this work does not use any priority sharing policy.

4. RECAPITULATE THE HEURISTICS-BASED PRIORITY ASSIGNMENT ALGORITHM

4.1 Lower and Upper Bound Analysis

During the priority assignment, two functions are used to assess the degree of schedulability for a partial priority assignment, those functions compute the *lower* and *upper* bounds of the worst-case response times. This notion is introduced in [7]. There are two types of interference which can occur on a NoC, *direct* and *indirect* interference. Assume that f_i and f_j share at least one link, and f_j has higher priority than f_i. The transmission of f_j can cause *direct interference* to f_i. On the other hand, assume that f_i and f_j do not share any links, they can still affect each other, if flow f_m

shares links with both f_i and f_j. Assume that f_j has higher priority than f_m, and f_m has higher priority than f_i. Then f_j can cause direct interference to f_m, which can decrease the minimum inter-arrival times between successive packets of f_m. As a result, f_i can experience extra delay due to the transmission of f_j. In this case, f_j causes indirect interference to f_i. The lower bound only considers the direct interfering flows to compute the worst case traversal time R_i^{LOW} of a flow f_i. For the upper bound, R_i^{UPP}, it is thus assumed that all unassigned flows can possibly interfere with f_i (i.e. both direct and indirect interferences are taken into account).

The computed upper-bounds are needed during the assignment of flows to priority levels. When we try to assign a flow f_i to a priority level we have a number of flows which are already assigned with priorities and a number of unassigned flows. For the lower bound, it is assumed that all unassigned flows which directly interfere with f_i are in the set S_i^D, and indirect interference is ignored in this case. Equation 1 can then be used to compute the lower bound.

$$R_i^{LOW} = \sum_{\forall f_j \in S_i^D} \lceil \frac{R_i^{LOW}}{T_j} \rceil \cdot C_j + C_i \quad (1)$$

where T_i represents the period of f_i, and C_i and C_j denote the basic transmission latency of f_i and f_j, respectively. If a flow f_i can meet its deadline according to the lower bound (i.e. $R_i^{LOW} \leq D_i$), f_i is potentially schedulable with the current priority.

The computation for the upper bound is shown in Equation 2. In addition to the directly interfering flows, all unassigned flows which can cause indirect interference to f_i are also considered. Assume that f_i gets indirect interference through f_j. The indirect interference is captured by an extra jitter (called interference jitter) of f_j, which can have a worst case of $D_j - C_j$.

$$R_i^{UPP} = \sum_{\forall f_j \in S_i^D} \lceil \frac{R_i^{UPP} + D_j - C_j}{T_j} \rceil \cdot C_j + C_i \quad (2)$$

If a flow f_i can meet its deadline according to the upper bound (i.e. $R_i^{UPP} \leq D_i$), f_i is guaranteed to be schedulable with the current priority.

4.2 The Heuristic Search Algorithm

The response time of a NoC flow is dependent on the flows with higher priorities as well as their relative priority ordering [7]. As a result, the well-known optimal priority assignment algorithms for tasks [6][14] cannot be directly applied on NoC flows. Therefore, Shi et al. propose a heuristic priority assignment algorithm for NoC flows [7] (called the Heuristic Search Algorithm (HSA)).

The HSA starts from the lowest priority level. At each priority level, the algorithm uses the *UpperBoundAnalysis* and the *LowerBoundAnalysis* to check the unassigned flows. If a flow f_i can pass the UpperBoundAnalysis at priority level P_k, f_i is guaranteed to be schedulable with P_k. If f_i can pass the LowerBoundAnalysis at priority level P_k, f_i is potential but not guaranteed to be schedulable with P_k. At a certain priority level P_k, the HSA first checks the unassigned flows using the UpperBoundAnalysis. If there exists a flow f_i which can pass the test, this flow will be directly assigned to the current priority level. Then the algorithm stops checking the remaining unassigned flows, and continues with the next priority level[1] (i.e. P_{k+1}). However, if no flow can pass the UpperBoundAnalysis at P_k, the algorithm will check the unassigned flows using the LowerBoundAnalysis. All the flows which can pass the test are potentially schedulable, and they will be added into a candidate list (denoted by CL_k). The algorithm uses a heuristic function to select a suitable candidate, and assigns the selected flow to P_k. Then the HSA continues with P_{k+1}. The above process is repeated until all the flows are assigned with priorities. Then a general schedulability test is applied, in order to check if the current priority assignments are correct. If the test fails, the HSA starts a backtracking process. The algorithm backtracks the priority levels which have multiple candidates (i.e. there are more than one items in the candidate list). When HSA backtracks to a certain priority P_k, all the flows which are assigned to priorities higher than or equal to P_k will be unassigned. Then the algorithm assigns another candidate from CL_k to P_k, and reassigns the remaining unassigned flows using the same procedures as before. If all the priority levels with multiple candidates are checked and the network still fails in the schedulability test, the HSA returns a result of unschedulable.

In [7], it is claimed that if a schedulable priority order for a set of flows exists, the HSA is assured to find it, because the algorithm searches all the possible priority orders. However, this is not true, since the current HSA does miss certain cases. On the other hand, even though the HSA uses heuristic functions and a branch pruning policy to reduce the search space, the algorithm is still not very efficient. The efficiency of the algorithm can be further improved by taking the relations between flows into account. In the following subsections, we present examples to illustrate the above problems of the HSA.

4.3 Missing Cases of The HSA

First, we use a simple example to show that the HSA [7] is not optimal, in the sense that the algorithm may miss certain schedulable cases. In other words, the HSA cannot assure to find a schedulable priority order of a set of NoC flows if it exists.

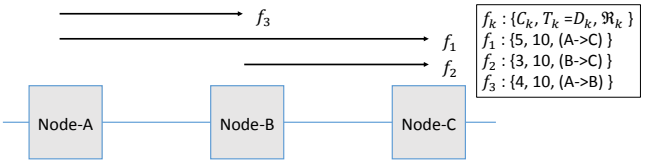

Figure 2: An example showing the missing schedulable cases of the HSA.

Example 1. *Assume that there are three flows in an on-chip network, whose parameters are shown in Figure 2. Now we use the existing HSA to assign priorities to this set of flows.*

P_1**:**

$R_1^{UPP} = \infty > D_1$
$R_2^{UPP} = 13 > D_2$
$R_3^{UPP} = 14 > D_3$
Since no flow can pass the UpperBoundAnalysis, the HSA will apply the LowerBoundAnalysis.
$R_1^{LOW} = \infty > D_1$
$R_2^{LOW} = 8 < D_2$
$R_3^{LOW} = 9 < D_3$
f_2 and f_3 can be added into the candidate list of priority level 1 (i.e. $CL_1 = \{f_2, f_3\}$). Then the algorithm assigns priority P_1 to f_2, and increases the priority level.

P_2**:**

$R_1^{UPP} = 7 < D_1$
Since f_1 can pass the UpperBoundAnalysis, the algorithm directly assigns P_2 to f_1.

[1]The priority level increases as the subscript goes up (i.e. P_1 represents the lowest priority level).

P_3:

The algorithm directly assigns P_3 to f_3.

Schedulability Test:

$R_1 = 7 < D_1$
$R_2 = 13 > D_2$
$R_3 = 4 < D_3$
Since f_2 misses its deadline, the algorithm needs to do the backtracking process.

Backtracking:

The HSA checks the candidate list at each priority level, and tries to reassign the priority to other candidates. In this example, only priority level 1 has other candidates (i.e. f_3). Therefore, the algorithm will unassign f_3, f_1 and f_2 step by step, and then reassign P_1 to f_3. Similar to the earlier steps, the algorithm will assign P_2 to f_1 and P_3 to f_2.

Schedulability Test:

$R_1 = 8 < D_1$
$R_2 = 3 < D_2$
$R_3 = 14 > D_3$
f_3 misses its deadline in this case. Since there is no other candidate at any priority level now, the algorithm will return a result of unschedulable.

However, this result is not correct. The network can be schedulable if we assign P_1 to f_2, P_2 to f_3 and P_3 to f_1 (or switch f_2 and f_3).

The main reason of missing schedulable cases is that the HSA only constructs candidate lists of flows which cannot pass the UpperBoundAnalysis. At a certain priority level P_k, if a flow is found to be able to pass the UpperBoundAnalysis (e.g. f_1 at priority level 2 in Example 1), the algorithm will continue to check the next priority level P_{k+1} without creating a candidate list of P_k. As a result, during the backtracking process of the HSA, the priority level of P_k will be skipped because there is no item in the candidate list (e.g. there is no other candidate at priority level 2 in Example 1). However, if we switch the priorities between f_1 and f_3 during the backtracking process in Example 1, a schedulable priority order can be found. In summary, the incomplete creation of the candidate list in the HSA results in missing of schedulable cases.

In fact, the backtracking process aims to find a schedulable priority ordering by checking other possible assignments at each priority level. Therefore, *the candidate list at a certain priority level of P_k should contain all the flows which are potentially schedulable at P_k (i.e. all the flows which can pass the UpperBoundAnalysis and LowerBoundAnalysis need to be taken into account).*

4.4 Inefficient Backtracking under The HSA

In this section, we use an example to show that the HSA lacks of efficiency. In other words, the HSA contains unnecessary operations which can be avoided directly.

Example 2. *As shown in Figure 3, assume that there are 7 flows in the network. We apply HSA on this set of flows.*

P_1: *P_1 is assigned to f_1, since f_1 can pass the UpperBoundAnalysis.*

P_2: *No flow can pass the UpperBoundAnalysis. After applying the LowerBoundAnalysis, a candidate list is created $CL_2 = \{f_2, f_4, f_7\}$, then P_2 is assigned to f_2.*

P_3: *P_3 is assigned to f_3, because f_3 can pass the UpperBoundAnalysis.*

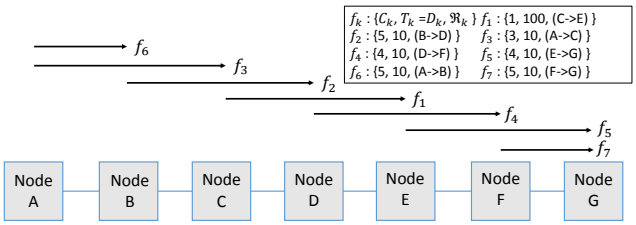

Figure 3: An example showing the unnecessary operations in the HSA.

P_4: *P_4 is assigned to f_6, because f_6 can pass the UpperBoundAnalysis.*

P_5: *No flow can pass the UpperBoundAnalysis. A candidate list is created $CL_5 = \{f_4, f_7\}$, then P_5 is assigned to f_4.*

P_6. *P_6 is assigned to f_7, because f_7 can pass the UpperBoundAnalysis.*

P_7. *P_7 is assigned to f_5.*

Schedulability Test:

After applying the schedulability test, $R_2 = 11 > D_2$.

Backtracking:

HSA backtracks from the highest priority. The algorithm will unassign f_5 and f_7, and goes back to priority level of P_5 which has a nonempty candidate list. Then the algorithm will reassign P_5 to another candidate f_7, and reassign P_6 and P_7 as well.

However, the reassignment of P_4 to f_7 is unnecessary, because this reassignment cannot affect the response time of f_2 at all. As shown in Figure 3, f_4, f_5 and f_7 do not share any links with f_2, so they cannot cause direct interference to f_2. Since f_1 has the lowest priority, f_4, f_5 and f_7 will not cause any indirect interference to f_2 either. As a result, no matter how we change the priority orders of f_4, f_5 and f_7, R_2 will not be affected. Therefore, the backtracking process of the HSA includes unnecessary operations, which makes the algorithm less efficient.

The problem is mainly due to that the HSA does not take the actual relations between flows into account. Since after the schedulability test, we already know which flow misses its deadline, the algorithm can skip the operations which do not reduce the response time of the deadline-missed flow. In summary, the pruning branch policy used in the HSA can be further improved by taking the relations between involved flows into account.

5. IMPROVED PRIORITY ASSIGNMENT OF NOC FLOWS

In this section, we present our undirected-graph based search algorithms for priority assignment of NoC flows (called *GESA* and *GHSA*). The GESA is basically an exhaustive search based solution where we use our undirected-graph based algorithm to decrease the search space by safely pruning branches. Using the GESA, the cases that the HSA misses can be covered. However, the complexity of GESA can be larger than the HSA. Alternatively, in the GHSA, we add the undirected-graph based search algorithm on top of the HSA, in order to further safely decrease the search space. Using the GHSA may not improve the schedulability compared to the HSA, but can improve the efficiency. The main procedure of the

GESA and the GHSA are quite similar. The only difference is the creation of the candidate list which is also the main difference between the HSA and a complete exhaustive search (called the ESA). We will present more details later in this section.

In the beginning, we construct an undirected-dependency-graph including all the related flows. A dependency-graph consists of a number of vertices which are connected by undirected edges (e.g. Figure 4). Each vertex represents a NoC flow, and the edges show the relations between flows. Each pair of connected flows share at least one link in the network. The search algorithm used in our approach is based on such constructed dependency-graphs. For example, the dependency-graph of the flow set presented in Example 2 can be constructed as Figure 4.

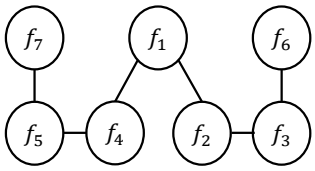

Figure 4: The dependency-graph of the flows described in Example 2.

All the flows are initially marked as *unassigned*. Once a flow is assigned to a certain priority level, this flow will be marked as *assigned*. This flow and its connected links will also be removed from the original dependency-graph, and a set of *independent subgraphs* can then be created.

Theorem 1. *At a certain priority level P, if we assign a flow f_i to P, the sub-dependency-graphs (denoted by $\mathcal{G}_i = \{G_i^1, G_i^2, ..., G_i^n\}$) created by removing f_i and f_i-connected links from the original dependency-graph, are independent from each other.*

Proof. Once we assign f_i to a certain priority level, the remaining flows are still marked as unassigned. Since we always assign priorities in an ascending order, the remaining flows will be assigned with higher priorities than f_i. As a result, f_i cannot cause any direct interference to the remaining unassigned flows.

Assume that f_m is a flow in one of the created sub-dependency-graphs G_i^p ($p \in [1,n]$). No matter how we assign priorities of flows in another sub-dependency-graph G_i^q ($q \in [1,n]$), the flows in G_i^q will not cause any direct interference to f_m, because these flows do not share any link with f_m. Moreover, these flows in G_i^q cannot cause indirect interference to f_m either, since these flows are only related to f_m through f_i while f_i has a lower priority than f_m. Therefore, the priority assignment of flows in G_i^q does not affect f_m. The above proof can be applied on any flow in any sub-dependency-graph in the same manner. This proves that the priority assignment of flows within one sub-dependency-graph is independent from any other sub-dependency-graphs. □

Definition 1. *When the algorithm assigns a flow f_i to the current priority level, f_i is removed from the dependency-graph and a set of* children subgraphs *$\mathcal{G}_i$ are created. f_i is called the **parent** of $\mathcal{G}_i$ and all the flows in $\mathcal{G}_i$. The subgraphs in $\mathcal{G}_i$ are called **sibling-graphs** of each other.*

Theorem 1 implies that when we assign priorities of flows in one dependency-graph, the flows from other *sibling-graphs* can be ignored. For example, if we assign f_1 in Example 2 to a certain priority, f_1 will be removed from the current dependency-graph (as shown in Figure 5). Accordingly, two subgraphs G_1^1 and G_1^2 whose parent is f_1 are created. G_1^1 and G_1^2 are sibling-graphs, and they can be processed independently.

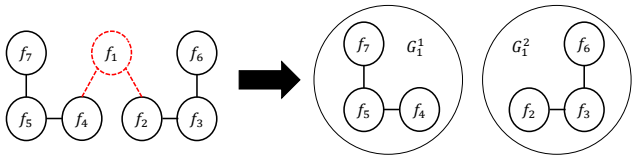

Figure 5: An example of independent subgraphs.

Similar to the HSA, our approach also starts from the lowest priority level. At each priority level, the algorithm needs to create a candidate list, and tries to select a suitable candidate to be assigned. As mentioned earlier, the GESA and the GHSA apply different creation processes of candidate lists.

In GESA: At each priority level, the algorithm (i.e. Alg. 1) first checks each flow in the current dependency-graph using the UpperBoundAnalysis[2]. If a flow can pass the UpperBoundAnalysis, this flow will be added into a candidate-list of this priority level CL_P^U (line 12-13, Alg. 1). For the flows which cannot pass the UpperBoundAnalysis, the algorithm checks them using the LowerBoundAnalysis. If a flow can pass the LowerBoundAnalysis, it will be added into another candidate-list of this priority level CL_P^L (line 14-15, Alg. 1). The GESA checks all the flows in the current dependency-graph at each priority level (line 11, Alg. 1).

In GHSA: At each priority level, the algorithm checks each flow in the current dependency-graph using the UpperBoundAnalysis. Once a flow passes the UpperBoundAnalysis, this flow will be added into the candidate list CL_P^U ((line 20-21). The algorithm then stops checking other flows even if they may also potentially schedulable with the current priority level. In other words, the candidate list CL_P^U only contains one flow which is the first flow passing the UpperBoundAnalysis. This process is the same as used in the HSA. On the other hand, if no flow can pass the UpperBoundAnalysis, the algorithm starts to check all the unassigned flows in the current dependency-graph using the LowerBoundAnalysis. The same as the GESA, all the flows which can pass the LowerBoundAnalysis will be added into the candidate list CL_P^L ((line 27-28).

Once the candidate-lists (i.e. CL_P^U and CL_P^L) are created at a certain priority level P, a suitable candidate will be selected to be assigned to P. If CL_P^U is not empty (i.e. at least one flow can pass the UpperBoundAnalysis), the algorithm will select a candidate from CL_P^U (line 34-35, Alg. 1). A flow which can pass the UpperBoundAnalysis is guaranteed to be schedulable, no matter how the flows with higher priorities are assigned. While choosing a suitable candidate (line 1-5, Alg. 2), we first select the flow with the most connections. This is because removing such a flow from the current dependency-graph can create more subgraphs. As implied from Theorem 1, sibling subgraphs are independent from each other. Therefore, by creating more subgraphs, the search space of the algorithm can be reduced. If several flows have the most connections, a candidate will be selected based on a heuristic function. In [7], the authors present a number of heuristic functions, and an experiment-based comparison is also provided. Once the best candidate f_i is selected, f_i will be removed from the candidate-list. f_i is then as-

[2]The UpperBoundAnalysis can be replaced by any schedulability tests where the effects of indirect interference are deterministically bounded in advance, and the LowerBoundAnalysis can be replaced by any schedulability tests where the effects of indirect interference are ignored. In order to make fair comparison with the HSA, we use the analysis presented in Section 4.1.

Alg. 1 GHSA and GESA

```
1: function main
2: P ← 1; P_p ← 1; backtracking ← False
3: current_graph ← G_root
4: while True do
5:   while P ≤ n do
6:     if backtracking = False then
7:       if P is assigned then
8:         break
9:       end if
10:      if doing an exhaustive search (i.e. for GESA) then
11:        for f_i ∈ current_graph do
12:          if UpperBoundAnalysis(f_i) then
13:            add f_i into CL_P^U
14:          else if LowerBoundAnalysis(f_i) then
15:            add f_i into CL_P^L
16:          end if
17:        end for
18:      else if doing a heuristic search (i.e. for GHSA) then
19:        for f_i ∈ current_graph do
20:          if UpperBoundAnalysis(f_i) then
21:            add f_i into CL_P^U
22:            break
23:          end if
24:        end for
25:        if CL_P^U = ∅ then
26:          for f_i ∈ current_graph do
27:            if LowerBoundAnalysis(f_i) then
28:              add f_i into CL_P^L
29:            end if
30:          end for
31:        end if
32:      end if
33:    end if
34:    if CL_P^U ≠ ∅ then
35:      f_p ← SelectBestCandidate(CL_P^U)
36:    else if CL_P^L ≠ ∅ then
37:      f_p ← SelectBestCandidate(CL_P^L)
38:    else
39:      P ← Backtrack(P, P_p)
40:      unassign priority level P
41:      backtracking ← True
42:      continue
43:    end if
44:    assign f_p to P
45:    backtracking ← False
46:    f_p.interferenceRegion ← size of current_graph − 1
47:    current_graph ← CreateSubgraph(f_p, current_graph)
48:    P++
49:  end while
50:  if SchedulabilityTest() then
51:    return SUCCESS
52:  else
53:    f_m ← get the unschedulable flow
54:    P_p ← get the priority of f_m
55:    P ← P + f_m.interferenceRegion
56:    backtracking ← True
57:  end if
58: end while
59: end function
```

signed to the current priority level, and marked as assigned. At the same time, f_i will also be removed from the current dependency-graph (i.e. all the related connections are removed as well), and a number of sub-graphs whose parent is f_i are created accordingly (line 9-10, Alg. 2). Then the algorithm will try to assign the next priority level to a flow in one of the subgraphs (line 13-16, Alg. 2). However, if f_i is the last flow in the current dependency-graph, no subgraph can be created. Then the algorithm will try to assign the next priority level to a flow in one of the sibling subgraphs of the current dependency-graph (line 11-12, Alg. 2). For example, in Figure 5, after assigning f_1, the algorithm will assign the next priority level to a flow in one of the subgraphs. Assume that the algorithm assigns flows in G_1^1 first. After assigning priorities to f_4, f_5 and f_7, the algorithm then starts to assign priorities to the flows in G_1^2.

On the other hand, if CL_P^U is empty, the algorithm tries to find a candidate from CL_P^L (line 36-37, Alg. 1). A flow in CL_P^L is potentially schedulable, since it can pass the LowerBoundAnalysis. However, the actual schedulability depends on the order of flows with higher priorities. The candidate selection and assignment processes are the same as processing flows in CL_P^U.

Once a flow is assigned to the current priority P, the algorithm will move to the next priority level $P+1$. However, if both candidate-lists of priority P are empty, we can stop the current priority assignment process, because no flow is able to be schedulable at this priority level. In this case, we need to apply a backtrack process, which means that we need to modify the priority assignment of the assigned flows. The algorithm backtracks from the closest lower priority level $P-1$ (line 39, Alg. 1), and tries to assign another candidate. If a candidate is found, the algorithm continues the assignment process as discussed above. However, if no candidate can be found at $P-1$, the algorithm will try to reassign priority $P-2$. When the backtrack process reaches the lowest priority level, and no available candidates can be found, the algorithm returns *UNSCHEDULABLE* which means that no valid priority assignment can be found (line 21-22, Alg. 2).

The above assignment process is repeated until all the flows have been assigned. Then a general schedulability test is applied in order to check if all the flows can meet their deadlines (line 50, Alg.1). If all the flows pass the test, the algorithm terminates and a valid priority assignment has been found (line 51, Alg. 1). Otherwise, we need to apply the backtrack process.

Theorem 2. *Assume that flow f_i misses its deadline after the general schedulability test. If a flow f_j with higher priority than f_i does not belong to any subgraphs whose parent is f_i, changing the priority of f_j cannot affect the response time of f_i at all.*

Proof.
Case 1- f_j and f_i are in the same dependency-graph
In this case, f_j and f_i must be related through other flows. Otherwise, f_j can directly interfere f_i, which means that f_j must be in one of the children subgraphs of f_i. This contradicts to the condition in the theorem.

Assume that f_j and f_i are related through f_p. If f_p has higher priority than f_i, f_j must be in one of the children subgraphs of f_i which conflicts with the condition as well. Therefore, f_p can only have lower priority than f_i. In this case, f_j can cause neither direct nor indirect interference to f_i, which means that f_j cannot affect the response time of f_i.
Case 2- f_j and f_i are not in the same dependency-graph
Theorem 1 has already proved that if two flows are not in the same dependency-graph, they can be processed independently. □

Alg. 2 Utilized Functions

```
1: function SelectBestCandidate(CL)
2: F ← find flows with most connections
3: select f ∈ F based on a heuristic function
4: remove f from CL
5: return f
6: end function
7:
8: function CreateSubgraph(f_p, current_graph)
9:  remove f_i from current_graph
10:  G ← identify remaining independent subgraphs
11: if G = ∅ then
12:  current_graph ← the last graph in unassigned_graphs
13: else
14:  current_graph ← select the largest graph from G
15:  unassigned_graphs ← add remaining graphs from G
16: end if
17: return current_graph
18: end function
19:
20: function Backtrack(P, P_p)
21: if P − 1 < 1 then
22:  return UNSCHEDULABLE
23: else if P == P_p then
24:  P ← FindParent(P)
25: else
26:  P ← P − 1
27: end if
28: return P
29: end function
30:
31: function FindParent(P)
32: tmpP = P
33: while tmpP > 0 do
34:  f_p ← get flow at tmpP
35:  if f_p.interferenceRegion + tmpP ≥ P then
36:  return tmpP
37:  else
38:  tmpP = tmpP − 1
39:  end if
40: end while
41: end function
```

According to Theorem 2, given a certain flow f_i which misses its deadline, the algorithm first needs to check the order of higher priority flows which only belong to the subgraphs whose parent is f_i (line 53-55, Alg.1). The same as the previous backtrack process, at each priority level, the algorithm tries to reassign to other candidates. If there is no available candidate at a certain priority level, the algorithm will backtrack to a lower priority level. The process is repeated until a valid priority level is found, or the algorithm reaches the priority level of f_i. After checking other candidates at the priority level of f_i, if no valid priority order is found, the algorithm will directly continue to check the priority level of the parent of f_i (line 23-24, Alg.2). This is because the priority levels between f_i and its parent must belong to sibling-graphs of f_i. According to Theorem 1, the flows from other sibling-graphs of f_i cannot affect the response time of f_i. Therefore, the priority levels between f_i and its parent can be skipped. An example is given in Figure 6. First, assume that all the priorities have been assigned to the given flows, the schedulability test shows that f_4 misses its deadline. As shown in Figure 6-a, the backtrack process can skip the priority

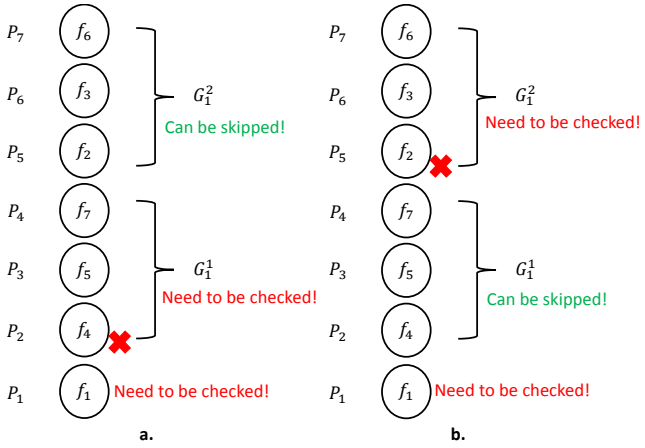

Figure 6: Examples of the backtrack process.

levels of f_2, f_3 and f_6, because they belong to the sibling-graph of G_1^1, which means that they cannot affect the response time of f_4. On the other hand, assume that the schedulability test shows that f_2 misses its deadline. As shown in Figure 6-b, the backtrack process first needs to check the priority levels in G_1^2, and then directly move to the priority level of the parent of f_2. In other words, the priority levels between f_2 and its parent f_1 (i.e. P_2, P_3 and P_4) can be skipped.

If no valid priority order can be found after checking the priority level of the parent of f_i, the algorithm needs to check all the lower priority levels, until a suitable priority order is found or the algorithm reaches the lowest priority level which means that no valid priority order exists.

6. EVALUATION

In order to evaluate the performance of our undirected-graph based search algorithm for priority assignment of NoC flows, we have generated a number of experiments. The experiments focus on the schedulability ratio using our algorithm as well as on the required number of operations[3]. We compare our algorithms (i.e. GESA and GHSA) with the HSA [7] and the exhaustive search solution (denoted as the ESA).

In these experiments, the system uses a 4×4 2D-meshed on-chip network. The flows are transmitted using the XY-routing mechanism, where the sources and destinations are randomly generated. The number of flows is generated from (10, 20, 30), and the average traffic utilization per link is controlled within the range of [0.2,1]. The basic transmission time of each flow is randomly generated from [1, 1000] time unit. The utilization of each flow is randomly generated using the UUnifast-Discard algorithm [21]. Given the basic transmission time C_i and the utilization U_i of each flow, the period is computed as C_i/U_i.

6.1 Experiment Result: Schedulability Ratio

In this section, we present the evaluation results of our algorithms regarding the schedulability ratio for different link utilizations, U_{link}, in the NoC. The experiments are categorized into three groups according to the number of flows in the whole network.

Table 1 shows the experimental results with the setting of 10 flows. The first half of the table presents the results of all the collected samples (more than 4000 samples). The results show that

[3]An operation refers to a unique potential priority order, which should to be checked using a complete schedulability test.

the high and low utilization range yields to similar performance for all four algorithms. Thus, we highlight the intermediate utilization range in smaller granularity in the second halve of Table 1. Here we further show the explicit difference between the evaluated algorithms. It can be observed that, as the average utilization per link goes up, the schedulability ratios of all the the algorithms decrease. The GHSA and the HSA always achieve the same schedulability ratio, while the GESA has the same performance as the ESA. In general, the GESA and the ESA always achieve higher schedulability ratios than the GHSA and the HSA. When the average utilization per link is (0.9-1.0), the GESA and the ESA are around 20%[4] better than GHSA and the HSA.

Table 1: Results regarding the schedulability ratio with the setting of 10 flows. The first column represents the average utilization per link.

U_{link}	GHSA	HSA	GESA	ESA	$\frac{GESA}{v.s.HSA}$	$\frac{GESA}{v.s.ESA}$
0.2-0.5	99.7%	99.7%	99.7 %	99.7%	0	0
0.5-0.8	61.65%	61.65%	62.28%	62.28%	1.03%	0
0.8-1	6.28%	6.28%	7.16%	7.16%	14.06%	0
0.5-0.6	93.89%	93.89%	93.89%	93.89%	0	0
0.6-0.7	60%	60%	60.65%	60.65%	1.08%	0
0.7-0.8	36.03%	36.03%	37.18%	37.18%	3.21%	0
0.8-0.9	11.51%	11.51%	13.01%	13.01%	12.96%	0
0.9-1	2.61%	2.61%	3.14%	3.14%	20.31%	0

In the second group of the experiments, the network contains 20 flows. As shown in Table 2, the GHSA and the HSA still have the same results, but the improvement of the GESA becomes more obvious. When the average utilization per link goes up from 0.6 to 0.9, the improvement (GESA v.s. HSA) increases from 6.97% to 21.84%. Moreover, the GESA also performs slightly better (i.e. around 1%) than the ESA. This is mainly because we set an upper-bound (which is 1000) of the number of operations for the evaluated algorithms. In other words, when the algorithm already finishes 1000 operations, it will terminate and return a result of *UNSCHEDULABLE*. This upper-bound is randomly selected which is just used to limit the time cost of the experiments, since the GESA and the ESA may take long processing time for certain cases. Theoretically, if we do not constrain the number of operations, the GESA and the ESA will achieve the same schedulability ratio. According to the results, the GESA can achieve higher schedulability ratio compared to the ESA, which means that our algorithm can find valid priority orders faster than the ESA.

In the third group of experiments, the total number of flows is 30. As shown in Table 3, the GESA still performs better than the GHSA and the HSA. However, the improvements are not as obvious as the other two groups of experiments. The largest observed improvement is 6.52%. This is because when the total number of flows goes up, the number of possible priority orders also increases dramatically. As a result, for some cases, the GESA reaches the given upper-bound of the number of operations before finding a valid priority order. On the other hand, the improvement of the GESA compared to the ESA becomes more obvious (i.e. the largest observed improvement is 4.26%), because the ESA reaches the upper-bound of the number of operations more frequently.

According to the above experimental results, the GHSA always achieves the same schedulability ratio as the HSA. Using the GESA can result in higher schedulability ratios, because the GESA can cover the missing cases of the HSA. However, the GHSA requires a larger number of operations as a price. In order to limit the processing time of the algorithm, we set an upper-bound of the number of operations. The selection of such upper-bound is a trade-off between the schedulability ratio and the required time cost. A smaller upper-bound can result in shorter processing time but lower schedulability ratio.

[4]The improvement presented in this section is calculated as $(X\ v.s.\ Y) \stackrel{def}{=\!=} \frac{V_X - V_Y}{V_Y}$, where V_X and V_Y represents the results obtained from the algorithm X and Y respectively.

Table 2: Results regarding the schedulability ratio with the setting of 20 flows. The first column represents the average utilization per link.

U_{link}	GHSA	HSA	GESA	ESA	$\frac{GESA}{v.s.HSA}$	$\frac{GESA}{v.s.ESA}$
0.2-0.5	99.98%	99.98%	99.98%	99.98%	0	0
0.5-0.8	50.59%	50.59%	54.76%	54.49%	8.23%	0.51%
0.8-1	5.02%	5.02%	6.10%	6.05%	21.53%	0.79%
0.5-0.6	90%	90%	90.81%	90.81%	0.9%	0
0.6-0.7	61.49%	61.49%	65.78%	65.62%	6.97%	0.25%
0.7-0.8	35.58%	35.58%	40.37%	39.97%	13.46%	1.01%
0.8-0.9	8.81%	8.81%	10.74%	10.65%	21.84%	0.8%
0.9-1	0.28%	0.28%	0.28%	0.28%	0	0

Table 3: Results regarding the schedulability ratio with the setting of 30 flows. The first column represents the average utilization per link.

U_{link}	GHSA	HSA	GESA	ESA	$\frac{GESA}{v.s.HSA}$	$\frac{GESA}{v.s.ESA}$
0.2-0.5	100%	100%	100%	99.95%	0	0.05%
0.5-0.8	51.61%	51.61%	52.31%	52.06%	1.37%	0.49%
0.8-1	2.33%	2.33%	2.48%	2.38%	6.52%	4.25%
0.5-0.6	90.14%	90.14%	93.66%	93.66%	3.91%	0
0.6-0.7	67.86%	67.86%	68.37%	67.86%	0.75%	0.75%
0.7-0.8	33.41%	33.41%	33.78%	33.66%	1.09%	0.36%
0.8-0.9	8.35%	8.35%	8.89%	8.53%	6.52%	4.26%

6.2 Experiment Result: Number of Assignments

In this section, we present the evaluation results regarding the number of operations required by the evaluated algorithms.

Figure 7 shows the experimental results with the setting of 30 flows. When the network utilization is low (e.g. $0.2-0.5$), all the algorithms process quite fast, because valid priority orders can be easily found in the first try without backtracking steps. As the utilization goes up, the algorithms have higher probability to require more backtracking processes. Therefore, the required number of operations increase for all the algorithms. However, we notice that the numbers of operations of the GHSA and the GESA at utilization $[0.8, 1.1]$ are lower than the numbers at utilization $[0.5, 0.8]$. This is mainly because the network has very low schedulability ratio with utilization $[0.8, 1.1]$. In this case, the GHSA and the GESA can determine that the network is unschedulable using a lower number of backtracking steps compared to the cases with utilization $[0.5, 0.8]$. As shown in the results, the GHSA only has a slight variation, which is much smaller than the other three algorithms. The HSA is slower than the GHSA, but still much faster than the GESA and the ESA. Moreover, even though the GESA is based on an exhaustive search, it is still much faster than the ESA. By com-

paring the GHSA with the HSA, and the GESA and the ESA, we can observe that our undirected-graph based solution can obviously decrease the search space.

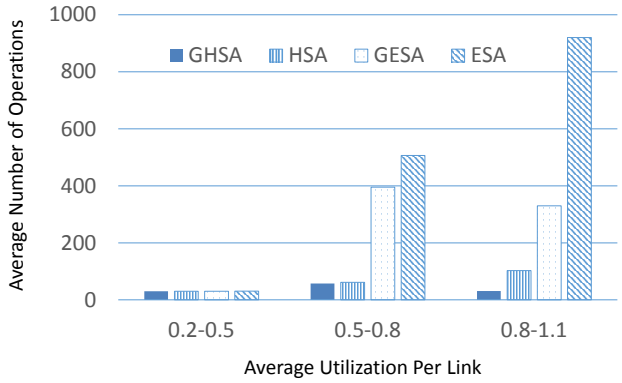

Figure 7: Number of operations with the setting of 30 flows.

We generate another set of experiments to further evaluate the GHSA compared to the HSA regarding the processing efficiency. In this set of experiments, the network is changed to an 8×8 2-D meshed NoC. The number of flows in the network increases to 50 and 80. The rest of experimental settings are the same as earlier.

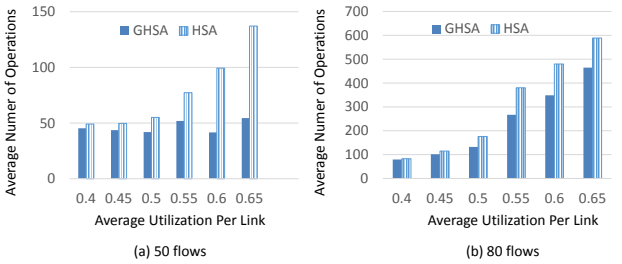

Figure 8: Number of operations with the setting of 50 & 80 flows.

As shown in Figure 8-a (the number of flows is 50), as the network utilization goes up, the required number of operations of the GHSA only gains a slight increase. However, the required operations of the HSA increases much faster. When the network utilization is above 0.6, the difference between the GHSA and the HSA becomes very obvious. In Table 4, we further show the details of the improvement. The improvement of each experiment is represented as V_{HSA}/V_{GHSA}, which shows how much faster the GHSA is compared to the HSA. When the link utilization is [0.4-0.45), the HSA requires 1.34 times more operations than the GHSA on average, while for certain cases the HSA requires 8.8 times more operations. When the link utilization is [0.6-0.65), the GHSA becomes 3.73 times faster than the HSA on average.

The results of the experiments with the setting of 80 flows are shown in Figure 8-b. By comparing Figure 8-a and Figure 8-b, we can observe that as the number of flows in the network increases, both the GHSA and the HSA require more operations, since the search space becomes much larger. In this set of experiments, the GHSA is still much faster than the HSA. As presented in Table 4, as the link utilization increases from 0.4 to 0.65, the improvement of the GHSA increases from 1.06 to 2.6 on average. However, we also notice that the improvement in the experiments with 80 flows are not as obvious as the experiments with 50 flows. This is mainly

Table 4: Results showing the improvements of the GHSA compared to the HSA regarding the efficiency.

50 flows			80 flows		
U_{link}	MEAN	MAX	U_{link}	MEAN	MAX
0.4-0.45	1.34	8.8	0.4-0.45	1.06	5.67
0.45-0.5	1.44	6.83	0.45-0.5	1.17	11
0.5-0.55	1.89	14.33	0.5-0.55	1.57	26.34
0.55-0.6	2.53	30.53	0.55-0.6	2.33	32.29
0.6-0.65	3.73	66.73	0.6-0.65	2.60	29.44

caused by the setting of the upper-bound of operations (i.e. 1000). When the number of flows increases to 80, the search space becomes much larger. In this case, both the GHSA and the HSA have high probability to reach the specified upper-bound, which results in equal performance (i.e. the improvement is 1). Such cases occur less frequent in the experiments with 50 flows, due to the smaller search space.

According to the above experiment, the GESA and the ESA can achieve higher schedulability ratio than the GHSA and the HSA. However, the GESA and the ESA require much more operations as a price. Even though the GESA already becomes much faster than the ESA due to the use of the undirect-graph based search algorithm, the large number of operations may still limit its applicability in reality. On the other hand, the GHSA and the HSA have much better scalability with little sacrifice of the schedulability. As the undirect-graph based algorithm can significantly decrease the search space, the GHSA can be much more efficient compared to the HSA. In general, we can conclude that the GHSA is faster than the HSA while achieving the same schedulability ratios.

7. CONCLUSION AND FUTURE WORKS

In this paper, we present two algorithms (the GHSA and the GESA) for priority assignment of messages in wormhole switched NoC. Moreover we point out the drawbacks of the existing heuristic priority assignment algorithm for on-chip communications (called HSA): missing cases and inefficiency. In the proposed algorithms, we introduce an undirected-graph based search solution, where we take the dependencies between messages into account. Such solution can safely decrease the search space thus improve the efficiency of the algorithms. A number of experiments have been generated. The results show that our proposed algorithm GHSA can be much faster than the HSA while achieving the same schedulability ratio. On other hand, the GESA, which can achieve higher schedulability compared to the HSA, is slower than the HSA but much faster than the ESA.

Looking at future work, the proposed algorithms assume distinct priorities, which means that each flow has a unique priority. Priority-based preemptions are achieved by virtual-channels, and each priority level requires a single virtual-channel. However, most of the existing NoC implementations have a limited number of virtual-channels. As a result, in order to afford more flows in an on-chip network, multiple flows need to share the same priority level. Therefore, we need to extend our algorithm to take priority sharing policies into account (e.g. [18][19]) as well.

8. REFERENCES

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

[2] *Epiphany Architecture Reference*, Adapteva Inc., Adapteva Inc. 1666 Massachusetts Ave, Suite 14 Lexington, MA 02420 USA, 2012.

[3] Tilera, *Tile processor: user architecture manual*, 2011. [Online]. Available: www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf

[4] L. Ni and P. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, vol. 26, no. 2, pp. 62–76, 1993.

[5] W. Dally, "Virtual-channel flow control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, 1992.

[6] N. Audsley and Y. Dd, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," YCS164, Dept. Computer Science, University of York, 1991.

[7] Z. Shi and A. Burns, "Priority assignment for real-time wormhole communication in on-chip networks," in *29th IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 421–430.

[8] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed services of the NoC of a manycore processor," in *International Workshop on Network on Chip Architectures (NoCArc)*, 2014, pp. 11–16.

[9] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Conference on Design, Automation & Test in Europe (DATE)*, 2014.

[10] V. Nélis, P. M. Yomsi, L. M. Pinho, J. C. Fonseca, M. Bertogna, E. Quiñones, R. Vargas, and A. Marongiu, "The Challenge of Time-Predictability in Modern Many-Core Architectures," in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), vol. 39, 2014, pp. 63–72.

[11] H. Song, B. Kwon, and H. Yoon, "Throttle and preempt: a new flow control for real-time communications in wormhole networks," in *International Conference on Parallel Processing*, 1997, pp. 198–202.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[13] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237 – 250, 1982.

[14] N. C. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[15] M. W. Mutka, "Using rate monotonic scheduling technology for real-time communications in a wormhole network," in *2nd Workshop on Parallel and Distributed Real-Time Systems*, 1994, pp. 194–199.

[16] Z. Lu, A. Jantsch, and I. Sander, "Feasibility analysis of messages for on-chip networks using wormhole routing," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2005.

[17] R. I. Davis and A. Burns, "On optimal priority assignment for response time analysis of global fixed priority pre-emptive scheduling in multiprocessor hard real-time systems," *University of York, UK, Tech. Rep YCS-2010-451*, 2010.

[18] B. Nikolić, H. I. Ali, S. M. Petters, and L. M. Pinho, "Are virtual channels the bottleneck of priority-aware wormhole-switched NoC-based many-cores?" in *21st International Conference on Real-Time Networks and Systems (RTNS)*, 2013.

[19] Z. Shi and A. Burns, "Real-time communication analysis with a priority share policy in on-chip networks," in *21st Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2009, pp. 3–12.

[20] B. Nikolić and S. M. Petters, "EDF as an arbitration policy for wormhole-switched priority-preemptive NoCs: Myth or fact?" in *14th International Conference on Embedded Software (EMSOFT)*, 2014, pp. 28:1–28:10.

[21] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *30th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2009, pp. 398–409.